Visual Studio 2005 JScript

JScript

JScript 8.0 は、多様なアプリケーションを備えた最新のスクリプト言語です。本格的なオブジェクト指向言語でありながら、"スクリプト" としての感覚を維持しています。JScript 8.0 は、JScript の以前のバージョンと完全な下位互換性を維持する一方で、新しい機能を組み込み、共通言語ランタイムと .NET Framework へのアクセスを提供します。

次のトピックでは、JScript 8.0 の基本的なコンポーネントを紹介し、言語の使用方法についての情報を示します。最新のプログラミング言語と同様、JScript は、多くの一般的なプログラミング構造と言語要素をサポートします。

他の言語でプログラムを作成した経験がある場合、このセクションで説明されている内容の多くが、既知の内容である可能性もあります。プログラミング構造の多くは以前のバージョンの JScript と同じですが、JScript 8.0 では、他のクラス ベースのオブジェクト指向言語と同様の強力な新しい構造が導入されています。

プログラミングの経験がない場合、このセクションの内容は、コードを記述するときの基本になります。基本を理解すると、JScript を使用して強力なスクリプトとアプリケーションを作成できます。

このセクションの内容

JScript について

JScript 8.0 の新機能を紹介します。

JScript コードの作成、コンパイル、およびデバッグ

JScript 8.0 でコードを作成、編集、およびデバッグする方法を説明するトピックへのリンクを示します。

JScript での情報の表示

コマンド ライン プログラム、ASP.NET、およびブラウザでの情報の表示方法を説明するトピックへのリンクを示します。

正規表現の概説

JScript 8.0 の正規表現に関する要素と手順について説明します。正規表現の概念、正しい構文、および適切な使用方法について説明します。

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

関連するセクション

Devenv コマンド ライン スイッチ

Visual Studio を起動する方法、およびコマンド プロンプトからビルドを実行する方法を説明する、言語リファレンス トピックの一覧を示します。

各言語の比較

Visual Basic、C++、C#、および JScript の、キーワード、データ型、演算子、およびプログラム可能なオブジェクト (コントロール) を比較します。

.NET Framework クラス ライブラリ リファレンス

.NET Framework クラス ライブラリの名前空間の説明を含むトピックへのリンクを示し、クラス ライブラリ ドキュメントの使用方法について説明します。

Visual Studio のコマンドおよびスイッチ

コマンド ウィンドウおよび [検索] ボックスから、コマンドを使用して IDE とやり取りする方法を説明する、言語リファレンス トピックの一覧を示します。

Visual Studio に関するチュートリアル

特定のアプリケーション開発に伴う手順、および主要なアプリケーション機能を使用する方法を説明するトピックへのリンクを示します。

JScript について

JScript 8.0 では、JScript 言語からさまざまな面が進化しています。Visual Studio 開発環境との統合、多様な新しい機能、および .NET Framework クラスへのアクセスを考えると、JScript から JScript 8.0 への移行は最初は難しいと感じられます。

実際には、ほとんどの変更は機能の追加であり、JScript の中心的な機能はこれまでと同じであることがわかります。実際に、すべての JScript スクリプトは、変更を加えなくても JScript 8.0 で (高速モードをオフにして) 動作します。高速モード (ASP.NET でサポートされるモード) で実行する場合は、スクリプトにわずかな変更が必要な場合もあります。

コードに新しい機能を段階的に追加できるため、JScript から JScript 8.0 への移行は簡単です。JScript 8.0 についてより詳しく学習してから新しい機能を追加し、スクリプトを自分のペースでアップグレードできます。

次のドキュメントは、既存のアプリケーションをアップグレードし、JScript 8.0 で加えられた変更を理解するために役立ちます。

このセクションの内容

JScript 8.0 の概要

JScript 8.0 の総合的な概要、ECMAScript との関連、および前バージョンの JScript からの変更点について説明します。

JScript 8.0 の新機能

JScript 8.0 の新しい機能の一覧を示します。ECMAScript Edition 4 と共に開発された機能や、ECMAScript にはない追加機能が含まれています。

JScript 8.0 の概要 (JScript プログラマ対象)

JScript と JScript 8.0 の違いを説明します。経験を積んだ JScript プログラマは、背景情報をすばやく理解できます。

JScript バージョンの Hello World! プログラム

JScript 8.0 での Hello World! プログラムの作成方法を説明します。

前のバージョンの JScript で作成されたアプリケーションのアップグレード

既存の JScript アプリケーションを JScript 8.0 で動作するようにアップグレードする方法を説明します。

以前のバージョンの共通言語ランタイムでの JScript アプリケーションの実行

あるバージョンのランタイムで作成された JScript 8.0 アプリケーションを、以前のバージョンのランタイムで実行できるように構成する方法を説明します。

JScript プログラマのための追加の技術情報

JScript プログラミングの一般的な問題に対する回答を提供するサイトおよびニュースグループを示します。

関連するセクション

JScript のバージョン情報

JScript の機能をすべて一覧に示し、それぞれの機能が導入されたバージョンを示します。

JScript 言語の紹介

JScript コードを記述するための要素やプロシージャを紹介します。また、言語要素やコード構文の背景についての詳細な説明へのリンクも示します。

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

JScript 8.0 の概要

JScript 8.0 は、Microsoft による ECMA 262 言語の新たな実装です。JScript 8.0 では、これまでの JScript の機能にクラス ベース言語の機能を組み合わせることで、両方の最適な部分を提供しています。ECMAScript Edition 4 と共に開発されている JScript 8.0 では、コンパイル済みコード、型指定された変数と型指定されていない変数、遅延バインディングおよび事前バインディングされたクラス (継承、関数のオーバーロード、プロパティ アクセサなどを含む)、パッケージ、言語間サポート、および .NET Framework への完全なアクセスを利用できるようになりました。

新機能

JScript 8.0 は、真のオブジェクト指向スクリプト言語です。JScript 8.0 は、クラス、型、その他信頼性の高いアプリケーションを記述するための高度な言語機能を利用できるようになりましたが、型宣言の不要なプログラミング、expando 関数とクラス、**eval** を使用した動的なコード実行などをサポートしており、"スクリプト" 感覚はこれまでと変わりありません。

JScript 8.0 は、型宣言が不要な言語であるだけでなく、型指定の厳密な言語にもなりました。以前のバージョンの JScript では、型制限の緩い構造により、変数の型は代入された値の型になりました。現実には、以前のバージョンでは変数の型を宣言できませんでした。JScript 8.0 はこれまでの JScript よりも柔軟で、変数に型の注釈を指定できます。変数は特定のデータ型にバインドされ、変数は指定した型のデータしか保存できなくなります。

プログラミング言語で、型指定が厳密であることには多くの利点があります。使用しているデータに適切なデータ型を使用できることの他にも、次のような利点があります。

- 実行速度の向上
- 実行時/コンパイル時の型チェック
- わかりやすいコード

JScript 8.0 は、他のプログラミング言語を凝縮したものではなく、別の言語のコンパクト バージョンでもありません。多様なアプリケーションを備えた最新のスクリプト言語です。

メモ

紹介されている JScript 8.0 のコード例の多くは、理解しやすいように実際のスクリプト コードよりも明示的で簡潔にしてあります。また、概念的な説明を目的としているため、最適なコーディング方法やコーディング スタイルについての説明は省略しています。実際にコードを記述するときには、読みやすくわかりやすいコードを記述するようにしてください。

参照

その他の技術情報

JScript リファレンス

JScript 8.0 の新機能

JScript 8.0 は、Microsoft JScript の後継言語として位置付けられ、Web の言語を使用して Microsoft .NET プラットフォームにすばやく簡単にアクセスできるようにデザインされています。JScript 8.0 の主な役割は、ASP.NET を使用した Web サイトの構築と、.NET Framework のスクリプトを使用したアプリケーションのカスタマイズです。

ECMAScript 規格と互換性のある JScript 8.0 には、コンパイル済みコード、共通言語仕様 (CLS: Common Language Specification) に準拠した言語間サポート、.NET Framework へのアクセスなど、ECMAScript で指定されていない機能も追加されています。
Visual Studio .NET 2002 に含まれている JScript .NET では .NET Framework に固有のセキュリティを利用していましたが、
JScript .NET 2003 では、**eval** メソッドの制限付きセキュリティ コンテキストを追加することにより、セキュリティを維持しています。

JScript 8.0 の新機能のいくつかは、CLS を利用します。CLS は、データ型、オブジェクトを公開する方法、オブジェクトを相互運用する方法などを標準化するための規則です。CLS に準拠した言語では、JScript 8.0 で作成したクラス、オブジェクト、およびコンポーネントを使用できます。JScript 開発者は CLS 準拠のプログラミング言語から、データ型などの言語に固有な相違を考慮することなく、クラス、コンポーネント、およびオブジェクトにアクセスできます。JScript 7.0 プログラムが使用する CLS 機能には、名前空間、属性、参照渡しによるパラメータ、およびネイティブな配列があります。

JScript .NET および JScript 8.0 の新機能のいくつかを次に示します。

JScript 8.0 の新機能

/platform コンパイラ オプション

/platform オプションは、出力ファイルがターゲットとするプロセッサの種類を指定するために使用します。x86 は、32 ビットの Intel 互換プロセッサを指定します。Itanium は、Intel の 64 ビット プロセッサを指定します。x64 は、AMD の 64 ビット プロセッサを指定します。既定値 (anycpu) を使用すると、あらゆるプラットフォームで出力ファイルを実行できるようになります。

JScript .NET 2003 の新機能

eval メソッドの制限付きセキュリティ コンテキスト

セキュリティを維持するため、組み込みの **eval** メソッドは、呼び出し元のアクセス許可に関係なく、制限付きセキュリティ コンテキストでスクリプトを既定で実行します。省略可能な 2 番目のパラメータに "unsafe" を指定して **eval** を呼び出すと、スクリプトは呼び出し元のアクセス許可で実行されます。これにより、ファイル システム、ネットワーク、またはユーザー インターフェイスにアクセスできます。詳細については、「eval メソッド」を参照してください。

JScript .NET 2002 の新機能

クラス ベースのオブジェクト

JScript .NET は、JScript と同様に、プロトタイプ ベースのオブジェクトによって継承をサポートします。JScript .NET では、オブジェクトのデータと動作を定義するクラスを宣言できるようにすることで、クラス ベースのオブジェクトもサポートしています。JScript .NET で作成されたクラスは、他の .NET 言語で使用および拡張できます。クラスは、基本クラスのプロパティとメソッドを継承できます。いくつかの属性は、動作と参照可能範囲を変更するクラスおよびクラス メンバに適用できます。詳細については、「クラス ベースのオブジェクト」を参照してください。

JScript のデータ型

JScript .NET では、JScript と同様に、変数のデータ型を指定せずにプログラムを記述できます。また、すべての変数を特定のデータ型に連結する、型指定の厳密な言語として使用したり、型指定された変数と型指定されていない変数を混在させることもできます。JScript .NET では、多くの新しいデータ型が用意されています。クラスと .NET 型もデータ型として使用できます。詳細については、「JScript のデータ型」を参照してください。

条件付きコンパイル

ディレクティブは、JScript .NET プログラムのコンパイルを制御します。たとえば @debug ディレクティブは、スクリプトの特定の部分に対して、デバッグ情報の出力をオンまたはオフにします。詳細については、「@debug ディレクティブ」を参照してください。@position ディレクティブは、デバッガ用に現在の行の行番号を設定します。詳細については、「@position ディレクティブ」を参照してください。これらのディレクティブはどちらも、他のスクリプトに組み込まれるコードを記述する場合に役立ちます。詳細については、「条件付きコンパイル」を参照してください。

JScript の名前空間

名前空間を使用すると、クラス、インターフェイス、およびメソッドが階層で整理され、名前の競合が起こらなくなります。JScript .NET では、独自の名前空間を定義できます。JScript .NET から、独自に定義した名前空間を含め、任意の .NET Framework 名前空間にもアクセスできます。package ステートメントを使用すると、開発が簡単になるように関連するクラスをパッケージ化して、名前の競合を避けることができます。詳細については、「package ステートメント」を参照してください。import ステートメントは、スクリプトが .NET Framework 名前空間を利用できるようにして、スクリプトが名前空間のクラスおよびインターフェイスにアクセスできるように、します。詳細については、「import ステートメント」を参照してください。

JScript の変数と定数

JScript .NET には、定数値を表す識別子を定義する、const ステートメントが用意されています。詳細については、「JScript の変数と定数」を参照してください。

列挙型

JScript .NET には、列挙型を作成できる enum ステートメントが用意されています。列挙型を使用して、データ型の値にわかりやすい名前を指定できます。詳細については、「enum ステートメント」を参照してください。

参照

概念

JScript プログラマのための追加の技術情報
Visual Basic 言語の新機能

その他の技術情報

Visual Studio 2005 の新機能
修飾子
データ型 (JScript)
ディレクティブ
ステートメント
JScript リファレンス

JScript 8.0 の概要 (JScript プログラマ対象)

ここに示す情報は、既に JScript でのプログラミング経験があり、JScript 8.0 で導入された新しい機能について学習することを考えているプログラマを対象としています。

一般的なタスク

プログラムのコンパイル方法

JScript 8.0 コマンド ライン コンパイラは、JScript プログラムから実行可能ファイルとアセンブリを作成します。詳細については、「方法 : コマンド ラインで JScript コードをコンパイルする」を参照してください。

"Hello World" プログラムの作成方法

JScript 8.0 バージョンの "Hello World" プログラムは簡単に作成できます。詳細については、「JScript バージョンの Hello World! プログラム」を参照してください。

データ型の使用方法

JScript 8.0 では、コロンは変数宣言または関数定義の型を指定します。既定の型は **Object** で、この型は任意の型を保持できます。詳細については、「JScript の変数と定数」および「JScript の関数」を参照してください。

JScript 8.0 には、いくつかの組み込みデータ型 (**int**、**long**、**double**、**String**、**Object**、**Number** など) があります。詳細については、「JScript のデータ型」を参照してください。適切な名前空間をインポートすると、.NET Framework データ型も使用できます。詳細については、「.NET Framework クラス ライブラリ リファレンス」を参照してください。

名前空間のアクセス方法

名前空間には、**import** ステートメント (コマンド ライン コンパイラを使用する場合) または **@import** ディレクティブ (ASP.NET を使用する場合) を使ってアクセスします。詳細については、「import ステートメント」を参照してください。**/autoref** オプション (既定でオンになっています) を指定すると、JScript .NET プログラムで使用されている名前空間に対応するアセンブリを自動的に参照します。詳細については、「/autoref」を参照してください。

型指定された (ネイティブ) 配列の作成方法

型指定された配列のデータ型は、データ型の名前の後に角かっこ ([]) を指定して宣言します。**Array** コンストラクタで作成した、JScript 配列オブジェクトも使用できます。詳細については、「配列の概要」を参照してください。

クラスの作成方法

JScript 8.0 では、独自のクラスを定義できます。クラスには、メソッド、フィールド、プロパティ、静的な初期化子、およびサブクラスを含めることができます。完全に新しいクラスを作成したり、既存のクラスやインターフェイスを継承したりできます。修飾子は、クラス メンバの参照可能範囲、メンバの継承、およびクラス全体の動作を制御します。カスタム属性も使用できます。詳細については、「クラス ベースのオブジェクト」および「JScript の修飾子」を参照してください。

参照

概念

前のバージョンの JScript で作成されたアプリケーションのアップグレード

その他の技術情報

JScript について

JScript バージョンの Hello World! プログラム

次のコンソール プログラムは、"Hello World!" プログラムの JScript バージョンです。このプログラムでは、`Hello World!` という文字列が表示されます。

例

```
// A "Hello World!" program in JScript.
print("Hello World!");
```

このプログラムでは次の点が重要です。

- コメント
- 出力
- コンパイルと実行

コメント

この例の 1 行目はコメントです。コンパイラはコメントを無視するため、コメントにはテキストを自由に記述できます。ここでは、プログラムの目的を説明しています。

```
// A "Hello World!" program in JScript.
```

2 つのスラッシュ (`//`) は、その行のスラッシュ以降の部分がコメントであることを表します。1 行全体をコメントにすることも、次に示すように、他のステートメントの最後にコメントを追加することもできます。

```
var area = Math.PI*r*r;  // Area of a circle with radius r.
```

複数行のコメントも使用できます。複数行コメントは、スラッシュとアスタリスク (`/*`) で始まり、その 2 つを逆に組み合わせた記号 (`*/`) で終わります。

```
/*
Multiline comments allow you to write long comments.
They can also be used to "comment out" blocks of code.
*/
```

詳細については、「JScript のコメント」を参照してください。

出力

この例では、**print** ステートメントを使用して "`Hello World!`" という文字列を表示しています。

```
print("Hello World!");
```

詳細については、「print ステートメント」を参照してください。

ユーザーにメッセージを表示するには別の方法もあります。クラス System.Console では、コンソールを使ったユーザーとのやり取りを簡単にするためのメソッドとプロパティが公開されています。詳細については、「**Console**」を参照してください。クラス System.Windows.Forms.MessageBox では、Windows フォームを使ったユーザーとのやり取りを簡単にするためのメソッドとプロパティが公開されています。詳細については、「System.Windows.Forms」を参照してください。

コンパイルと実行

コマンド ライン コンパイラを使用して、"Hello World!" プログラムをコンパイルできます。

コマンド ラインからプログラムをコンパイルおよび実行するには

1. テキスト エディタを使ってソース ファイルを作成し、Hello.js などのファイル名で保存します。
2. コンパイラを起動するには、コマンド プロンプトで次のように入力します。

```
jsc Hello.js
```

メモ

プログラムは、Visual Studio コマンド プロンプトからコンパイルしてください。詳細については、
「方法 : コマンド ラインで JScript コードをコンパイルする」を参照してください。

プログラムにコンパイル エラーがない場合は、コンパイラによって Hello.exe ファイルが作成されます。

3. プログラムを実行するには、コマンド プロンプトで次のコマンドを入力します。

```
Hello
```

参照

その他の技術情報

JScript コードの作成、コンパイル、およびデバッグ
JScript での情報の表示
JScript リファレンス

前のバージョンの JScript で作成されたアプリケーションのアップグレード

既存の JScript コードのほとんどは、さまざまな点が強化された JScript 8.0 でも適切に動作します。JScript 8.0 は、前のバージョンの JScript に対してほとんど完全な下位互換性を持ちます。JScript 8.0 の新しい機能は、新しい領域を拡張しています。

既定では、JScript 8.0 プログラムは "高速モード" でコンパイルされます。高速モードではコードにいくつかの制限があるため、プログラムはより効率的になり、実行速度が速くなります。ただし、以前のバージョンで利用できた機能の一部は、高速モードでは利用できません。ほとんどの場合、これらの機能はマルチスレッド アプリケーションと互換性がなく、非効率的なコードを生成します。コマンド ライン コンパイラでコンパイルされるプログラムの場合は、高速モードをオフにして、下位互換性を完全に維持できます。このようにしてコンパイルされたコードは処理速度が遅く、エラーが発生しやすくなります。ASP.NET アプリケーションでは、安定性の問題があるため、高速モードをオフにできません。詳細については、「/fast」を参照してください。

高速モード

高速モードでは、JScript は次のように動作します。

- すべての変数を宣言する必要があります。
- 関数は定数になります。
- 組み込みオブジェクトに拡張プロパティは指定できません。
- 組み込みオブジェクトでは、プロパティを一覧で示したり変更したりできません。
- arguments オブジェクトは利用できません。
- 読み取り専用の変数、フィールド、またはメソッドには代入できません。
- eval メソッドは外側のスコープで識別子を定義できません。
- eval メソッドは制限付きのセキュリティ コンテキストでスクリプトを実行します。

すべての変数を宣言する必要があります

以前のバージョンの JScript では、変数の明示的な宣言は必要ありませんでした。この場合、プログラマの入力量は少なくなりますが、エラーを追跡するのは困難になります。たとえば、スペルを間違っている変数に値を代入する可能性があります。この場合、エラーも生成されず、目的の結果も得られません。また、宣言されていない変数はグローバル スコープを持ち、さらに混乱を広げる原因となります。

高速モードでは、変数の明示的な宣言が要求されます。これにより、上で説明したようなエラーの発生を防止でき、より高速に実行されるコードを生成できます。

JScript .NET では、型の注釈を指定した変数もサポートされます。各変数は特定のデータ型に連結され、変数は指定した型のデータしか格納できなくなります。型の注釈は必須ではありませんが、指定することにより、変数に間違ったデータを格納することによるエラーを防止して、プログラムの実行速度を向上できます。

詳細については、「JScript の変数と定数」を参照してください。

関数は定数になります

以前のバージョンの JScript では、**function** ステートメントを使って宣言された関数は、**Function** オブジェクトを保持する変数と同じように処理されました。特に、関数の識別子は、任意の型のデータを格納するための変数として使用できました。

高速モードでは、関数は定数になります。したがって、関数に新しい値を代入したり、関数を再定義したりできません。これにより、関数の意味が変更されることを防ぎます。

関数の変更が必要なスクリプトの場合は、**Function** オブジェクトのインスタンスを保持する変数を明示的に使用できます。ただし、**Function** オブジェクトは処理が低速です。詳細については、「Function オブジェクト」を参照してください。

組み込みオブジェクトに expando プロパティを指定できません

以前のバージョンの JScript では、組み込みオブジェクトにも expando プロパティを追加できました。この機能は、**String** オブジェクトにメソッドを追加して、文字列の先頭の空白を削除する場合などに使用できます。

高速モードでは、この操作は実行できません。この機能に依存しているスクリプトは変更が必要です。オブジェクトに関数をメソッドとして追加する代わりに、関数をグローバル スコープで定義できます。その後、オブジェクトから expando メソッドが呼び出されているスクリプトで、各インスタンスを作成し直し、オブジェクトが適切な関数に渡されるようにします。

Global オブジェクトは例外で、expando プロパティを指定できます。グローバル スコープの識別子はすべて、**Global** オブジェクトのプロパティで

す。したがって、新しいグローバル変数の追加をサポートするために、**Global** オブジェクトは動的に拡張できる必要があります。

組み込みオブジェクトでは、プロパティをリストしたり変更したりできません

以前のバージョンの JScript では、組み込みオブジェクトの定義済みのプロパティに対して削除、列挙、および書き込みを行うことができました。たとえば、**Date** オブジェクトの既定の **toString** メソッドを変更できました。

高速モードでは、この操作は実行できません。組み込みオブジェクトに expando プロパティは指定できないため、この機能は不要になりました。各オブジェクトのプロパティは、参照セクションにリストされます。詳細については、「オブジェクト」を参照してください。

arguments オブジェクトは利用できません

以前のバージョンの JScript では、関数定義内部に **arguments** オブジェクトが用意されていました。これにより、関数は任意の数の引数を受け取ることができました。**arguments** オブジェクトは、呼び出し元の関数だけでなく、現在の関数への参照も提供しました。

高速モードでは、**arguments** オブジェクトは利用できません。ただし JScript 8.0 では、関数宣言の関数パラメータ リストに "パラメータ配列" を指定できます。これにより、関数は任意の数の引数を受け取ることができます。**arguments** オブジェクトの一部の機能は、このように置き換えられます。詳細については、「function ステートメント」を参照してください。

高速モードでは、現在の関数または呼び出し元の関数に対して、直接アクセスおよび参照を行う方法はありません。

読み取り専用の変数、フィールド、またはメソッドには代入できません

以前のバージョンの JScript では、読み取り専用の識別子に値を代入するステートメントを記述できました。この場合、代入は通知なしで失敗します。代入が失敗したことは、値が実際に変更されたかどうかを確認しなければわかりませんでした。読み取り専用の識別子への代入は、通常は間違いの結果であり、効果はありません。

高速モードでは、読み取り専用の識別子に値を代入しようとすると、コンパイル エラーが発生します。代入を削除するか、読み取り専用でない識別子に代入します。

高速モードをオフにした場合、読み取り専用の識別子への代入は実行時に通知されずに失敗しますが、コンパイル時に警告が生成されます。

eval メソッドは外側のスコープで識別子を定義できません

以前のバージョンの JScript では、**eval** メソッドを呼び出すことで、関数や変数をローカルまたはグローバル スコープで定義できました。

高速モードでは、関数や変数を **eval** メソッドの呼び出しで定義できますが、その呼び出しでしかアクセスできません。**eval** が終了すると、**eval** 内部で定義された関数や変数にはアクセスできなくなります。**eval** 内で実行された計算の結果は、現在のスコープでアクセスできる変数に代入できます。**eval** メソッドの呼び出しは低速です。このメソッドを含むコードは、作成し直すことをお勧めします。

高速モードがオフの場合、**eval** メソッドの動作はこれまでと同じになります。

eval メソッドは制限付きのセキュリティ コンテキストでスクリプトを実行します

以前のバージョンの JScript では、**eval** メソッドに渡されるコードが、呼び出し元のコードと同じセキュリティ コンテキストで実行されます。

ユーザーを保護するため、**eval** メソッドに渡されるコードは、文字列 "unsafe" が 2 番目のパラメータとして渡されていない限り、制限付きのセキュリティ コンテキストで実行されます。制限付きのセキュリティ コンテキストでは、ファイル システム、ネットワーク、ユーザー インターフェイスなどのシステム リソースへのアクセスが禁止されます。これらのリソースにアクセスしようとすると、セキュリティ例外が生成されます。

eval の 2 番目のパラメータが文字列 "unsafe" である場合、**eval** メソッドに渡されるコードは、呼び出し元のコードと同じセキュリティ コンテキストで実行されます。これにより、**eval** メソッドの以前の動作が復元されます。

セキュリティに関するメモ
eval は、既知のソースから取得したコード文字列を実行する場合だけ、unsafe モードで使用してください。

参照

関連項目

/fast

概念

JScript 8.0 の概要 (JScript プログラマ対象)

その他の技術情報

JScript について

以前のバージョンの共通言語ランタイムでの JScript アプリケーションの実行

特に指定されていない限り、JScript アプリケーションは、コンパイラがアプリケーションの作成に使用する共通言語ランタイムのバージョンで実行するように作成されます。ただし、あるバージョンのランタイムで作成された .exe または ASP.NET Web アプリケーションを、任意のバージョンのランタイムで実行することはできます。

他のランタイム バージョンの使用

これを行うために、.exe アプリケーションは、ランタイムのバージョン情報と supportedRuntime タグを含む app.config ファイルを必要とします。他の Visual Studio 言語には、プロジェクトのプロパティ ページ ダイアログ ボックスで app.config ファイルを変更するための、統合開発環境 (IDE: Integrated Development Environment) サポートが用意されています。たとえば、Visual C# の Windows アプリケーションの **SupportedRuntimes** プロパティを変更し、更新した app.config ファイルを JScript アプリケーションで使用します。

実行時には、app.config ファイルの名前が *filename.ext*.config であることが必要です。*filename.ext* は、アプリケーションを起動した実行可能ファイルの名前です。また、app.config ファイルはその実行可能ファイルと同じディレクトリに格納されている必要があります。たとえば、アプリケーションの名前が TestApp.exe の場合、app.config ファイルの名前は TestApp.exe.config になります。

複数のランタイム バージョンを指定し、複数のランタイム バージョンをインストールしたコンピュータでアプリケーションを実行する場合は、そのシステムで使用できるインストール済みのランタイムと一致するバージョンのうち、config ファイルで最初に指定されているバージョンが使用されます。

詳細については、「方法 : アプリケーション構成ファイルを使用して対象とする .NET Framework のバージョンを指定する」を参照してください。

JScript ASP.NET Web ページは単一ファイルの Web フォーム ページであるため、コンパイラに関連付けられている .NET Framework アセンブリに依存する .dll にはプリコンパイルされません。このため、ページは実行時にコンパイルされ、web.config ファイルにランタイム バージョン情報は不要です。

参照

関連項目

SupportedRuntimes プロパティ

概念

ASP.NET Web ページのコード モデル

JScript プログラマのための追加の技術情報

以下のサイトおよびニュースグループでは、JScript プログラミングの一般的な問題に対する回答が示されています。

Web 上の Microsoft リソース

マイクロソフト プロダクト サポート サービス (http://support.microsoft.com/)

サポート技術情報の文書、ダウンロードと更新、Support Webcast、およびその他のサービスにアクセスできます。

MSDN Newsgroups (http://msdn.microsoft.com/newsgroups/)

コミュニティとして世界中の専門家とやり取りする手段を提供します。

Microsoft ASP.NET (http://www.asp.net/)

JScript での Web 開発に関する文書、デモ、ツールのプレビュー、およびその他の情報を提供します。

Microsoft Windows Script (http://www.microsoft.com/japan/msdn/scripting/default.asp)

JScript 5.6 の開発者に役立つ文書、サンプル、およびその他の情報が含まれています。

ニュースグループの Microsoft リソース

microsoft.public.dotnet.languages.jscript

JScript について質問したり、意見を交わしたりするフォーラムを提供します。

microsoft.public.dotnet.scripting

.NET Framework のスクリプトについて質問したり、意見を交わしたりするフォーラムを提供します。

microsoft.public.vsnet.documentation

JScript のドキュメントについて質問したり、意見を交わしたりするフォーラムを提供します。

microsoft.public.scripting.jscript

JScript 5.*x* について質問したり、意見を交わしたりするフォーラムを提供します。

microsoft.public.scripting.wsh

Microsoft WSH (Windows Script Host) での JScript 5.*x* の使用方法について質問したり、意見を交わしたりするフォーラムを提供します。

Web 上のその他のリソース

4GuysFromRolla.com (http://www.4guysfromrolla.com/)

JScript での Web 開発に関する文書、デモ、ツールのプレビュー、およびその他の情報を提供します。

DevX (http://www.devx.com/)

JScript での開発に関する文書、デモ、ツールのプレビュー、およびその他の情報を提供します。

参照

その他の技術情報

JScript について

JScript コードの作成、コンパイル、およびデバッグ

すべての言語に共通の開発環境である Visual Studio 統合開発環境 (IDE: Integrated Development Environment) には、信頼性のあるコードを作成するためのツールと検証スキームが用意されています。IDE にはデバッグ機能も用意されており、矛盾を修正したり、コードの誤りを解決したりできます。

このセクションの内容

方法 : コマンド ラインで JScript コードをコンパイルする

コマンド ライン コンパイラを使用してコンパイル済みの JScript プログラムを生成する方法について説明します。

Visual Studio での JScript コードの作成

Visual Studio 統合開発環境 (IDE: Integrated Development Environment) を使用して、JScript コードを作成および編集する方法について説明します。

条件付きコンパイル

条件付きコンパイルの使用方法と、条件付きコンパイルを使用する状況について説明します。条件付きコンパイルを使用すると、コンパイル時にさまざまなコード セクションをデバッグ プロセス用に含めることができます。また、下位互換性を犠牲にすることなく、JScript の新しい機能を簡単に使用できます。

ブラウザの機能の検出

スクリプト エンジンの関数および条件付きコンパイルを使用して、Web ブラウザのエンジンがサポートする JScript のバージョンを判断する方法について説明します。

データのコピー、受け渡し、および比較

参照および値によるデータの格納方法の違いと、データの型に応じてどちらが使用されるかについて説明します。

JScript でのメソッドのオーバーロード

同じ名前を持つ一方で異なるシグネチャを持つメソッドをオーバーロードする方法について説明します。

方法 : JScript でイベントを処理する

イベント ハンドラのメソッドを既存のイベントにリンクしてイベントを処理する方法について説明します。

Visual Studio での JScript のデバッグ

コマンド ライン プログラムまたは ASP.NET プログラムのデバッグを有効にする手順を示します。

共通言語ランタイム デバッガによる JScript のデバッグ

コマンド ライン プログラムまたは ASP.NET プログラムで、共通言語ランタイム コンパイラ デバッガを有効にする手順を示します。

JScript スクリプトのトラブルシューティング

構文、スクリプト解釈の順序、自動型変換などの特定の領域での、一般的なスクリプトの誤りを避けるためのヒントを提供します。

関連するセクション

コマンド ラインからのビルド

コマンド ラインからコンパイラを起動する方法について説明します。また、特定の結果を返すコマンド ライン構文の例を示します。

アプリケーションのデバッグとプロファイリング

.NET Framework アプリケーションのデバッグ処理と、Visual Studio がインストールされていない場合に、.NET Framework を使用するアプリケーションを最適化する方法について説明します。

方法：コマンド ラインで JScript コードをコンパイルする

実行可能な JScript プログラムを生成するには、コマンド ライン コンパイラの jsc.exe を使用する必要があります。コンパイラはさまざまな方法で起動できます。

Visual Studio をインストールしている場合は、Visual Studio のコマンド プロンプトを使用して、コンピュータの任意のディレクトリからコンパイラにアクセスできます。The Visual Studio のコマンド プロンプトは、[Microsoft Visual Studio] プログラム グループの [Visual Studio ツール] フォルダにあります。

Windows のコマンド プロンプトでもコンパイラを起動できます。Visual Studio がインストールされていない場合は、これが一般的な方法です。

Windows のコマンド プロンプト

Windows のコマンド プロンプトからコンパイラを起動するには、アプリケーションと同じディレクトリから実行するか、実行可能ファイルへの絶対パスを入力する必要があります。この既定の方法を使用しない場合は、PATH 環境変数を変更して、コンパイラ名を入力するだけで任意のディレクトリからコンパイラを実行できるようにする必要があります。

PATH 環境変数を変更するには

1. Windows の検索機能を使用して、ローカル ドライブで jsc.exe を検索します。jsc.exe が格納されているディレクトリの名前は、Windows ディレクトリの名前と場所、およびインストールされている .NET Framework のバージョンによって決まります。複数のバージョンの .NET Framework がインストールされている場合は、使用するバージョン (通常は最新のバージョン) を決定する必要があります。

   たとえば、コンパイラは `C:\WINNT\Microsoft.NET\Framework\v1.0.2914` などにあります。

2. デスクトップの [マイ コンピュータ] アイコンを右クリックし (Windows 2000 の場合)、ショートカット メニューの [プロパティ] をクリックします。

3. [詳細] タブをクリックし、[環境変数] をクリックします。

4. [システム環境変数] の一覧の [Path] を選択し、[編集] をクリックします。

5. [システム変数の編集] ダイアログ ボックスで、[変数値] ボックスの文字列の末尾にカーソルを移動し、セミコロン (;) と手順 1 で検索されたディレクトリの完全パスを入力します。

   手順 1 の例の場合は、次のように入力します。

   ```
   ;C:\WINNT\Microsoft.NET\Framework\v1.0.2914
   ```

6. [OK] をクリックして編集を完了し、ダイアログ ボックスを閉じます。

PATH 環境変数を変更したら、Windows のコマンド プロンプトを使用して、コンピュータの任意のディレクトリから JScript コンパイラを実行できます。

コンパイラの使用方法

コマンド ライン コンパイラには組み込みのヘルプがあります。ヘルプ画面は、**/help** または **/?** コマンド ライン オプションを使用するか、オプションを指定せずにコンパイラを起動することで表示できます。次に例を示します。

```
jsc /help
```

JScript には、2 とおりの使用方法があります。コマンド ラインからコンパイルするプログラムを作成するか、または ASP.NET で実行するプログラムを作成します。

jsc を使用してコンパイルするには

- コマンド プロンプトで、「jsc file.js」と入力します。

  このコマンドにより、file.js というプログラムがコンパイルされ、file.exe という名前の実行可能ファイルが生成されます。

jsc を使用して .dll ファイルを生成するには

- コマンド プロンプトで、「jsc /target:library file.js」と入力します。

  このコマンドにより、file.js というプログラムが **/target:library** オプションを指定してコンパイルされ、file.dll という名前のライブラリ ファイルが生成されます。

jsc を使用して別の名前の実行可能ファイルを生成するには

- コマンド プロンプトで、「jsc /out:newname.exe file.js」と入力します。

  このコマンドにより、file.js というプログラムが **/out:** オプションを指定してコンパイルされ、newname.exe という名前の実行可能ファイルが生成されます。

jsc を使用してデバッグ情報と共にコンパイルするには

- コマンド プロンプトで、「jsc /debug file.js」と入力します。

  このコマンドにより、file.js というプログラムが **/debug** オプションを指定してコンパイルされ、file.exe という名前の実行可能ファイルと、デバッグ情報を含む file.pdb という名前のファイルが生成されます。

JScript コマンド ライン コンパイラでは、他にも多数のコマンド ライン オプションを使用できます。詳細については、「JScript コンパイラ オプション」を参照してください。

参照

その他の技術情報

JScript コードの作成、コンパイル、およびデバッグ
JScript コンパイラ オプション
条件付きコンパイル

Visual Studio での JScript コードの作成

Visual Studio 統合開発環境 (IDE: Integrated Development Environment) では、JScript の開発用に他の言語と同様のツールが用意されています。[新しいファイル] ダイアログ ボックスには、さまざまな JScript ファイルのフレームワークとなるテンプレートが用意されています。

IDE での手順

Visual Studio を使用して新しい JScript コードを作成するには

1. Microsoft Visual Studio を起動します。
2. [ファイル] メニューの [新しいファイル] をクリックします。
3. [カテゴリ] ダイアログ ボックスで、[スクリプト] フォルダをクリックします。
4. [テンプレート] ダイアログ ボックスで、[JScript ファイル] または [JScript .NET Web フォーム] を選択し、[開く] をクリックします。

Visual Studio を使用して JScript コードを編集するには

1. Microsoft Visual Studio を起動します。
2. [ファイル] メニューの [開く] をポイントし、[ファイル] をクリックします。
3. [ファイルを開く] ダイアログ ボックスで、ソース ファイルを選択し、[開く] をクリックします。

使用する言語に応じて、各ファイルではキーワードが着色されて表示されます。コードを編集するときに、[ダイナミック ヘルプ] ウィンドウに状況依存のヘルプが表示されます。

Visual Studio で JScript コードを保存するには

- [ファイル] メニューの [<ファイル名> の保存] または [名前を付けて <ファイル名> を保存] をクリックします。

メモ
Visual Studio IDE では、JScript のコードをコンパイルできません。コンパイルは、コマンド ラインから、または ASP.NET ページで実行する必要があります。

参照

概念

コマンド ラインからのビルド
Visual Studio での JScript のデバッグ
共通言語ランタイム デバッガによる JScript のデバッグ

その他の技術情報

JScript コードの作成、コンパイル、およびデバッグ
Visual Studio の統合開発環境
IntelliSense の使用方法

条件付きコンパイル

条件付きコンパイルを使用すると、JScript の新しい言語機能を利用しながら、その新機能をサポートしていない以前のバージョンとの互換性も保持できます。JScript の新機能を使用する場合、スクリプトにデバッグ サポートを埋め込む場合、コード実行をトレースする場合などは、条件付きコンパイルを使用するのが一般的です。

このセクションの内容

条件付きコンパイル ディレクティブ

スクリプトのコンパイル方法を制御するディレクティブの一覧を示し、各ディレクティブの正しい構文について説明する情報へのリンクを示します。

条件付きコンパイル ステートメント

条件付きコンパイル変数の値に応じてスクリプトのコンパイルを制御するステートメントについて説明します。

条件付きコンパイル変数

条件付きコンパイルで使用可能な定義済み変数の一覧を示します。

関連するセクション

JScript コードの作成、コンパイル、およびデバッグ

統合開発環境 (IDE: Integrated Development Environment) を使用した JScript コードの作成方法について説明するトピックへのリンクを示します。

方法 : コマンド ラインで JScript コードをコンパイルする

コマンド ライン コンパイラを使用してコンパイル済みの JScript プログラムを生成する方法について説明します。

ブラウザの機能の検出

スクリプト エンジンの関数および条件付きコンパイルを使用して、Web ブラウザのエンジンがサポートする JScript のバージョンを判断する方法について説明します。

JScript コンパイラ オプション

JScript コマンド ライン コンパイラで利用可能なコンパイラ オプションの一覧を示す情報へのリンクを示します。

条件付きコンパイル ディレクティブ

次のディレクティブは、コードをデバッグ用にコンパイルするときの既定の動作を変更します。

ディレクティブ

ディレクティブ	説明
@debug	デバッグ シンボルの出力をオンまたはオフにします。
@position	わかりやすい位置情報をエラー メッセージに表示します。

これらのディレクティブは、ホスト環境 (ASP.NET など) によって自動的に JScript プログラムに含まれるコードをデザインする開発者用に用意されています。含まれるコードは、ホスト環境で実行されるスクリプトの作成者には関係がありません。これらのコード作成者がコードをデバッグする場合は、開発ツールが生成したコードや追加箇所を参照せずに、自分が作成した部分だけを表示する方が便利です。

上記の条件付きコンパイル ディレクティブを使用すると、デバッグ シンボルの出力をオフにして行の位置を再設定することで、コードを "隠ぺい" できます。これにより、スクリプトの作成者は、スクリプトのデバッグ時に自分が作成したコードだけを表示できます。

参照

関連項目

/debug

概念

条件付きコンパイル変数

条件付きコンパイル ステートメント

その他の技術情報

条件付きコンパイル

条件付きコンパイル ステートメント

次のステートメントを利用すると、条件付きコンパイル変数の値に応じてスクリプトのコンパイルを制御できます。JScript に用意された変数を使用するか、**@set** ディレクティブまたは **/define** コマンド ライン オプションを使用して独自の変数を定義できます。

ステートメント

ステートメント	説明
@cc_on	条件付きコンパイルの機能をアクティブにします。
@if	条件式の値を評価し、条件に応じて適切なステートメントを実行します。
@set	条件付きコンパイル ステートメントで使用する変数を作成します。

@cc_on、**@if**、または **@set** ステートメントは、条件付きコンパイルを有効にします。JScript の新機能を使用する場合、スクリプトにデバッグ サポートを埋め込む場合、コード実行をトレースする場合などは、条件付きコンパイルを使用するのが一般的です。

Web ブラウザで実行するスクリプトを作成する場合、条件付きコンパイル コードは常にコメント内に記述します。これにより、条件付きコンパイルをサポートしないホストでは、これらのコードが無視されます。次に例を示します。

```
/*@cc_on @*/
/*@if (@_jscript_version >= 5)
document.write("JScript Version 5.0 or better.<BR>");
@else @*/
document.write("You need a more recent script engine.<BR>");
/*@end @*/
```

この例では、**@cc_on** ステートメントによって条件付きコンパイルがアクティブにされた場合にだけ使用される、特殊なコメント区切り文字を使用しています。条件付きコンパイルをサポートしないスクリプト エンジンでは、新しいスクリプト エンジンが必要であることを示すメッセージだけが表示され、エラーは生成されません。条件付きコンパイルをサポートするエンジンは、エンジンのバージョンに応じて、1 番目または 2 番目の `document.write` をコンパイルします。JScript .NET のバージョン番号は 7*.x* です。詳細については、「ブラウザの機能の検出」を参照してください。

条件付きコンパイルは、サーバー側のスクリプトやコマンド ライン プログラムでも役立ちます。これらのアプリケーションでは、条件付きコンパイルを使用してプログラムに追加の関数をコンパイルし、デバッグ モードでプロファイルを実行できます。

参照

関連項目

/define

概念

条件付きコンパイル変数
条件付きコンパイル ディレクティブ
ブラウザの機能の検出

その他の技術情報

条件付きコンパイル

条件付きコンパイル変数

条件付きコンパイルで使用可能な定義済み変数を次に示します。

変数

変数	説明
@_win32	Win32 システム上で実行されており、/platform オプションが指定されていないか、/platform:anycpu オプションが指定されている場合は **true** です。それ以外の場合は **NaN** です。
@_win16	Win16 システム上で実行されている場合は **true** です。それ以外の場合は **NaN** です。
@_mac	Apple Macintosh システム上で実行されている場合は **true** です。それ以外の場合は **NaN** です。
@_alpha	DEC Alpha プロセッサ上で実行されている場合は **true** です。それ以外の場合は **NaN** です。
@_x86	Intel プロセッサ上で実行されており、/platform オプションが指定されていないか、/platform:anycpu オプションが指定されている場合は **true** です。それ以外の場合は **NaN** です。
@_mc680x0	Motorola 680x0 プロセッサ上で実行されている場合は **true** です。それ以外の場合は **NaN** です。
@_PowerPC	Motorola PowerPC プロセッサ上で実行されている場合は **true** です。それ以外の場合は **NaN** です。
@_jscript	常に **true** です。
@_jscript_build	JScript スクリプト エンジンのビルド番号です。
@_jscript_version	"メジャー番号.マイナ番号" の形式の JScript のバージョン番号です。
@_debug	デバッグ モードでコンパイルされている場合は **true** です。それ以外の場合は **false** です。
@_fast	高速モードでコンパイルされている場合は **true** です。それ以外の場合は **false** です。

メモ

JScript .NET で報告されるバージョン番号は 7.*x* です。JScript 8.0 で報告されるバージョン番号は 8.*x* です。

条件付きコンパイル変数を使用するには、あらかじめ条件付きコンパイルを有効にする必要があります。**@cc_on** ステートメントは条件付きコンパイルを有効にできます。条件付きコンパイル変数は、Web ブラウザ用に作成されたスクリプトでよく使用されます。ASP ページまたは ASP.NET ページ用に作成されたスクリプトや、コマンド ライン プログラム用に作成されたスクリプトでは、コンパイラの機能が他の方法を使用して決定されるため、条件付きコンパイル変数を使用することは一般的ではありません。

Web ページ用のスクリプトを作成する場合、条件付きコンパイル コードは常にコメント内に記述します。これにより、条件付きコンパイルをサポートしないホストでは、これらのコードが無視されます。次に例を示します。

```
/*@cc_on
  document.write("JScript version: " + @_jscript_version + ".<BR>");
  @if (@_win32)
     document.write("Running on 32-bit Windows.<BR>");
  @elif (@_win16)
     document.write("Running on 16-bit Windows.<BR>");
  @else
     document.write("Running on a different platform.<BR>");
  @end
```

```
@*/
```

条件付きコンパイル変数を使用して、スクリプトを解釈するエンジンのバージョン情報を判断できます。これにより、下位互換性を維持する一方で、最新のバージョンの JScript で利用できる機能も使用できます。詳細については、「ブラウザの機能の検出」を参照してください。

参照

概念

条件付きコンパイル ディレクティブ
条件付きコンパイル ステートメント
ブラウザの機能の検出

その他の技術情報

条件付きコンパイル

ブラウザの機能の検出

ブラウザは、ほとんどの JScript の機能に対応していますが、.NET Framework 用の新機能であるクラス ベースのオブジェクト、データ型、列挙型、条件付きコンパイル ディレクティブ、および **const** ステートメントは、サーバー側だけでサポートされています。このため、これらの機能はサーバー側のスクリプトだけで使用します。詳細については、「JScript のバージョン情報」を参照してください。

JScript スクリプトでは、スクリプトを解釈またはコンパイルするエンジンの機能を検出できます。サーバー側の (ASP または ASP.NET で実行される) アプリケーションやコマンド ライン プログラムを作成している場合、この処理は不要です。これらの場合、サポートされる JScript のバージョンやコードは簡単にわかります。ただし、クライアント側のスクリプトをブラウザで実行する場合は、スクリプトとブラウザの JScript エンジンに互換性があることを確実にするために、この検出が重要になります。

JScript の互換性を確認する方法には、スクリプト エンジンの関数を使用する方法と、条件付きコンパイルを使用する方法の 2 つがあります。どちらの方法にもそれぞれ利点があります。

スクリプト エンジンの関数

スクリプト エンジンの関数
(**ScriptEngine**、**ScriptEngineBuildVersion**、**ScriptEngineMajorVersion**、**ScriptEngineMinorVersion**) は、スクリプト エンジンの現在のバージョンに関する情報を返します。詳細については、「関数 (JScript)」を参照してください。

互換性を維持するために、サポートされる JScript のバージョンを確認するページでは、JScript Version 1 の機能だけを使用してください。エンジンが JScript の Version 1.0 より後のバージョンをサポートしている場合は、高度な機能を含む別のページにリダイレクトできます。つまり、サポートする JScript のバージョンごとに、異なる Web ページが必要となります。ほとんどの場合、最も効果的な解決策は 2 つのページを用意することです。1 つのページは JScript の特定のバージョン用にデザインし、もう 1 つのページは JScript を使用せずに動作するようにデザインします。

メモ

高度な機能を利用する JScript コードは、互換性のないエンジンを持つブラウザで実行されない、別のページに配置する必要があります。ブラウザのスクリプト エンジンは、ページに含まれているすべての JScript コードを解釈するため、この処理が必要です。**if...else** ステートメントを使用して、最新の JScript を使用するコードと JScript Version 1 のコードを切り替える方法は、古いエンジンでは動作しません。

次の例は、スクリプト エンジンの関数の使用例を示しています。これらの関数は JScript Version 2.0 で導入されたため、関数を使用する前に、エンジンが関数をサポートしているかどうかを確認する必要があります。エンジンが JScript Version 1.0 しかサポートしていない場合、または JScript を認識しない場合、**typeof** 演算子は各関数名に対して "undefined" を返します。

```
if("undefined" == typeof ScriptEngine) {
   // This code is run if the script engine does not support
   // the script engine functions.
   var version = 1;
} else {
   var version = ScriptEngineMajorVersion();
}
// Display the version of the script engine.
alert("Engine supports JScript version " + version);
// Use the version information to choose a page.
if(version >= 5) {
   // Send engines compatible with JScript 5.0 and better to one page.
   var newPage = "webpageV5.htm";
} else {
   // Send engines that do not interpret JScript 5.0 to another page.
   var newPage = "webpagePre5.htm";
}
location.replace(newPage);
```

条件付きコンパイル

条件付きコンパイル変数および条件付きコンパイル ステートメントは、条件付きコンパイルをサポートしていないエンジンから JScript コードを隠ぺいできます。Web ページに代替コードを少しだけ含める場合は、この方法が役に立ちます。

メモ

条件付きコンパイル ブロックでは、複数行のコメントを使用しないでください。条件付きコンパイルをサポートしていないエンジンが、解釈を誤る可能性があります。

```
<script>
/*@cc_on
@if(@_jscript_version >= 5 )
// Can use JScript Version 5 features such as the for...in statement.
// Initialize an object with an object literal.
var obj = {"a" : "Athens" , "b" : "Belgrade", "c" : "Cairo"};
var key;
// Iterate the properties.
for (key in obj) {
   document.write("The "+key+" property has value "+obj[key]+".<BR>");
}
@else
@*/
alert("Engine cannot interpret JScript Version 5 code.");
//@end
</script>
```

条件付きの **@if** ブロックに多くのコードが含まれる場合は、スクリプト エンジンの関数を使用する方法が簡単です。

参照

概念

JScript のバージョン情報

その他の技術情報

JScript コードの作成、コンパイル、およびデバッグ

関数 (JScript)

条件付きコンパイル

データのコピー、受け渡し、および比較

JScript がデータをコピー、受け渡し、および比較する方法は、データの格納方法によって決まります。また、データの格納方法は、データの型によって決まります。JScript は、データを値または参照によって格納します。

"値" と "参照" によるデータのコピー、受け渡し、および比較

JScript では、数値およびブール値 (**true** と **false**) のコピー、受け渡し、および比較を、"値によって" 実行します。この処理では、コンピュータ メモリに領域が割り当てられ、元の値がこの領域にコピーされます。元の値とコピーされた値は別の要素と見なされるため、元の値を変更しても、コピーされた値には影響ありません (逆の場合も同様です)。2 つの数値またはブール値が同じ値である場合、2 つの値は等しいと見なされます。

オブジェクト、配列、および関数の場合、コピー、受け渡し、および比較は "参照によって" 行われます。この処理では、元の項目への参照が作成され、作成された参照がコピーのように使用されます。元の項目に変更を加えると、元の項目とコピーされた項目の両方が変更されます (逆の場合も同様です)。実際に存在する項目は 1 つだけであり、コピーされた項目はデータへの別の参照です。

参照による比較を正しく行うには、2 つの変数が同じ項目を参照している必要があります。たとえば、2 つの異なる **Array** オブジェクトは、まったく同じ要素を持っていても、比較においては常に等しくないものとして識別されます。比較処理を正しく行うには、一方の変数がもう一方の変数の参照である必要があります。2 つの Array オブジェクトが同じ要素を持つかどうかを確認するには、**toString()** メソッドの結果を比較します。

文字列は、参照によってコピーおよび受け渡しされます。文字列の比較方法は、文字列がオブジェクトかどうかによって決まります。**new String** ("something") で作成された **String** オブジェクトどうしは、参照によって比較されます。文字列のいずれか (または両方) がリテラルかプリミティブ文字列値である場合は、値によって比較されます。詳細については、「JScript の代入と等価比較」を参照してください。

メモ

ASCII 文字セットと ANSI 文字セットの構成方法により、文字の順序の中で大文字は小文字よりも先に来ます。たとえば、"Zoo" は "aardvark" より小さいと評価されます。大文字と小文字を区別しない一致検証を行うには、双方の文字列で **toUpperCase()** または **toLowerCase()** を呼び出します。

関数パラメータ

関数に値渡しでパラメータを渡す場合は、そのパラメータの新しいコピーが作成されます。作成されたコピーは、その関数の中だけで存在します。オブジェクトや配列は参照によって渡されますが、関数内の新しい値で直接上書きする場合、関数の外ではこの新しい値は反映されません。関数の外でも有効なのは、オブジェクトのプロパティや配列の要素に対する変更だけです。

たとえば、次のプログラムには 2 つの関数があります。最初の関数は入力パラメータを上書きしています。これにより、パラメータが変更されても入力された元の引数が影響を受けないようにします。2 番目の関数は、オブジェクトを上書きせずに、オブジェクトのプロパティを変更しています。

```
function clobber(param) {
   // Overwrite the parameter; this will not be seen in the calling code
   param = new Object();
   param.message = "This will not work.";
}

function update(param) {
    // Modify the property of the object; this will be seen in the calling code.
    param.message = "I was changed.";
}

// Create an object, and give it a property.
var obj = new Object();
obj.message = "This is the original.";

// Call clobber, and print obj.message.
clobber(obj);
print(obj.message);

// Call update, and print obj.message.
update(obj);
print(obj.message);
```

このコードの出力は次のようになります。

```
This is the original.
I was changed.
```

データの比較

JScript では、データを値または参照によって比較できます。値による比較を実行する場合は、2 つの項目を比較して、お互いに等しいかどうかを調べます。通常、この比較はバイトごとに実行されます。参照による比較では、2 つの項目が同じ項目を参照しているかどうかを確認します。参照先の項目が同じである場合、両者は等しいと評価されます。参照先の項目が異なる場合は、値渡しで比較した場合に等しくなる値が格納されていても、両者は等しくないと評価されます。

文字列は、オブジェクトかどうかに応じて、値または参照によって比較されます。両方の文字列が **String** オブジェクトである場合、文字列は参照によって比較されます。それ以外の場合は、値によって比較されます。したがって、2 つの文字列が個別に作成されていても、それぞれの内容が同じであると、2 つの文字列は等しいと評価されます。2 つの **String** オブジェクトの値を比較するには、まず **toString** または **valueOf** メソッドを使用してオブジェクトを非オブジェクトの文字列に変換し、変換された文字列を比較します。詳細については、「JScript の代入と等価比較」を参照してください。

参照

概念

JScript の代入と等価比較
データ型の概要

その他の技術情報

JScript 言語の紹介
JScript の関数

JScript でのメソッドのオーバーロード

クラスの 2 つ以上の JScript メンバ (関数またはプロパティ) が同じ名前を持つ一方でそのシグネチャが異なる場合、そのメンバは "オーバーロードされた" 関数 (プロパティ) と呼ばれます。関数の "シグネチャ" は、受け取るパラメータの数、型、および順序に基づきます。2 つの関数が同じ型の引数を同じ数だけ同じ順序で受け取る場合、それらは同じシグネチャを持つことになります。関数が同じ型の引数を異なる順序で受け取る場合や、異なる数または型の引数を受け取る場合、関数のシグネチャは異なります (引数の名前はシグネチャとは関係ありません)。静的関数はオーバーロードの対象となりますが、戻り値の型と同様に、メソッドの静的ステータスはシグネチャに影響を与えません。したがって、インスタンス メソッドと同じ名前を持つ静的メソッドは、異なるパラメータ リストを持つ必要があります。

処理ロジック

オーバーロードされた関数が呼び出されると、関数に渡される引数の実際の型に応じて、渡された引数とその引数が最も一致するオーバーロードされた関数が呼び出されます。引数の型が特定のオーバーロードに正確に一致する場合は、そのオーバーロードが呼び出されます。引数の型がいずれのオーバーロードにも正確に一致しない場合、呼び出されるオーバーロードは消去法によって決定されます。この消去法においては、利用できるオーバーロードの型に実際の型をどれだけ簡単に変換できるかが基準となります。詳細については、「JScript における型の強制変換」を参照してください。この例の MethodOverload クラスには、Greetings という名前のオーバーロードされたメソッドが 3 つあります。1 番目のオーバーロードはパラメータを受け取りません。2 番目のオーバーロードは **String** 型のパラメータを 1 つ受け取ります。3 番目のオーバーロードは、**String** 型と **int** 型の 2 つのパラメータを受け取ります。

```
var methodOverload = new MethodOverload();
methodOverload.Greetings();
methodOverload.Greetings("Mr. Brown");
methodOverload.Greetings(97, "Mr. Brown");

class MethodOverload
{
   function Greetings()
   {
     print("Hello, and welcome!");
   }
   function Greetings(name : String)
   {
     print("Hello, " + name + "!");
   }
   function Greetings(ticket : int, name : String)
   {
     print("Hello, " + name + "! Your ticket number is " + ticket + ".");
   }
}
```

このプログラムの出力は次のようになります。

```
Hello, and welcome!
Hello, Mr.Brown!
Hello, Mr.Brown! Your ticket number is 97.
```

参照

その他の技術情報

JScript 言語の紹介
JScript のデータ型

方法：JScript でイベントを処理する

イベントとは、マウス ボタンをクリックする、キーを押す、データを変更する、文書やフォームを開くなど、一般にユーザーが実行するアクションです。さらに、プログラム コードもアクションを実行できます。イベント ハンドラは、イベントに関連付けられたメソッドです。イベントが発生すると、イベント ハンドラのコードが実行されます。JScript .NET イベント ハンドラは、あらゆる種類の .NET アプリケーション (ASP.NET、Windows フォーム、コンソールなど) のイベントに関連付けることができます。ただし、新しいイベントを JScript で宣言することはできません。JScript コードでは、既存のイベントだけを使用できます。

ボタン コントロールの Click イベントのイベント ハンドラを作成するには

- 次のコードを追加します。

```
// Events.js
import System;
import System.Windows.Forms;

class EventTestForm extends Form
{
  var btn : Button;

  function EventTestForm()
  {
    btn = new Button;
    btn.Text = "Fire Event";
    Controls.Add(btn);
    // Connect the function to the event.
    btn.add_Click(ButtonEventHandler1);
    btn.add_Click(ButtonEventHandler2);
  }

  // Add an event handler to respond to the Click event raised
  // by the Button control.
  function ButtonEventHandler1(sender, e : EventArgs)
  {
    MessageBox.Show("Event is Fired!");
  }

  function ButtonEventHandler2(sender, e : EventArgs)
  {
    MessageBox.Show("Another Event is Fired!");
  }
}

Application.Run(new EventTestForm);
```

メモ

それぞれのイベント ハンドラには、2 つのパラメータがあります。最初のパラメータ sender では、イベントを発生させたオブジェクトへの参照を示します。2 番目のパラメータ (上の例の e) では、処理されるイベントに応じた特定のオブジェクトを渡します。このオブジェクトのプロパティ (場合によってはメソッド) を参照すると、マウス イベントのマウスの位置や、ドラッグ アンド ドロップ イベントで転送されるデータなどの情報を取得できます。

コードをコンパイルするには

1. Visual Studio に付属のコマンド ライン コンパイラ jsc.exe を使用します。
2. 次のコマンド ライン ディレクティブを入力して、Events.exe という名前の Windows 実行可能 (EXE) プログラムを作成します。

jsc /target:winexe Events.js

メモ

必要に応じてあるイベントに複数の関数が関連付けられている場合、1 つのイベントを発生させたときに複数のイベント ハンドラが呼び出されます。

```
btn.add_Click(ButtonEventHandler1);
btn.add_Click(ButtonEventHandler2);
. . .
```

参照

処理手順

方法 : コマンド ラインで JScript コードをコンパイルする

その他の技術情報

JScript コードの作成、コンパイル、およびデバッグ

Visual Studio での JScript のデバッグ

JScript プログラムには、コマンド ラインで実行されるようにデザインされている場合や ASP.NET ページで実行するようにデザインされている場合があります。プログラムの種類はデバッグの方法に影響します。

プロシージャ

コマンド ライン プログラムに対してデバッグを設定するには

1. **/debug** フラグを使用してデバッグするプログラムをコンパイルします。詳細については、「/debug」を参照してください。
2. Microsoft Visual Studio を起動します。
3. [ファイル] メニューの [開く] をポイントし、[プロジェクト] をクリックします。
4. [プロジェクトを開く] ダイアログ ボックスで、コンパイルしたプログラム (拡張子が .exe のファイル) を選択し、[開く] をクリックします。
5. [ファイル] メニューの [開く] をポイントし、[ファイル] をクリックします。
6. [ファイルを開く] ダイアログ ボックスで、ソース コード (拡張子が .js のファイル) を選択し、[開く] をクリックします。
7. [ファイル] メニューの [すべてを保存] をクリックします。
8. 名前と場所を選択して、新しいプロジェクトを保存します。

この設定が完了したら、「Visual Studio を使用してデバッグを行うには」の手順を実行できます。

ASP.NET プログラムに対してデバッグを設定するには

1. Microsoft Visual Studio を起動します。
2. デバッグする ASP.NET ファイルを開きます。
3. **@page** ディレクティブでデバッグ フラグを true に設定します。次に例を示します。

```
<%@page Language=jscript debug=true %>
```

4. ブラウザでページを開いて、ページをコンパイルします。
5. Visual Studio で、[ツール] メニューの [デバッグ プロセス] をクリックします。
6. [プロセス] ダイアログ ボックスで、[システム プロセスを表示] および [すべてのセッションのプロセスを表示] をクリックします。
7. [プロセス] ダイアログ ボックスの選択可能なプロセス ペインで、Web アプリケーションを実行する ASP.NET ワーカー プロセスを選択し、[アタッチ] をクリックします。

   既定では、ワーカー プロセスは、IIS 5.*x* (Windows 2000 および Windows XP) の場合は aspnet_wp.exe、IIS 6.0 (Windows Server 2003) の場合は w3wp.exe です。
8. [プロセスにアタッチ] ダイアログ ボックスの [Common Language Runtime] を選択し、[OK] をクリックします。
9. [プロセス] ダイアログ ボックスで、[閉じる] をクリックします。

この設定が完了したら、「Visual Studio を使用してデバッグを行うには」の手順を実行できます。

Visual Studio を使用してデバッグを行うには

1. Visual Studio IDE で、上で説明した方法を使用して、デバッグするファイルを開きます。
2. ファイルのブレークポイントを設定する場所にカーソルを移動し、F9 キーを押します。
3. 前の手順を繰り返して、ブレークポイントを追加します。
4. [デバッグ] メニューの [開始] をクリックします。

   プログラムは、ブレークポイントに到達するまで、またはランタイム エラーが発生するまで実行されます。
5. いくつかのウィンドウが開き、その他のデバッグ タスクを実行できます。詳細については、「デバッガのロードマップ」を参照してください。
6. デバッグを終了してもプログラムの実行を続けるには、[デバッグ] メニューの [すべてデタッチ] をクリックします。

この設定を行わない場合は、デバッグを中止するとプログラムも終了します。

解説

コマンド ラインからコンパイルされたプログラムをデバッグする場合、Visual Studio は、デバッグを開始するたびにコンパイル済みプログラムを再読み込みします。したがって、JScript コードを修正してコードを再コンパイルすると、変更の効果を確認できます。

参照

処理手順

Visual Studio での JScript コードの作成

概念

コマンド ラインからのビルド

コマンド ラインからのビルド

共通言語ランタイム デバッガによる JScript のデバッグ

その他の技術情報

デバッガのロードマップ

Web アプリケーションのデバッグ

共通言語ランタイム デバッガによる JScript のデバッグ

JScript プログラムには、コマンド ラインで実行されるようにデザインされている場合や ASP.NET ページで実行するようにデザインされている場合があります。プログラムの種類はデバッグの方法に影響します。

共通言語ランタイム デバッガである dbgclr.exe は、.NET Framework がインストールされているディレクトリの GuiDebug ディレクトリにあります。

dbgclr.exe を使用するには、プログラム名をパス名で修飾するか、または検索パスにパスを追加する必要があります。

プロシージャ

コマンド ライン プログラムに対してデバッグを設定するには

1. 任意のエディタを使ってプログラムを作成し、テキスト形式で保存します。
2. /debug フラグを使用してプログラムをコンパイルします。詳細については、「/debug」を参照してください。
3. 共通言語ランタイム デバッガ dbgclr を起動します。
4. dgbclr で、[ファイル] メニューの [開く] をポイントし、[ファイル] をクリックします。
5. [ファイルを開く] ダイアログ ボックスで、デバッグするソース ファイル (拡張子が .js のファイル) を開きます。
6. [デバッグ] メニューの [デバッグするプログラム] をクリックします。
7. [デバッグするプログラム] ダイアログ ボックスで、プログラム ペインの横にある省略記号ボタンをクリックします。
8. [デバッグするプログラムの検索] ウィンドウで、コンパイルしたプログラム (拡張子が .exe のファイル) を選択し、[開く] をクリックします。
9. [デバッグするプログラム] ダイアログ ボックスで、[OK] をクリックします。

この設定が完了したら、「共通言語ランタイム デバッガを使用してデバッグするには」の手順を実行できます。

ASP.NET プログラムに対してデバッグを設定するには

1. 任意のエディタを使ってプログラムを作成し、テキスト形式で保存します。
2. ASP.NET の HTML ラッパーを記述します。コードに次の行を挿入して、JScript コードをデバッグすることを指定します。

   ```
   <%@page Language=jscript debug=true %>
   ```

3. ブラウザでページを開いて、ページをコンパイルします。
4. 共通言語ランタイム デバッガ dbgclr を起動します。
5. dgbclr で、[ツール] メニューの [デバッグ プロセス] をクリックします。
6. [プロセス] ウィンドウで、[システム プロセスを表示] および [すべてのセッションのプロセスを表示] をクリックします。
7. [選択可能なプロセス] ダイアログ ボックスで、Web アプリケーションを実行する ASP.NET ワーカー プロセスを選択し、[アタッチ] をクリックします。次に、[閉じる] をクリックします。

   既定では、ワーカー プロセスは、IIS 5.*x* (Windows 2000 および Windows XP) の場合は aspnet_wp.exe、IIS 6.0 (Windows Server 2003) の場合は w3wp.exe です。
8. [ファイル] メニューの [開く] をポイントし、[ファイル] をクリックします。
9. [ファイルを開く] ウィンドウでソース コードを選択し、[開く] をクリックします。

この設定が完了したら、「共通言語ランタイム デバッガを使用してデバッグするには」の手順を実行できます。

共通言語ランタイム デバッガを使用してデバッグするには

1. ファイルのブレークポイントを設定する場所にカーソルを移動し、F9 キーを押します。
2. 前の手順を繰り返して、ブレークポイントを追加します。
3. [デバッグ] メニューの [開始] をクリックします。

   プログラムは、ブレークポイントに到達するまで、またはランタイム エラーが発生するまで実行されます。いくつかのウィンドウが開き、その他のデバッグ タスクを実行できます。

4. デバッグを終了してもプログラムの実行を続けるには、[デバッグ] メニューの [すべてデタッチ] をクリックします。

   この設定を行わない場合は、デバッグを中止するとプログラムも終了します。

解説

コマンド ラインからコンパイルされたプログラムをデバッグする場合、dgbclr は、デバッグを開始するたびにコンパイル済みプログラムを再読み込みします。したがって、JScript コードを修正してコードを再コンパイルすると、変更の効果を確認できます。

参照

処理手順

Visual Studio での JScript コードの作成

概念

コマンド ラインからのビルド

Visual Studio での JScript のデバッグ

CLR デバッガ (DbgCLR.exe)

その他の技術情報

デバッガのロードマップ

JScript スクリプトのトラブルシューティング

すべてのプログラミング言語には、初心者だけでなく経験を積んだユーザーでも陥りやすい、思いがけない危険があります。ここでは、JScript スクリプトを記述するときに発生する可能性のある問題をいくつか紹介します。

構文エラー

プログラミング言語の構文は自然言語の文法に比べてかなり厳密であるため、スクリプトを記述するときには細心の注意を払う必要があります。たとえば、文字列のパラメータを指定するときに、引用符で囲むのを忘れると問題が発生します。

スクリプトの解釈の順序

Web ページでは、JScript の解釈はブラウザの HTML 解析プロセスに依存します。<HEAD> タグ内のスクリプトは、<BODY> タグの内部にあるテキストよりも先に解釈されます。したがって、<BODY> タグで作成されるオブジェクトは、ブラウザが <HEAD> 要素を解析する時点では存在せず、スクリプト内では操作できません。

メモ
この動作は Internet Explorer 固有の動作です。ASP および WSH には、他のホストで適用される異なる実行モデルがあります。

型の強制変換

JScript は、自動強制変換が行われる、型指定の緩い言語です。このため、異なる型を持つ値どうしは厳密には等しくありませんが、次に示す式の例は **true** として評価されます。

```
"100" == 100;
false == 0;
```

型と値の両方が同じであることを確認するには、厳密等価演算子 === を使用します。次の例は、どちらも false に評価されます。

```
"100" === 100;
false === 0;
```

演算子の優先順位

式が評価されるときの演算の実行順序は、演算子が記述されている順序よりも、演算子の優先順序に依存します。次の式では、減算演算子の方が乗算演算子よりも前にありますが、乗算が先に実行されます。

```
theRadius = aPerimeterPoint - theCenterpoint * theCorrectionFactor;
```

詳細については、「演算子の優先順位」を参照してください。

オブジェクトでの for...in ループの使用

オブジェクトの各プロパティに対して **for...in** ループを使用する場合、オブジェクトの各フィールドがどのような順序でループ カウンタ変数に代入されるかは、予測することも制御することもできません。さらに、実装する言語が変われば、この順序も変わる可能性があります。詳細については、「for...in ステートメント」を参照してください。

with キーワード

with キーワードは、指定したオブジェクトに既に存在するプロパティを扱う場合には便利ですが、このキーワードを使用してオブジェクトにプロパティを追加することはできません。オブジェクト内に新しいプロパティを作成するには、オブジェクトを明示的に指定する必要があります。詳細については、「with ステートメント」を参照してください。

this キーワード

this キーワードはオブジェクトの定義内に存在しますが、**this** キーワードやその他の類似したキーワードを使用しても、関数がオブジェクトの定義中にない場合は、現在実行中の関数を参照できません。ただし、その関数がメソッドとしてオブジェクトに割り当てられる場合は、関数内で **this** キーワードを使用してそのオブジェクトを参照できます。詳細については、「this ステートメント」を参照してください。

Internet Explorer または ASP.NET でスクリプトを出力するスクリプトの作成

</SCRIPT> タグがインタープリタで解釈されると、現在のスクリプトは終了します。"</SCRIPT>" 自体を表示するには、2 つ以上の文字列として記述してください。たとえば、"</SCR" と "IPT>" のように記述すると、それらを表示するステートメントの中で 2 つを連結できます。

Internet Explorer での暗黙のウィンドウ参照

複数のウィンドウを同時に開くことができるため、暗黙的なウィンドウ参照はすべて、現在のウィンドウを指します。その他のウィンドウを指すには、明示的な参照を使用する必要があります。

参照

処理手順

Visual Studio での JScript コードの作成

概念

Visual Studio での JScript のデバッグ

その他の技術情報

JScript コードの作成、コンパイル、およびデバッグ

JScript での情報の表示

通常、プログラムはユーザーに対して情報を表示します。最も基本的な方法は、テキスト文字列を表示する方法です。JScript プログラムは、プログラムの使用方法に応じて、さまざまな方法で情報を表示できます。JScript は、通常、ASP.NET ページ、クライアント側の Web ページ、およびコマンド ラインの 3 とおりの方法で使用されます。それぞれの環境で表示を行う方法を説明します。

このセクションの内容

コマンド ライン プログラムからの表示

JScript の **print** ステートメントまたは .NET Framework の Show メソッドを使用して、コマンド ライン プログラムからデータを表示する方法について説明します。

ASP.NET からの表示

<%= %> を使用して、ASP.NET プログラムからデータを表示する方法について説明します。

ブラウザへの情報表示

write メソッドまたは **writeln** メソッドを使用して、データを直接ブラウザに表示する方法について説明します。

メッセージ ボックスの使用

ウィンドウ オブジェクトの **alert**、**confirm**、および **prompt** メソッドを使用して、ユーザーからの入力を要求する、入力メッセージ ボックスを作成する方法について説明します。

関連するセクション

JScript リファレンス

JScript 言語リファレンスを構成する要素の一覧を示し、各要素の適切な構文例を示します。

System.Windows.Forms.MessageBox.Show

MessageBox.Show メソッドの構文と、さまざまな結果を返すオプションについて説明します。

ASP.NET からの表示

ASP.NET プログラムから情報を表示する方法はいくつかあります。1 つは <%= %> を使用する方法です。また、**Response.Write** ステートメントを使用する方法もあります。

<%= %> の使用方法

ASP.NET プログラムから情報を最も簡単に表示するには、<%= %> を使用します。等号の後に入力した値が、現在のページに表示されます。次のコードは、変数 `name` の値を表示します。

```
Hello <%= name %>!
```

name の値が "Frank" である場合、現在のページに次の文字列が表示されます。

```
Hello Frank!
```

単一の情報を表示する場合は、**<%= %>** が最も便利です。

Response.Write ステートメント

テキストを表示する別の方法として、**Response.Write** ステートメントを使用する方法があります。ステートメントは <% %> ブロックで囲むことができます。

```
<% Response.Write("Hello, World!") %>
```

Response.Write ステートメントは、スクリプト ブロック内の関数またはメソッドでも使用できます。次の例は、**Response.Write** ステートメントを含む関数を示しています。

メモ

ASP.NET ページでは、関数や変数を <script> ブロック内に定義する必要があります。実行可能コードは <% %> ブロックで囲みます。

```
<script runat="server" language="JScript">
   function output(str) {
      Response.Write(str);
   }
   var today = new Date();
</script>
Today's date is <% output(today); %>. <BR>
```

Response.Write ステートメントの出力は、処理されるページに組み込まれます。これにより、**Response.Write** の出力を使って、テキストを表示するコードを記述できます。たとえば、次のコードは、ページにアクセスしているブラウザの警告ウィンドウに、サーバーの現在の日付を表示するスクリプト ブロックを作成します。<script> タグは、サーバーがタグを処理しないように分割されています。

```
<script runat="server" language="JScript">
   function popup(str) {
      Response.Write("<scr"+"ipt> alert('"+str+"') </scr"+"ipt>");
   }
   var today = new Date();
</script>
<% popup(today); %>
```

詳細については、「Response」を参照してください。

参照

概念

ASP.NET の概要

その他の技術情報

JScript での情報の表示

コマンド ライン プログラムからの表示

JScript がコマンド ライン プログラムからデータを表示するには、3 つの方法があります。Microsoft JScript コマンド ライン コンパイラは、**print** ステートメントを提供します。Console クラスでは、コンソールを使ってユーザーとのやり取りを簡単にするメソッドが用意されています。

Show メソッドは、情報を表示して、ポップアップ ボックスからの入力を受け取ります。

print ステートメント

情報を表示する最も一般的な方法は **print** ステートメントです。このステートメントは、文字列型の引数を 1 つ受け取り、受け取った文字列の最後に改行文字を付加して、コマンド ライン ウィンドウに表示します。

単一引用符または二重引用符で文字列を囲んで、文字列に引用符やアポストロフィを含めることができます。

```
        print("Pi is approximately equal to " + Math.PI);
print();
```

メモ

print ステートメントは、JScript コマンド ライン コンパイラでコンパイルされるプログラムでしか利用できません。ASP.NET ページで **print** を使用すると、コンパイル エラーが発生します。

Console クラス

Console クラスは、コンソールによるユーザーとのやり取りを簡単にするメソッドとプロパティを公開します。**Console** クラスの **WriteLine** メソッドは、**print** ステートメントに似た機能を提供します。**Write** メソッドは、改行文字を付加せずに文字列を表示します。**Console** クラスの **ReadLine** メソッドも役に立ちます。このメソッドは、コンソールから入力されたテキストを読み取ります。

.NET Framework のクラスおよびメソッドを使用するには、まず **import** ステートメントを使用して、クラスが属している名前空間をインポートします。メソッドを呼び出すには、完全限定名を使用します。現在のスコープに同じ名前のメソッドがない場合は、名前だけを使用することもできます。

```
import System;
System.Console.WriteLine("What is your name: ");
var name : String = Console.Readline();
Console.Write("Hello ");
Console.Write(name);
Console.Write("!");
```

このプログラムは、コンソールから名前が入力されることを要求しています。名前として「Pete」と入力すると、次のように表示されます。

```
What is your name:
Pete
Hello Pete!
```

詳細については、「**Console**」を参照してください。

Show メソッド

Show メソッドは、オーバーロードされるため、さまざまな用途に使用されます。最も簡単なオーバーロードでは、表示するテキスト文字列を表す引数を 1 つ受け取ります。メッセージ ボックスはモーダルです。

メモ

明示的に閉じるまで表示されたままになるウィンドウやフォームはモーダルです。ダイアログ ボックスやメッセージは、通常はモーダルです。たとえば、モーダル ダイアログ ボックスでは、ダイアログ ボックスの [OK] をクリックするまで、他のウィンドウにはアクセスできません。

```
import System.Windows.Forms;
System.Windows.Forms.MessageBox.Show("Welcome! Press OK to continue.");
MessageBox.Show("Great! Now press OK again.");
```

Show メソッドの別のオーバーロードを使用して、キャプション、その他のボタン、アイコン、または既定のボタンも設定できます。詳細については、「**Show**」を参照してください。

参照

関連項目

print ステートメント

import ステートメント

その他の技術情報

JScript での情報の表示

ブラウザへの情報表示

ブラウザは、ほとんどの JScript の機能に対応していますが、.NET Framework 用の新機能であるクラス ベースのオブジェクト、データ型、列挙型、条件付きコンパイル ディレクティブ、および **const** ステートメントは、サーバー側だけでサポートされています。このため、これらの機能はサーバー側のスクリプトだけで使用します。詳細については、「JScript のバージョン情報」を参照してください。

スクリプトをブラウザ (クライアント側) で実行する場合、経験の豊富な開発者は、スクリプト エンジンのバージョンを検知するコードを挿入します。スクリプトでエンジンのバージョンを検知した後、ブラウザのスクリプト エンジンと互換性のあるスクリプトを使用したページにブラウザをリダイレクトできます。詳細については、「ブラウザの機能の検出」を参照してください。

JScript は、ブラウザのドキュメント オブジェクトの **write** および **writeln** メソッドを使用して、ブラウザに情報を表示します。また、ブラウザ内のフォームや、警告、入力、および確認の各メッセージ ボックスにも情報を表示できます。詳細については、「メッセージ ボックスの使用」を参照してください。

document.write と document.writeln を使用する

情報を表示する最も一般的な方法は、document オブジェクトの write メソッドを使用することです。このメソッドには、ブラウザに表示する文字列を受け取るための引数が 1 つあります。引数に指定する文字列は、書式なしのテキストでも HTML でもかまいません。

文字列を囲む引用符には、単一引用符 (') も二重引用符 (") も使用できるので、文字列の中で引用符やアポストロフィを使用できます。

```
document.write("Pi is approximately equal to " + Math.PI);
document.write();
```

メモ

次の簡単な関数を使用すると、ブラウザ ウィンドウにテキストを表示しようとするたびに「`document.write`」と入力する手間を省くことができます。この関数では、記述しようとした内容が未定義であってもエラーは発生せず、代わりにコマンド `w();` を実行して空白行を表示します。

```
function w(m) { // Write function.
   m = String(m); //  Make sure that the m variable is a string.
   if ("undefined" != m) { // Test for empty write or other undefined item.
      document.write(m);
   }
   document.write("<br>");
}

w('<IMG SRC="horse.gif">');
w();
w("This is an engraving of a horse.");
w();
```

writeln メソッドは、**write** メソッドとほとんど同じですが、文字列の最後に必ず改行文字を追加します。HTML では通常、この改行は単なる空白文字に変換されますが、<PRE> タグや <XMP> タグの内部で使用した場合は、そのまま改行文字としてブラウザに表示されます。

write メソッドが呼び出されたときに、ドキュメントが開いて解析するプロセスにない場合、**write** メソッドはそのドキュメントを開いてクリアします。これにより、予測できない結果となる場合があります。次の例は、1 分おきに時刻を表示するように記述したスクリプトです。プロセス中に自分自身をクリアしてしまうため、2 回目以降の表示処理は失敗します。

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JScript">
function singOut()  {
   var theMoment = new Date();
   var theHour = theMoment.getHours();
   var theMinute = theMoment.getMinutes();
   var theDisplacement = (theMoment.getTimezoneOffset() / 60);
   theHour -= theDisplacement;
   if (theHour > 23)  {
      theHour -= 24
   }
   // The following line clears the script the second time it is run.
```

```
    document.write(theHour + " hours, " + theMinute + " minutes, Coordinated Universal Time.
");
    window.setTimeout("singOut();", 60000);
}
</SCRIPT>
</HEAD>
<BODY>
<SCRIPT>
singOut();
</SCRIPT>
</BODY>
</HTML>
```

document.write の代わりに window オブジェクトの **alert** メソッドを次のように使うと、このスクリプトは正常に動作します。

```
    // This line produced the intended result.
    window.alert(theHour + " hours, " + theMinute + " minutes, Coordinated Universal Time.")
;
```

Internet Explorer Version 5 以降では、**element.innerText** または **element.innerHTML** の使用が望ましい方法です。

現在のドキュメントをクリアする

document オブジェクトの **clear** メソッドは、現在のドキュメントの内容をクリアします。このメソッドは、ドキュメントのほかの部分と一緒にスクリプトもクリアするため、使用には十分注意する必要があります。

```
document.clear();
```

参照

概念

メッセージ ボックスの使用

ブラウザの機能の検出

その他の技術情報

JScript での情報の表示

メッセージ ボックスの使用

ブラウザは、ほとんどの JScript の機能に対応していますが、.NET Framework 用の新機能であるクラス ベースのオブジェクト、データ型、列挙型、条件付きコンパイル ディレクティブ、および **const** ステートメントは、サーバー側だけでサポートされています。このため、これらの機能はサーバー側のスクリプトだけで使用します。詳細については、「JScript のバージョン情報」を参照してください。

スクリプトをブラウザ (クライアント側) で実行する場合、経験の豊富な開発者は、スクリプト エンジンのバージョンを検知するコードを挿入します。スクリプトでエンジンのバージョンを検知した後、ブラウザのスクリプト エンジンと互換性のあるスクリプトを使用したページにブラウザをリダイレクトできます。詳細については、「ブラウザの機能の検出」を参照してください。

JScript では、ブラウザの警告、確認、入力の各メッセージ ボックスを使用して、ユーザーからの入力を受け取ります。これらのボックスを表示するには、**window** オブジェクトのメソッドを使用します。**window** オブジェクトはオブジェクト階層構造の最上位に位置するため、実際には、どのメッセージ ボックスを表示する場合も、完全な名前 (`window.alert()` など) を使用する必要はありません。ただし、これらのボックスがどのオブジェクトに属するのかを覚えておくために、完全な名前を使用することをお勧めします。

警告メッセージ ボックス

警告メッセージ ボックスを表示する **alert** メソッドの引数は、1 つです。この引数には、警告メッセージ ボックスに表示するテキスト文字列を指定します。指定する文字列は、HTML ではありません。メッセージ ボックスには、ボックスを閉じるための [OK] ボタンが表示されます。メッセージ ボックスはモーダルであるため、ユーザーが作業を続行するにはメッセージ ボックスを閉じる必要があります。

```
window.alert("Welcome! Press OK to continue.");
```

確認メッセージ ボックス

確認メッセージ ボックスには、[OK] ボタンおよび [キャンセル] ボタンと、2 とおりの回答を持つ質問が表示されます。それに応じて、**confirm** メソッドから真 (**true**) または偽 (**false**) が返されます。このメッセージ ボックスもモーダルで、作業を続行する前に、ボタンをクリックしてメッセージ ボックスを閉じる必要があります。

```
var truthBeTold = window.confirm("Click OK to continue. Click Cancel to stop.");
if (truthBeTold)
   window.alert("Welcome to our Web page!");
else
   window.alert("Bye for now!");
```

入力メッセージ ボックス

入力メッセージ ボックスには、[OK] ボタンおよび [キャンセル] ボタンと、ユーザーが入力要求に応じてテキストを入力するためのテキスト フィールドが表示されます。このメソッドに 2 番目の引数 (文字列) を渡すと、その文字列が既定の回答としてテキスト フィールドに表示されます。2 番目の引数 (文字列) を渡さなかった場合、既定のテキストは "undefined" となります。

alert メソッドや confirm メソッドと同様、prompt メソッドで表示されるメッセージ ボックスもモーダル型です。ユーザーは作業を続行する前に、メッセージ ボックスを閉じる必要があります。

```
var theResponse = window.prompt("Welcome?","Enter your name here.");
document.write("Welcome "+theResponse+".<BR>");
```

参照

概念

ブラウザへの情報表示

ブラウザの機能の検出

その他の技術情報

JScript での情報の表示

正規表現の概説

次のセクションでは、正規表現の概念と、JScript で正規表現を作成および使用する方法を説明します。

各トピックは独立したものですが、これらのトピックを順に読み進めることでより深い理解が得られます。多くのトピックは、前のトピックで説明されている機能や概念の理解を基に説明されています。

このセクションの内容

正規表現

多くのユーザーが知っている概念との比較により、正規表現の概念について説明します。

正規表現の使用方法

実際的な例を通じて、正規表現で標準的な検索条件を拡張する方法を示します。

正規表現の構文

正規表現を構成する文字、メタ文字を構成する文字、およびメタ文字の動作を説明します。

正規表現の作成

正規表現のコンポーネント、およびコンポーネントと区切り文字の関係を説明します。

優先順位

正規表現の評価方法と、正規表現の評価順序と構文が結果に与える影響を説明します。

通常文字

通常文字とメタ文字を識別し、単一文字の正規表現を組み合わせて、より大きな表現を作成する方法を説明します。

JScript の特殊文字

エスケープ文字の概念と、メタ文字に対応する正規表現を作成する方法を説明します。

印字できない文字

正規表現中で印字されない文字を表すために使用する、エスケープ シーケンスの一覧を示します。

文字の一致

正規表現で、特殊な結果を返すピリオド、エスケープ文字、および順序を作成するかっこの使用方法を説明します。

JScript の量指定子

一致する文字数を指定できない場合に正規表現を作成する方法を説明します。

アンカー

正規表現を行頭または行末に固定する方法、および正規表現を単語内、単語の先頭、または単語の末尾に指定する方法を示します。

代替とグループ化

"|" 文字を使用して 2 つ以上の代替表現から選択する方法、およびより正確な結果を得るために代替表現とグループ化を組み合わせる方法を説明します。

JScript での逆参照

一致したパターンを検出した正規表現を作成し直すことなく、記憶されているパターンの一部にアクセスする正規表現を作成する方法を説明します。

関連するセクション

.NET Framework の正規表現

正規表現のパターン一致表記方法を説明します。これにより、開発者は大量のテキストをすばやく解析して特定の文字パターンを検出したり、テキスト部分文字列を抽出、編集、置換、または削除できます。また、レポートを生成するために、抽出した文字列をコレクションに追加できます。

正規表現の例

一般的なアプリケーションでの正規表現の使用方法を説明した、コード例へのリンクの一覧を示します。

正規表現

これまでに正規表現を使用したことがないユーザーは、"正規表現" という用語自体を見慣れていない場合もあります。しかし、スクリプト以外の領域では、いくつかの正規表現を使用したことがあると思われます。

正規表現の例

たとえば、ハード ディスク上のファイルを検索するために、? や * などのワイルドカード文字を使用することがあります。? ワイルドカード文字はファイル名の 1 文字に一致し、* は 0 個以上の文字に一致します。たとえば、data?.dat というパターンを指定すると、次のファイルが検索されます。

data1.dat

data2.dat

datax.dat

dataN.dat

? の代わりに * を指定すると、ファイルの検索対象が広がります。data*.dat というパターンは、次のファイルすべてに一致します。

data.dat

data1.dat

data2.dat

data12.dat

datax.dat

dataXYZ.dat

このような検索方法は便利ですが、使用できる範囲は限られています。ワイルドカード文字の ? と * は、正規表現の概念を理解する上での助けになりますが、正規表現の機能はより豊富で、高い柔軟性を備えています。

参照

その他の技術情報

正規表現の概説

正規表現の使用方法

一般的な検索および置換操作では、検索するテキストを正確に指定する必要があります。静的テキストを対象とする単純検索や置換作業にはこの方法で十分ですが、動的テキストでは柔軟性に欠け、少なくとも検索は困難になります。

サンプル シナリオ

正規表現を使用すると、次のことができます。

- 文字列内のパターンのテスト。

  たとえば、入力文字列をテストして、文字列内に電話番号またはクレジットカード番号のパターンがあるかどうかをチェックできます。これをデータ確認と呼びます。

- 文字列の置換。

  正規表現を使用して、ドキュメント内にある文字列を特定し、削除したり、他の文字列と置換したりできます。

- パターン一致に基づく、文字列からの部分文字列の抽出。

  ドキュメントまたは入力フィールド内の特定の文字列を検索できます。

たとえば、Web サイト全体を検索して、古い内容を削除したり、HTML タグを置換したりする必要がある場合について考えてみます。この場合、正規表現を使用して、各ファイルに削除対象の内容や置換対象の HTML タグがあるかどうかを確認できます。この処理により、削除や変更の対象となる内容を含むファイルを絞り込みます。続いて、正規表現を使用して古い内容を削除できます。さらに、正規表現を使用して、タグを検索して置換できます。

正規表現は、JScript や C などの言語や、文字列処理の機能がわからない言語でも役に立ちます。

参照

その他の技術情報

正規表現の概説

正規表現の構文

正規表現は、通常の文字 (a ～ z など) と、"メタ文字" という特殊文字で構成される文字列のパターンです。パターンによって、テキストを検索するときに一致する 1 つ以上の文字列を指定します。

正規表現の例

正規表現	一致する内容
/^\s*$/	空行に一致します。
/\d{2}-\d{5}/	2 桁の数字、ハイフン、および 5 桁の数字で構成される ID 番号を検査します。
/<\s*(\S+)(\s[^>]*)?>[\s\S]*<\s*\/\1\s*>/	HTML タグに一致します。

次の表は、メタ文字の全リストと、正規表現におけるメタ文字の動作を示しています。

文字	説明
\	次に続く文字が特殊文字、リテラル、前方参照、または 8 進エスケープであることを示します。たとえば、'n' は文字 "n" と一致しますが、'\n' は改行文字と一致します。'\\' は "\" と、"\(" は "(" と一致します。
^	入力文字列の先頭と一致します。**RegExp** オブジェクトの **Multiline** プロパティが設定されている場合は、'\n' または '\r' の直後とも一致します。
$	入力文字列の末尾と一致します。**RegExp** オブジェクトの **Multiline** プロパティが設定されている場合は、'\n' または '\r' の直前とも一致します。
*	直前の文字または部分式と 0 回以上一致します。たとえば、zo* は "z" と "zoo" のどちらにも一致します。* は {0,} と同じ意味です。
+	直前の文字または部分式と 1 回以上一致します。たとえば、'zo+' は "zo" や "zoo" とは一致しますが、"z" とは一致しません。+ は {1,} と同じ意味です。
?	直前の文字または部分式と 0 回または 1 回一致します。たとえば、"do(es)?" は "do" または "does" の "do" と一致します。? は {0,1} と同じ意味です。
{*n*}	*n* には、0 以上の整数を指定します。直前の文字と正確に *n* 回一致します。たとえば、'o{2}' は "Bob" の 'o' とは一致しませんが、"food" の 2 つの 'o' とは一致します。
{*n*,}	*n* には、0 以上の整数を指定します。少なくとも *n* 回一致します。たとえば、'o{2,}' は "Bob" の "o" とは一致しませんが、"foooood" のすべての "o" とは一致します。'o{1,}' は 'o+' と同じ意味です。'o{0,}' は 'o*' と同じ意味です。
{*n*,*m*}	*m* および *n* には 0 以上の整数を指定します。*n* は *m* 以下です。*n* ～ *m* 回一致します。たとえば、"o{1,3}" は "fooooood" の最初の 3 つの "o" と一致します。'o{0,1}' は 'o?' と同じ意味です。コンマと数の間には空白を入れません。
?	他の量指定子 (*, +, ?, {*n*}, {*n*,}, {*n*,*m*}) の直後に指定すると、一致パターンが最短一致になります。既定のパターンでは、できるだけ多くの文字列と一致するのに比べて、最短一致パターンでは、できるだけ少ない文字列と一致します。たとえば、文字列 "oooo" に対して、'o+?' を指定すると 1 つの "o" と一致し、'o+' を指定するとすべての "o" と一致します。
.	"\n" を除く任意の 1 文字に一致します。'\n' を含めて任意の文字と一致させるには、'[\s\S]' などのパターンを指定します。

(*pattern*)	pattern と一致した文字列を記憶する部分式です。一致する文字列が見つかったら記憶され、その部分は Matches コレクションから **$0 ...$9** プロパティを使用して取得できます。かっこ ( ) と一致させるには、'\(' または '\)' を指定します。
(?:*pattern*)	*pattern* と一致しても、その文字列が記憶されない部分式です。後で使用できません。"または" を意味する (|) を使用して、パターンの一部を結合するときに便利です。たとえば、"industry|industries" と指定する代わりに、"industr(?:y|ies)" と指定する方が簡潔です。
(?=*pattern*)	肯定先読みを実行する部分式です。*pattern* に一致する文字列が始まる位置にある検索文字列と一致します。一致した文字列は記憶されず、後で使用することはできません。たとえば "Windows (?=95|98|NT|2000)" は、"Windows 2000" の "Windows" には一致しますが、"Windows 3.1" の "Windows" には一致しません。先読み処理では、読み進まれた文字は処理済みとは見なされません。一致の検出後、次の検索処理は先読みされた文字列の後からではなく、一致文字列のすぐ後から開始されます。
(?!*pattern*)	否定先読み検索を実行する部分式です。*pattern* に一致しない文字列が始まる位置にある検索文字列と一致します。一致した文字列は記憶されず、後で使用することはできません。たとえば "Windows (?!95|98|NT|2000)" は、"Windows 3.1" の "Windows" には一致しますが、"Windows 2000" の "Windows" には一致しません。先読み処理では、読み進まれた文字は処理済みとは見なされません。一致の検出後、次の検索処理は先読みされた文字列の後からではなく、一致文字列のすぐ後から開始されます。
x|*y*	*x* または *y* と一致します。たとえば、"z|food" は "z" または "food" と一致します。"(z|f)ood" は、"zood" または "food" に一致します。
[*xyz*]	文字セットを指定します。角かっこで囲まれた文字のいずれかに一致します。たとえば、'[abc]' は "plain" の 'a' と一致します。
[^*xyz*]	除外する文字セットを指定します。角かっこで囲まれた文字以外の文字に一致します。たとえば、'[^abc]' は "plain" の 'p' と一致します。
[*a-z*]	文字の範囲を指定します。指定した範囲に含まれる任意の文字に一致します。たとえば、'[a-z]' は小文字の英字 a ～ z の範囲にある任意の文字と一致します。
[^*a-z*]	否定の文字の範囲を指定します。指定した範囲に含まれていない任意の文字に一致します。たとえば、'[^a-z]' は小文字の英字 a ～ z の範囲外にある任意の文字と一致します。
\b	単語の境界と一致します。単語の境界とは、単語と空白との間の位置のことです。たとえば、'er\b' は "never" の 'er' と一致しますが、"verb" の 'er' とは一致しません。
\B	単語の境界以外と一致します。たとえば、'er\B' は "verb" の 'er' と一致しますが、"never" の 'er' とは一致しません。
\c*x*	*x* で指定した制御文字と一致します。たとえば、\cM は Control-M または復帰文字と一致します。*x* の値は、A ～ Z または a ～ z の範囲内で指定します。それ以外を指定すると、リテラル文字 'c' と認識されます。
\d	任意の 10 進文字と一致します。[0-9] と同じ意味です。
\D	10 進数字以外の任意の文字と一致します。[^0-9] と同じ意味です。
\f	フォーム フィード文字と一致します。\x0c および \cL と同じ意味です。

\n	改行文字と一致します。\x0a および \cJ と同じ意味です。
\r	キャリッジ リターン文字と一致します。\x0d および \cM と同じ意味です。
\s	空白、タブ、フォーム フィードなどの任意の空白文字と一致します。[\f\n\r\t\v] と同じ意味です。
\S	空白文字以外の任意の文字と一致します。[^ \f\n\r\t\v] と同じ意味です。
\t	タブ文字と一致します。\x09 および \cI と同じ意味です。
\v	垂直タブ文字と一致します。\x0b および \cK と同じ意味です。
\w	単語に使用される任意の文字と一致します。アンダースコアも含まれます。'[A-Za-z0-9_]' と同じ意味です。
\W	単語に使用される文字以外の任意の文字と一致します。'[^A-Za-z0-9_]' と同じ意味です。
\x*n*	*n* (16 進数のエスケープ値) と一致します。16 進数のエスケープ値は 2 桁である必要があります。たとえば、'\x41' は "A" と一致します。'\x041' は '\x04' および "1" と同じ意味です。この表記により、正規表現で ASCII コードを使用できるようになります。
num	*num* (正の整数) と一致します。既に見つかって記憶されている部分を参照します。たとえば、'(.)\1' は、連続する 2 つの同じ文字と一致します。
n	8 進数のエスケープ値または前方参照を指定します。*n* の前に少なくとも *n* 個の記憶された部分式がある場合、*n* は前方参照になります。それ以外の場合で *n* が 8 進数値 (0 ～ 7) である場合は、*n* は 8 進エスケープ値です。
nm	8 進数のエスケープ値または前方参照を指定します。*nm* の前に少なくとも *nm* 個の記憶された部分式がある場合、*nm* は前方参照になります。*nm* の前に少なくとも *n* 個の記憶された部分式がある場合は、*n* が前方参照で、リテラル *m* が後に続きます。どちらの条件にも当てはまらない場合で *n* および *m* が 8 進数 (0 ～ 7) である場合、*nm* は 8 進数のエスケープ値 *nm* と一致します。
nml	*n* が 8 進数値の 0 ～ 3 で、*m* と *l* が 8 進数値 (0 ～ 7) の場合は、8 進エスケープ値 *nml* と一致します。
\u*n*	*n* と一致します。*n* には Unicode 文字で表した 4 桁の 16 進数を指定します。たとえば、\u00A9 は著作権の記号 (©) と一致します。

参照

その他の技術情報

正規表現の概説

正規表現の作成

正規表現の作成方法は、算術式の作成方法と似ています。つまり、さまざまなメタ文字と演算子で小さな式を組み合わせて大きな式にします。

区切り記号

正規表現では、表現パターンのさまざまな要素を 1 対の区切り記号で囲みます。JScript では、スラッシュ文字 (/) を区切り記号として使用します。次に例を示します。

```
/expression/
```

上の例では、正規表現のパターン (*expression*) は **RegExp** オブジェクトの **Pattern** プロパティに保存されます。

正規表現の要素には文字、文字セット、文字の範囲、または選択した文字を使用でき、これらを組み合わせることもできます。

参照

その他の技術情報

正規表現の概説

優先順位

正規表現は、算術式と同じ要領で左から右に評価され、優先順位も適用されます。

演算子

正規表現の各演算子の優先順位を高いものから低いものの順に次の表に示します。

演算子	説明
\	エスケープ文字
(), (?:), (?=), []	かっこおよび角かっこ
*、+、?、{n}、{n,}、{n,m}	量指定子
^, $, \メタ文字、任意の文字	アンカーおよびシーケンス
|	代替

文字は代替演算子よりも高い優先順位を持ちます。したがって、"m|food" は "m" または "food" に一致します。"mood" または "food" に一致させるには、かっこを使って "(m|f)ood" という部分式を作成します。

参照

その他の技術情報

正規表現の概説

通常文字

通常文字は、メタ文字として割り当てられていない文字で、印刷できる文字と印刷できない文字のすべての文字で構成されます。大文字および小文字のすべてのアルファベット、すべての数字、すべての句読点、およびいくつかの記号が含まれます。

単純な式

正規表現の最も単純な形式は、検索文字列の 1 文字と一致する通常文字です。たとえば、A などの 1 文字のパターンは、検索文字列に A が見つかるたびに一致します。1 文字の正規表現パターンの例を次に示します。

```
/a/
/7/
/M/
```

1 文字の正規表現をいくつかまとめて大きな式にすることもできます。たとえば次の正規表現は、1 文字のパターンの a、7、および M を組み合わせています。

```
/a7M/
```

この場合、連結演算子は使用しません。各文字を続けて入力します。

参照

その他の技術情報

正規表現の概説

JScript の特殊文字

メタ文字と一致させる場合、多くのメタ文字には特別な対処が必要です。これらの特殊文字と一致させるには、円記号 (\) を先頭に指定して、最初にこの文字を "エスケープ" する必要があります。次の表は、特殊文字とその意味を示しています。

特殊文字の表

特殊文字	コメント
$	入力文字列の末尾と一致します。**RegExp** オブジェクトの **Multiline** プロパティが設定されている場合は、\n または \r の直前とも一致します。$ と一致させる場合は \$ と指定します。
( )	部分式の開始と終了を指定します。部分式は記憶され、後で使用できます。かっこと一致させる場合は、\ (および \) と指定します。
*	直前の文字または部分式と 0 回以上一致します。* と一致させる場合は * と指定します。
+	直前の文字または部分式と 1 回以上一致します。+ と一致させる場合は \+ と指定します。
.	改行文字 \n 以外の 1 文字と一致します。"." と一致させる場合は \. と指定します。
[ ]	角かっこの式の開始を指定します。角かっこと一致させる場合は、\[および \] と指定します。
?	直前の文字または部分式と 0 回または 1 回一致するか、最短量指定子を意味します。? と一致させる場合は \? と指定します。
\	次に続く文字が特殊文字、リテラル、前方参照、または 8 進エスケープであることを示します。たとえば、n は文字の n に一致しますが、\n は改行文字に一致します。\\ ? \ ??\(? (???????
/	リテラル正規表現の開始または終了を示します。/ と一致させる場合は、\/ と指定します。
^	入力文字列の先頭と一致します。かっこの中で使用した場合は、かっこ内の文字セットを否定します。^ と一致させる場合は \^ と指定します。
{ }	量指定子の式の開始を指定します。中かっこと一致させる場合は、\{ および \} と指定します。
|	2 つの項目から選択します。| と一致させる場合は、| と指定します。

参照

その他の技術情報

正規表現の概説

印字できない文字

正規表現には、印字できない文字も指定できます。印字できない文字を表すエスケープ シーケンスを次の表に示します。

エスケープ シーケンス

文字	説明
\c*x*	*x* で指定した制御文字と一致します。たとえば、\cM は Control-M または復帰文字と一致します。*x* の値は、A ～ Z または a ～ z の範囲内で指定します。それ以外を指定すると、リテラル文字の c と認識されます。
\f	フォーム フィード文字と一致します。\x0c および \cL と同じ意味です。
\n	改行文字と一致します。\x0a および \cJ と同じ意味です。
\r	キャリッジ リターン文字と一致します。\x0d および \cM と同じ意味です。
\s	空白、タブ、フォーム フィードなどの任意の空白文字と一致します。[\f\n\r\t\v] と同じ意味です。
\S	空白文字以外の任意の文字と一致します。[^ \f\n\r\t\v] と同じ意味です。
\t	タブ文字と一致します。\x09 および \cI と同じ意味です。
\v	垂直タブ文字と一致します。\x0b および \cK と同じ意味です。

参照

その他の技術情報

正規表現の概説

文字の一致

ピリオド (.) は、文字列内の印字できる文字または印字できない文字のうち、任意の 1 文字と一致します。ただし、改行文字 (\n) とは一致しません。次の正規表現は、aac、abc、acc、adc、a1c、a2c、a-c、a#c などと一致します。

```
/a.c/
```

ファイル名のように、入力文字列の一部にピリオド (.) を含む文字列と一致させる場合は、正規表現のピリオドの前に円記号 (\) を付加します。たとえば、次の正規表現は filename.ext と一致します。

```
/filename\.ext/
```

これらの正規表現は、任意の 1 文字とだけ一致します。リストに指定した複数の文字に一致させることもできます。たとえば、数字を使った章の見出し (Chapter 1、Chapter 2 など) も検索できます。

角かっこの表現

一致文字のリストを作成するには、1 つ以上の文字を角かっこ ([と]) で囲みます。角かっこで囲んだ文字のリストは "角かっこ表現" と呼びます。角かっこ内では、他の場合と同じように、通常文字はその文字自体を表します。つまり、入力文字列にその文字の一致候補がある場合に一致します。角かっこ表現内では、大半の特殊文字の意味は失われます。ただし、次の例外があります。

-] は、最初の項目でない場合はリストを終了します。リストで] を一致させるには、開始の [の直後に記述します。
- \ はエスケープ文字として機能します。\ 文字と一致させるには、\\ を使用します。

角かっこ表現内の文字は、一致候補の 1 文字だけと一致します。たとえば、次の正規表現は Chapter 1、Chapter 2、Chapter 3、Chapter 4、および Chapter 5 と一致します。

```
/Chapter [12345]/
```

Chapter およびその直後の空白の位置は、角かっこ内の文字を基準にして固定されます。角かっこ表現は、Chapter と空白の直後の 1 文字に一致する文字群だけを指定するために使用されます。この場合は 9 文字目がそれに当たります。

文字自体ではなく、文字の範囲を指定して一致文字を表現するには、範囲の開始文字と終了文字をハイフン (-) でつなぎます。指定する各文字の文字値によって、範囲内の相対位置が決まります。次の正規表現は、上に示した角かっこリストと同じ意味の範囲表現です。

```
/Chapter [1-5]/
```

この範囲指定では、範囲の開始文字と終了文字も範囲に含まれます。開始文字は Unicode の並べ替え順序において、終了文字よりも前である必要があります。

角かっこ表現でハイフンを指定するには、次のいずれかの方法を使用します。

- 円記号でエスケープする。

  ```
  [\-]
  ```

- ハイフンを角かっこリストの先頭または最後に指定する。次の正規表現は、すべての小文字とハイフンに一致します。

  ```
  [-a-z]
  [a-z-]
  ```

- 範囲の開始文字をコード順でハイフンよりも前の文字にし、終了文字をハイフンと等しいかハイフンよりも後の文字にする。次の正規表現は、どちらもこの条件を満たしています。

  ```
  [!--]
  [!-~]
  ```

リストまたは範囲にないすべての文字を検索するには、リストの先頭にカレット (^) を指定します。リストの先頭以外に指定したカレットは、カレット文字と一致します。次の正規表現は、5 を超える番号の章見出し (Chapter) と一致します。

```
/Chapter [^12345]/
```

上の例では、9 文字目が 1、2、3、4、または 5 以外のものと一致します。たとえば、Chapter 7 や Chapter 9 と一致します。

上の正規表現は、ハイフン (-) を使用して表すこともできます。

```
/Chapter [^1-5]/
```

角かっこ表現の一般的な使用方法として、任意の大文字または小文字のアルファベット、または任意の数字と一致させる場合があります。この正規表現の例を次に示します。

```
/[A-Za-z0-9]/
```

参照

その他の技術情報

正規表現の概説

JScript の量指定子

一致する文字数を指定できない場合は、正規表現の量指定子の概念を利用します。これらの量指定子を使用して、正規表現の任意の要素を何回一致させるかを指定できます。

量指定子の意味

文字	説明
*	直前の文字または部分式と 0 回以上一致します。たとえば、zo* は z とも zoo とも一致します。* は {0,} と同じ意味です。
+	直前の文字または部分式と 1 回以上一致します。たとえば、zo+ は zo や zoo とは一致しますが、z とは一致しません。+ は {1,} と同じ意味です。
?	直前の文字または部分式と 0 回または 1 回一致します。たとえば、do(es)? は do または does の do と一致します。? は {0,1} と同じ意味です。
{*n*}	*n* には、0 以上の整数を指定します。直前の文字と正確に *n* 回一致します。たとえば、o{2} は、Bob の o とは一致しませんが、food の 2 つの o とは一致します。
{*n*,}	*n* には、0 以上の整数を指定します。少なくとも *n* 回一致します。たとえば、o{2,} は Bob の o とは一致しませんが、foooood のすべての o とは一致します。o{1,} は o+ と同じ意味になり、o{0,} は o* と同じ意味です。
{*n*, *m*}	*m* および *n* には 0 以上の整数を指定します。*n* は *m* 以下です。*n* ～ *m* 回一致します。たとえば、o{1,3} は fooooood の最初の 3 つの o と一致します。o{0,1} は o? と同じ意味です。コンマと数の間には空白を入れません。

大きなドキュメントでは章番号が 9 を超える場合があるため、2 桁または 3 桁の章番号を扱えるようにする必要があります。この場合にも量指定子を使用します。次の正規表現は、任意の桁の章見出し (Chapter) と一致します。

```
/Chapter [1-9][0-9]*/
```

量指定子は範囲表現の後に指定します。したがって、0 ～ 9 だけの章を含むすべての範囲表現に適用されます。

2 文字目以降は必ずしも数字である必要はないため、+ 量指定子は使用しません。? も、桁数が 2 桁に固定されてしまうため使用しません。Chapter と空白文字に続く数字が少なくとも 1 桁あるものと一致します。

章番号が 99 以下であることが確実である場合は、次の正規表現を使用して、章番号が 1 桁以上 2 桁以下であるものと一致させることができます。

```
/Chapter [0-9]{1,2}/
```

上の正規表現の欠点は、章番号が 99 を超える場合も、その先頭の 2 桁と一致してしまう点です。また、Chapter 0 にも一致します。2 桁の章番号だけに一致するよう改善した正規表現を次に示します。

```
/Chapter [1-9][0-9]?/
```

または、次のようにも指定できます。

```
/Chapter [1-9][0-9]{0,1}/
```

*、+、? の各量指定子は、できるだけ多くの文字列と一致するため、"最長一致" と言われます。ただし、最短一致が望ましい場合もあります。

たとえば、HTML ドキュメントの H1 タグで囲まれている章タイトルを検索する場合を考えます。ドキュメントでは次のように記述されています。

```
<H1>Chapter 1 – Introduction to Regular Expressions</H1>
```

次の正規表現は、H1 タグの開始記号 (<) から終了記号 (>) までのすべてと一致します。

```
/<.*>/
```

H1 開始タグだけと一致させるには、次の正規表現を使用します。この場合は <H1> だけと一致します。

```
/<.*?>/
```

*、+、? の各量指定子の後に ? を指定すると、最長一致から最短一致に変わります。

参照

その他の技術情報

正規表現の概説

アンカー

これらの例では、章見出しを検索することにだけに注目しました。文字列 Chapter、空白、および章番号の組み合わせは、実際の章見出しである場合もあれば、別の章への参照を示している場合もあります。実際の章見出しは必ず行頭から始まるため、これを利用して参照を表す章見出しを除外し、実際の章見出しだけを検索します。

アンカーの働き

この場合には、アンカーを使用します。アンカーを使用すると、正規表現の一致箇所を行頭または行末に固定できます。また、単語内、単語の先頭、または単語の末尾と一致する正規表現を作成することもできます。正規表現のアンカーとその意味を次の表に示します。

文字	説明
^	入力文字列の先頭と一致します。**RegExp** オブジェクトの **Multiline** プロパティが設定されている場合は、\n または \r の直後とも一致します。
$	入力文字列の末尾と一致します。**RegExp** オブジェクトの **Multiline** プロパティが設定されている場合は、\n または \r の直前とも一致します。
\b	単語の境界と一致します。単語の境界とは、単語と空白との間の位置のことです。
\B	単語の境界以外と一致します。

量指定子とアンカーは一緒に使用できません。改行コードまたは単語の境界の前後を同時に指定することはできないため、^* などの正規表現は使用できません。

行頭の文字列と一致させるには、正規表現の先頭で ^ を指定します。角かっこ内で使用する ^ とは異なるため、間違えないようにしてください。

行末の文字列と一致させるには、正規表現の最後で $ を指定します。

アンカを使用した次の正規表現で章見出しを検索すると、行頭から始まる章見出しのうち、章番号が 2 桁までのものと一致します。

```
/^Chapter [1-9][0-9]{0,1}/
```

章見出しは行の先頭であるだけでなく、行全体が章見出しとなります。行の先頭であると同時に、同じ行の行末にもなります。次の正規表現では、参照を除外した、実際の章見出しだけに一致します。行の先頭と最後だけに一致するように正規表現で指定しています。

```
/^Chapter [1-9][0-9]{0,1}$/
```

単語の境界と一致させる場合は少し異なりますが、正規表現では重要な機能です。単語の境界とは、単語と空白との間の位置のことです。それ以外の位置は、単語の境界ではありません。次の正規表現は、Chapter の先頭の 3 文字と一致します。単語の境界の直後に指定しているためです。

```
/\bCha/
```

\b 演算子の位置が重要になります。一致させる文字列の先頭に演算子を指定した場合は、単語の先頭で一致する部分を検索します。演算子を文字列の末尾に指定した場合は、単語の末尾で一致する部分を検索します。たとえば、次の正規表現は、単語 Chapter の文字列 ter と一致します。単語の境界の前に指定しているためです。

```
/ter\b/
```

次の正規表現は、単語 Chapter の文字列 apt と一致しますが、aptitude の apt とは一致しません。

```
/\Bapt/
```

Chapter の apt は単語の境界にありませんが、aptitude の apt は単語の境界にあります。単語の境界ではない位置を指定する \B 演算子

では、単語の先頭または最後を基準にしないため、演算子を指定する位置は関係ありません。

参照

その他の技術情報

正規表現の概説

代替とグループ化

| 文字を使用して、2 つ以上の代替表現から選択します。たとえば、章見出しの正規表現を拡張して、章見出し以外のものとも一致させることができます。ただし、それほど単純ではありません。代替を使用すると、| で区切られた表現のうち最も大きい表現と一致します。

例

次に示す正規表現は、1 桁または 2 桁の数字が続く Chapter または Section のうち、行の先頭または最後にあるものと一致するように思われます。

```
/^Chapter|Section [1-9][0-9]{0,1}$/
```

実際には、上に示した正規表現は、行頭にある単語 Chapter、または後に任意の桁数の数字が続く Section のうち行の最後にあるものと一致します。入力文字列が Chapter 22 である場合、上の正規表現は Chapter としか一致しません。入力文字列が Section 22 である場合は、Section 22 と一致します。

正規表現をよりわかりやすくするには、かっこを使用して代替の範囲を制限し、代替表現が Chapter と Section の 2 つの単語だけに適用されるようにします。ただし、かっこは部分式を作成して、後で使用するように記憶する場合にも使用されます。部分式については、前方参照のセクションで説明しています。上の正規表現の適切な位置にかっこを追加して、Chapter 1 または Section 3 のどちらにも一致する正規表現を作成します。

次に示す正規表現では、かっこを使用して Chapter と Section をグループ化し、適切に動作するようにしています。

```
/^(Chapter|Section) [1-9][0-9]{0,1}$/
```

この正規表現は正しく動作しますが、Chapter|Section を囲むかっこにより、一致した単語が後で使用できるように記憶されます。上の正規表現では、かっこは 1 組だけであるため、記憶される "サブマッチ" も 1 つだけです。このサブマッチは、**RegExp** オブジェクトの **$1** ～ **$9** プロパティを使用して参照できます。

上の例では、単語の Chapter と Section をグループ化するためだけにかっこを使用しています。一致結果を記憶させないようにするには、かっこ内の正規表現パターンの前に ?: を指定します。次のように修正すると、サブマッチを記憶せずに同じ機能を得られます。

```
/^(?:Chapter|Section) [1-9][0-9]{0,1}$/
```

メタ文字 ?: の他にも、"先読み" 一致を作成する、サブマッチを記憶しない 2 つのメタ文字があります。?= を指定した肯定先読みは、かっこ内の正規表現と一致する文字列が始まる位置にある検索文字列と一致します。?! を指定した否定先読みは、正規表現と一致しない文字列が始まる位置にある検索文字列と一致します。

たとえば、Windows 3.1、Windows 95、Windows 98、および Windows NT への参照項目があるドキュメントを考えます。Windows 95、Windows 98、および Windows NT への参照をすべて Windows 2000 への参照に変更して、ドキュメントを更新する必要があるとします。次の正規表現は、肯定先読みを使用して、Windows 95、Windows 98、および Windows NT と一致します。

```
/Windows(?=95 |98 |NT )/
```

文字列の次の検索は、先読みに含まれる文字ではなく、パターンが一致した文字列の直後から開始されます。たとえば、上の正規表現で Windows 98 と一致した場合、検索は 98 の直後ではなく、Windows の直後から再開されます。

参照

概念

JScript での逆参照

その他の技術情報

正規表現の概説

JScript での逆参照

正規表現の最も重要な機能の 1 つに、一致したパターンの一部を後で使用できるように記憶しておくことがあります。前述したように、正規表現のパターンまたはパターンの一部をかっこで囲むと、かっこで囲んだ部分が一時バッファに保存されます。一致したパターンを記憶しないメタ文字の ?:、?=、または ?! を使用すると、記憶されたパターンを上書きできます。

前方参照の使用

記憶された各サブマッチは、正規表現パターンの左から右の順に保存されます。部分式が保存されるバッファ番号は 1 から始まり、最大 99 までです。各バッファにアクセスするには、*n* を使用します。ここで、*n* はバッファを指定する 1 桁または 2 桁の数字です。

前方参照の最も簡単で便利な用途は、文字列内で隣接している 2 つの同じ単語の一致候補を特定することです。次に例を示します。

```
Is is the cost of of gasoline going up up?
```

上の文では、いくつかの単語が重複しています。単語を 1 つずつ調べていくことなく、このような文を修正できると便利です。次の正規表現では、これを実現するために 1 つの部分式を使用しています。

```
/\b([a-z]+) \1\b/gi
```

記憶されたサブマッチには、[a-z]+ で指定されているように、1 つ以上のアルファベットが含まれています。正規表現の 2 番目の部分は、その前に記憶されたサブマッチへの参照です。つまり、かっこ内の正規表現と一致した 2 つ目の一致候補です。\1 は、1 つ目のサブマッチを表します。単語の境界を表すメタ文字によって、単語だけが検出されます。メタ文字を指定しないと、正規表現によって "is issued" や "this is" なども間違って検出されてしまいます。

通常の正規表現に続けてグローバル フラグ (g) を指定して、入力文字列にある一致候補とできるだけ多く一致するようにします。正規表現の最後に i フラグを指定して、大文字小文字を区別しないようにします。複数行フラグは、改行文字の前後に一致候補が存在する場合があることを表します。

上に示した正規表現を使用すると、次のコードでサブマッチ情報を使用して、テキスト内にある 2 つの連続したまったく同じ単語が、同じ単語 1 つと置換されます。

```
var ss = "Is is the cost of of gasoline going up up?.\n";
var re = /\b([a-z]+) \1\b/gim;       //Create regular expression pattern.
var rv = ss.replace(re,"$1");   //Replace two occurrences with one.
```

replace メソッドで **$1** を使用しているのは、1 番目に保存されたサブマッチを参照するためです。サブマッチが複数ある場合は、続けて **$2**、**$3** などを使用して参照します。

また、前方参照を使用して、URI (Universal Resource Indicator) をコンポーネントに分割できます。次の URI をプロトコル (ftp、http など)、ドメイン アドレス、およびページ/パスに分割するとします。

```
http://msdn.microsoft.com:80/scripting/default.htm
```

次の正規表現を使用して分割します。

```
/(\w+):\/\/([^/:]+)(:\d*)?([^# ]*)/
```

1 番目のかっこの部分式で、Web アドレスのプロトコルの部分を取り出します。この部分式は、コロンと 2 つのスラッシュで始まる単語と一致します。2 番目のかっこの部分式は、アドレスのドメイン アドレスの部分を取り出します。この部分式は、/ または : 以外の、1 文字以上の文字と一致します。3 番目のかっこの部分式は、ポート番号が指定されている場合に、その番号を取り出します。この部分式は、コロンに続く 0 桁以上の数字と一致します。この部分式は、1 回だけ繰り返すことができます。4 番目のかっこの部分式は、Web アドレスで指定されたパスとページの情報を取り出します。この部分式は、# または空白文字を含まない文字の並びと一致します。

この正規表現を上に示した URI に適用すると、サブマッチは次のようになります。

- **RegExp.$1** には "http" が記憶されます。
- **RegExp.$2** には "msdn.microsoft.com" が記憶されます。
- **RegExp.$3** には ":80" が記憶されます。

- **RegExp.$4** には "/scripting/default.htm" が記憶されます。

参照

その他の技術情報

正規表現の概説

JScript リファレンス

JScript プログラミング言語には、各種のプロパティ、メソッド、オブジェクト、関数などが含まれています。さらに、JScript では、対応する .NET Framework クラス ライブラリの多くの機能を利用できます。ここでは、これらの機能の適切な使用方法と、JScript での正しい構文を説明します。

このセクションの内容

機能の情報

JScript の言語機能に関するバージョン情報を示し、それらの機能を ECMAScript 規格で指定されている機能と比較します。

JScript 言語の紹介

JScript コードを記述するための要素やプロシージャを紹介します。また、言語要素やコード構文の背景についての詳細な説明へのリンクも示します。

JScript .NET 言語リファレンス

JScript オブジェクト指向プログラミング言語の基本的なコンポーネントの一覧と、言語の使用方法を説明するトピックへのリンクを示します。

JScript コンパイラ オプション

コマンド ライン コンパイラで使用可能なオプションの一覧を示し、オプションをアルファベット順またはカテゴリ別にまとめたトピックへのリンクを示します。

関連するセクション

.NET Framework リファレンス

.NET Framework クラス ライブラリおよびその他の基本要素の構文と構造を説明するトピックへのリンクを示します。

機能の情報

JScript プログラミング言語には、各種のプロパティ、メソッド、オブジェクト、関数などが含まれています。さらに、JScript では、対応する .NET Framework クラス ライブラリの多くの機能を利用できます。ここでは、これらの機能の適切な使用方法と、JScript での正しい構文を説明します。

このセクションの内容

ECMA 準拠の Microsoft JScript の機能

ECMAScript 言語仕様の一部である JScript の言語機能を説明します。

ECMA に準拠しない JScript の機能

ECMAScript 言語仕様に含まれていない JScript の言語機能を説明します。

バージョン情報

JScript の各バージョンの違いを示します。

関連するセクション

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

ECMA 準拠の Microsoft JScript の機能 (JScript)

JScript には、ECMAScript Edition 3 言語仕様のほとんどの機能が組み込まれています。また、JScript は ECMAScript Edition 4 と共に開発され、この言語について提案された多くの機能も同様に組み込んでいます。次の表に、JScript でサポートされる、ECMAScript 3 の機能と提案された ECMAScript 4 の機能を示します。

JScript でサポートされる機能

カテゴリ	ECMAScript 3 の機能/キーワード
配列操作	Array concat、join、length、reverse、slice、sort
代入	代入 (=)、加算代入 (+=)、ビットごとの AND 代入 (&=)、ビットごとの OR 代入 (|=)、ビットごとの XOR 代入 (^=)、除算代入 (/=)、左シフト代入 (<<=)、剰余代入 (%=)、乗算代入 (*=)、右シフト代入 (>>=)、減算代入 (-=)、符号なし右シフト代入 (>>>=)
Boolean 型	Boolean、true、false
コメント	/*...*/ または //
定数とリテラル	NaN、null、Infinity、undefined
制御フロー	break、continue、do...while、for、for...in、if...else、Labeled、return、switch、while
日付/時刻	Date、getDate、getDay、getFullYear、getHours、getMilliseconds、getMinutes、getMonth、getSeconds、getTime、getTimezoneOffset、getYear、getUTCDate、getUTCDay、getUTCFullYear、getUTCHours、getUTCMilliseconds、getUTCMinutes、getUTCMonth、getUTCSeconds、setDate、setFullYear、setHours、setMilliseconds、setMinutes、setMonth、setSeconds、setTime、setYear、setUTCDate、setUTCFullYear、setUTCHours、setUTCMilliseconds、setUTCMinutes、setUTCMonth、setUTCSeconds、toGMTString、toLocaleString、toUTCString、parse、UTC
宣言	Function、new、this、var、with
エラー処理	Error、description、number、throw、try...catch
関数の作成	caller、Function、arguments、length
グローバルメソッド	Global、escape、unescape、eval、isFinite、isNaN、parseInt、parseFloat
Math オブジェクト	Math、abs、acos、asin、atan、atan2、ceil、cos、exp、floor、log、max、min、pow、random、round、sin、sqrt、tan、E、LN2、LN10、LOG2E、LOG、PI、SQRT1_2、SQRT2
数値	Number、MAX_VALUE、MIN_VALUE、NaN、NEGATIVE_INFINITY、POSITIVE_INFINITY
オブジェクトの作成	Object、new、constructor、instanceof、prototype、toString、valueOf
演算子	加算 (+)、減算 (-)、剰余 (%)、乗算 (*)、除算 (/)、否定 (-)、等値 (==)、非等値 (!=)、小なり (<)、以下 (<=)、大なり (>)、以上 (>=)、論理 And (&&)、論理 Or (||)、論理 Not (!)、ビットごとの And (&)、ビットごとの Or (|)、ビットごとの Not (~)、ビットごとの Xor (^)、ビットごとの左シフト (<<)、ビットごとの右シフト、符号なし右シフト (>>>)、条件 (?:)、コンマ (,)、delete、typeof、void、デクリメント (--)、インクリメント (++)、厳密等価 (===)、厳密非等価 (!==)

オブジェクト	Array、Boolean、Date、Function、Global、Math、Number、Object、RegExp、Regular Expression、String
正規表現とパターン一致	RegExp、index、input、lastIndex、$1...$9、source、compile、exec、test、正規表現の構文
文字列	String、charAt、charCodeAt、fromCharCode、indexOf、lastIndexOf、split、toLowerCase、toUpperCase、length、concat、slice、match、replace、search、anchor、big、blink、bold、fixed、fontcolor、fontsize、italics、link、small、strike、sub、sup

カテゴリ	提案された ECMAScript 4 の機能/キーワード
クラス ベースのオブジェクト	class、extends、implements、interface、function get、function set、static、public、private、protected、internal、abstract、final、hide、override、static
宣言	const
列挙型	enum

参照

概念

ECMA に準拠しない JScript の機能 (JScript)

その他の技術情報

JScript リファレンス

ECMA に準拠しない JScript の機能 (JScript)

JScript には、ECMAScript Edition 3 のほとんどの機能と、ECMAScript Edition 4 で提案された機能の多くが組み込まれています。また JScript には、ECMAScript 言語では提供されていない、数多くの独自の機能があります。次の表に、これらの JScript に固有の機能を示します。

JScript に固有の機能

カテゴリ	機能/キーワード
配列操作	VBArray、dimensions、getItem、lbound、toArray、ubound
クラス ベースのオブジェクト	expando、super
条件付きコンパイル	@cc_on、@if ステートメント、@set ステートメント、@debug、@position、条件付きコンパイル変数
データ型	boolean、byte、char、decimal、double、float、int、long、Number、sbyte、short、String、uint、ulong、ushort
日付/時刻	getVarDate
情報の表示	print
列挙型	Enumerator、atEnd、item、moveFirst、moveNext
名前空間	package、import
オブジェクト	Enumerator、VBArray、ActiveXObject、GetObject
スクリプト エンジン ID	ScriptEngine、ScriptEngineBuildVersion、ScriptEngineMajorVersion、ScriptEngineMinorVersion

参照

概念

ECMA 準拠の Microsoft JScript の機能 (JScript)

その他の技術情報

JScript リファレンス

JScript のバージョン情報

JScript は発展し続ける言語であり、新しいバージョンごとに新しい機能が導入されています。特定のバージョンで提供されるすべての機能を利用するには、互換性のあるバージョンのコンパイラまたはスクリプト エンジンが必要です。

JScript のバージョンとホスト アプリケーション

サーバー側アプリケーションやコマンド ライン プログラムを作成する場合、コンパイラのバージョンやコンパイラがサポートする JScript のバージョンは、通常は明らかになっています。ただし、ブラウザのスクリプト エンジンで動作するクライアント側のスクリプトを作成する場合は、スクリプトの実行によりエンジンのバージョンが検出されます。エンジンのバージョンが判明すると、互換性のあるバージョンの JScript で記述されたスクリプトを実行できます。詳細については、「ブラウザの機能の検出」を参照してください。

次の表は、ホスト アプリケーションに実装される Microsoft JScript (以下 JScript) のバージョンです。

ホスト アプリケーション	1.0	2.0	3.0	4.0	5.0	5.1	5.5	5.6	.NET	8.0
Microsoft Internet Explorer 3.0	x									
Microsoft Internet Information Server 3.0		x								
Microsoft Internet Explorer 4.0			x							
Microsoft Internet Information Server 4.0			x							
Microsoft Internet Explorer 5.0					x					
Microsoft Internet Explorer 5.01						x				
Microsoft Windows 2000						x				
Microsoft Internet Explorer 5.5							x			
Microsoft Windows Millennium Edition							x			
Microsoft Internet Explorer 6.0								x		
Microsoft Windows XP								x		
Microsoft Windows Server 2003								x		
Microsoft .NET Framework 1.0									x	

メモ :

ScriptEngineMajorVersion 関数で取得されるバージョン番号と、**@_jscript_version** 条件付きコンパイル変数は、常に数値です。このため、バージョン番号との数値比較を実行できます。バージョンが .NET のアプリケーションの場合、バージョン番号は .NET ではなく 7.*x* になります。したがって、エンジンのバージョン番号が 7.*x* 以降である場合は、JScript 8.0 コードをコンパイルできます。

次の表は、JScript 言語の機能と、各機能が最初に導入されたバージョンの一覧です。

言語要素	1.0	2.0	3.0	4.0	5.0	5.5	.NET	8.0
0...n プロパティ						x		
$1...$9 プロパティ			x					
abs メソッド	x							

abstract 修飾子							x	
acos メソッド	x							
ActiveXObject オブジェクト			x					
加算演算子 (+)	x							
加算代入演算子 (+=)	x							
anchor メソッド	x							
apply メソッド						x		
arguments オブジェクト	x							
arguments プロパティ		x						
Array オブジェクト		x						
asin メソッド	x							
代入演算子 (=)	x							
atan メソッド	x							
atan2 メソッド	x							
atEnd メソッド			x					
big メソッド	x							
ビットごとの AND 演算子 (&)	x							
ビットごとの AND 代入演算子 (&=)	x							
ビットごとの左シフト演算子 (<<)	x							
ビットごとの NOT 演算子 (~)	x							
ビットごとの OR 演算子 (|)	x							
ビットごとの OR 代入演算子 (|=)	x							
ビットごとの右シフト演算子 (>>)	x							
ビットごとの XOR 演算子 (^)	x							
ビットごとの XOR 代入演算子 (^=)	x							
blink メソッド	x							

bold メソッド	x							
boolean 型							x	
Boolean オブジェクト		x						
break ステートメント	x							
byte 型							x	
call メソッド						x		
callee プロパティ						x		
caller プロパティ		x						
catch ステートメント					x			
@cc_on ステートメント			x					
ceil メソッド	x							
char 型							x	
charAt メソッド	x							
charCodeAt メソッド						x		
class ステートメント							x	
コンマ演算子 (,)	x							
// (単一行のコメント ステートメント)	x							
/*..*/ (複数行のコメント ステートメント)	x							
比較演算子	x							
compile メソッド			x					
concat メソッド (Array)			x					
concat メソッド (String)			x					
条件付きコンパイル			x					
条件付きコンパイル変数			x					
条件 (三項) 演算子 (?:)	x							
const ステートメント							x	
constructor プロパティ		x						

continue ステートメント	x							
cos メソッド	x							
型変換			x					
Date オブジェクト	x							
@debug ディレクティブ							x	
debugger ステートメント			x					
decimal 型							x	
decodeURI メソッド						x		
decodeURIComponent メソッド						x		
デクリメント演算子 (--)	x							
delete 演算子			x					
description プロパティ					x			
dimensions メソッド			x					
除算演算子 (/)	x							
除算代入演算子 (/=)	x							
do...while ステートメント			x					
double 型							x	
E プロパティ	x							
encodeURI メソッド						x		
encodeURIComponent メソッド						x		
enum ステートメント							x	
Enumerator オブジェクト			x					
等値演算子 (==)	x							
Error オブジェクト					x			
escape メソッド	x							
eval メソッド	x							
exec メソッド			x					

exp メソッド	x							
expando 修飾子							x	
false リテラル	x							
final 修飾子							x	
fixed メソッド	x							
float 型							x	
floor メソッド	x							
fontcolor メソッド	x							
fontsize メソッド	x							
for ステートメント	x							
for...in ステートメント					x			
fromCharCode メソッド			x					
function get ステートメント							x	
Function オブジェクト		x						
function set ステートメント							x	
function ステートメント	x							
getDate メソッド	x							
getDay メソッド	x							
getFullYear メソッド			x					
getHours メソッド	x							
getItem メソッド			x					
getMilliseconds メソッド			x					
getMinutes メソッド	x							
getMonth メソッド	x							
GetObject 関数			x					
getSeconds メソッド	x							
getTime メソッド	x							

getTimezoneOffset メソッド	x							
getUTCDate メソッド			x					
getUTCDay メソッド			x					
getUTCFullYear メソッド			x					
getUTCHours メソッド			x					
getUTCMilliseconds メソッド			x					
getUTCMinutes メソッド			x					
getUTCMonth メソッド			x					
getUTCSeconds メソッド			x					
getVarDate メソッド			x					
getYear メソッド	x							
Global オブジェクト			x					
global プロパティ						x		
大なり演算子 (>)	x							
以上演算子 (>=)	x							
hasOwnProperty メソッド						x		
hide 修飾子							x	
@if...@elif...@else...@end ステートメント			x					
if...else ステートメント	x							
ignoreCase プロパティ						x		
import ステートメント							x	
in 演算子	x							
インクリメント演算子 (++)	x							
index プロパティ			x					
indexOf メソッド	x							
非等値演算子 (!=)	x							

Infinity プロパティ			x					
input プロパティ ($_)			x					
instanceof 演算子					x			
int 型							x	
interface ステートメント							x	
internal 修飾子							x	
isFinite メソッド			x					
isNaN メソッド	x							
isPrototypeOf メソッド						x		
italics メソッド	x							
item メソッド			x					
JScript のデータ型							x	
join メソッド		x						
ラベル付きステートメント			x					
lastIndex プロパティ			x					
lastIndexOf メソッド	x							
lastMatch プロパティ ($&)						x		
lastParen プロパティ ($+)						x		
lbound メソッド			x					
leftContext プロパティ ($`)						x		
左シフト代入演算子 (<<=)	x							
length プロパティ (arguments)						x		
length プロパティ (Array)		x						
length プロパティ (Function)		x						
length プロパティ (String)	x							
小なり演算子 (<)	x							
以下演算子 (<=)	x							

link メソッド	x							
LN2 プロパティ	x							
LN10 プロパティ	x							
localeCompare メソッド						x		
log メソッド	x							
LOG2E プロパティ	x							
LOG10E プロパティ	x							
論理 AND 演算子 (&&)	x							
論理 NOT 演算子 (!)	x							
論理 OR 演算子 (||)	x							
long 型							x	
match メソッド			x					
Math オブジェクト	x							
max メソッド	x							
MAX_VALUE プロパティ		x						
message プロパティ						x		
min メソッド	x							
MIN_VALUE プロパティ		x						
剰余演算子 (%)	x							
剰余代入演算子 (%=)	x							
moveFirst メソッド			x					
moveNext メソッド			x					
multiline プロパティ						x		
乗算演算子 (*)	x							
乗算代入演算子 (*=)	x							
name プロパティ						x		
NaN プロパティ (Global)			x					

NaN プロパティ (Number)		x						
NEGATIVE_INFINITY プロパティ		x						
new 演算子	x							
比較演算子	x							
null リテラル	x							
Number 型							x	
Number オブジェクト		x						
number プロパティ					x			
Object オブジェクト			x					
演算子の優先順位	x							
override 修飾子							x	
package ステートメント							x	
parse メソッド	x							
parseFloat メソッド	x							
parseInt メソッド	x							
PI プロパティ	x							
pop メソッド						x		
@position ディレクティブ							x	
POSITIVE_INFINITY プロパティ		x						
pow メソッド	x							
print ステートメント							x	
private 修飾子							x	
propertyIsEnumerable プロパティ						x		
protected 修飾子							x	
prototype プロパティ		x						
public 修飾子							x	
push メソッド						x		

random メソッド	x							
RegExp オブジェクト			x					
Regular Expression オブジェクト			x					
正規表現の構文			x					
replace メソッド	x							
return ステートメント	x							
reverse メソッド		x						
rightContext プロパティ ($')						x		
右シフト代入演算子 (>>=)	x							
round メソッド	x							
sbyte 型							x	
ScriptEngine 関数		x						
ScriptEngineBuildVersion 関数		x						
ScriptEngineMajorVersion 関数		x						
ScriptEngineMinorVersion 関数		x						
search メソッド			x					
@set ステートメント			x					
setDate メソッド	x							
setFullYear メソッド			x					
setHours メソッド	x							
setMilliseconds メソッド			x					
setMinutes メソッド	x							
setMonth メソッド	x							
setSeconds メソッド	x							
setTime メソッド	x							
setUTCDate メソッド			x					

setUTCFullYear メソッド			x					
setUTCHours メソッド			x					
setUTCMilliseconds メソッド			x					
setUTCMinutes メソッド			x					
setUTCMonth メソッド			x					
setUTCSeconds メソッド			x					
setYear メソッド	x							
shift メソッド						x		
short 型							x	
sin メソッド	x							
slice メソッド (Array)			x					
slice メソッド (String)			x					
small メソッド	x							
sort メソッド		x						
source プロパティ			x					
splice メソッド						x		
split メソッド			x					
sqrt メソッド	x							
SQRT1_2 プロパティ	x							
SQRT2 プロパティ	x							
static 修飾子							x	
static ステートメント							x	
厳密等価演算子	x							
strike メソッド	x							
String 型							x	
String オブジェクト	x							
sub メソッド	x							

substr メソッド			x					
substring メソッド	x							
減算演算子 (-)	x							
減算代入演算子 (-=)	x							
sup メソッド	x							
super ステートメント							x	
switch ステートメント			x					
tan メソッド	x							
test メソッド			x					
this ステートメント	x							
throw ステートメント					x			
toArray メソッド			x					
toDateString メソッド						x		
toExponential メソッド						x		
toFixed メソッド						x		
toGMTString メソッド	x							
toLocaleDateString メソッド						x		
toLocaleLowerCase メソッド						x		
toLocaleString メソッド	x							
toLocaleTimeString メソッド						x		
toLocaleUpperCase メソッド						x		
toLowerCase メソッド	x							
toPrecision メソッド						x		
toString メソッド		x						
toTimeString メソッド						x		
toUpperCase メソッド	x							
toUTCString メソッド			x					

true リテラル	x							
try...catch...finally ステートメント					x			
型の注釈							x	
型変換							x	
typeof 演算子	x							
ubound メソッド			x					
uint 型							x	
ulong 型							x	
減算演算子 (-)	x							
undefined プロパティ						x		
unescape メソッド	x							
unshift メソッド						x		
符号なし右シフト演算子 (>>>)	x							
符号なし右シフト代入演算子 (>>>=)	x							
ushort 型							x	
UTC メソッド	x							
valueOf メソッド		x						
var ステートメント	x							
VBArray オブジェクト			x					
void 演算子		x						
while ステートメント	x							
with ステートメント	x							

参照

概念

JScript 8.0 の新機能

その他の技術情報

JScript リファレンス

JScript 言語の紹介

他の多くのプログラミング言語と同じように、Microsoft JScript のスクリプトまたはプログラムはテキスト形式で記述します。一般的に、スクリプトまたはプログラムは多数のステートメントとコメントで構成されます。ステートメント内では、変数、式、文字列や数字などのリテラル データを使用できます。

このセクションの内容

JScript の配列

JScript での配列の種類と使用方法を説明します。

JScript の代入と等価比較

JScript が変数、配列要素、およびプロパティ要素に値を代入するしくみ、および JScript で使用される等値構文について説明します。

JScript のコメント

正しい JScript 構文を使用して、コードにコメントを入力する方法を説明します。

JScript の式

キーワード、演算子、変数、およびリテラルを組み合わせて、新しい値を作成する方法を説明します。

JScript の識別子

JScript で識別子に有効な名前を付ける方法を説明します。

JScript のステートメント

JScript の命令の基本単位である、宣言ステートメントと実行ステートメントの概要について説明します。

JScript のデータ型

JScript のプリミティブ データ型、参照データ型、および .NET Framework データ型の使用方法について説明するトピックへのリンクを示します。

JScript の変数と定数

変数と定数の宣言方法と、これらを使用してオブジェクトを参照する方法を説明するトピックへのリンクを示します。

JScript オブジェクト

オブジェクトの概要について簡単に説明し、JScript でオブジェクトを作成および使用する方法を説明するトピックへのリンクを示します。

JScript の修飾子

可視性の修飾子、継承の修飾子、バージョン セーフ修飾子、expando 修飾子、および静的修飾子の使用方法を説明します。

JScript の演算子

算術演算子、論理演算子、ビット処理演算子、代入演算子、およびその他の演算子の一覧を示し、これらの演算子の効果的な使用方法について説明する情報へのリンクを示します。

JScript の関数

関数の概念を説明し、関数を使用および作成する方法を説明するトピックへのリンクを示します。

JScript における型の強制変換

JScript コンパイラが異なるデータ型の値をどのように処理するかを説明します。

データのコピー、受け渡し、および比較

JScript が配列、関数、オブジェクトなどを処理するときの、データのコピー、引き渡し、比較のそれぞれのしくみを説明します。

JScript の条件構造

JScript がプログラム フローを処理する方法について説明し、プログラムの実行フローの制御方法について説明する情報へのリンクを示します。

JScript の予約語

予約語の概念を説明し、JScript の予約語の一覧を示します。

JScript のセキュリティに関する考慮事項

JScript コードで一般的なセキュリティの問題を回避する方法を説明します。

関連するセクション

言語リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

.NET Framework クラス ライブラリ リファレンス

.NET Framework クラス ライブラリのクラス、およびクラス ライブラリ ドキュメントの使用方法を説明するトピックへのリンクを示します。

JScript の配列

配列は、関連するデータを 1 つの変数にグループ化します。インデックスまたは添字と呼ばれる一意の文字と、共有される変数の名前を組み合わせて、配列の各要素を識別します。配列を使用すると、インデックス番号を使用してループ内で任意の数の要素を効率的に使用できるため、多くの場合はコードが短く簡単になります。

JScript には 2 種類の配列が用意されています。1 つは JScript 配列オブジェクトであり、もう 1 つは型指定された配列です。JScript 配列オブジェクトは疎配列で、要素を動的に追加および削除できます。また、要素には任意の型を指定できます。型指定された配列は密な配列であり、サイズが固定されます。また、要素はすべて配列の基本型と同じ型にする必要があります。

このセクションの内容

関連するセクション

配列の概要

配列は、複数のデータを 1 つの変数にまとめたものです。1 つのインデックス番号 (1 次元配列の場合) または複数のインデックス番号 (配列の配列、または多次元配列の場合) を使って、配列内のデータを参照します。配列識別子の後に角かっこ (**[]**) で囲んだインデックス番号を指定すると、配列の各要素を参照できます。配列全体を参照するには、配列の識別子だけを使用します。データを配列にまとめることで、データ管理が簡単になります。たとえば、配列を使用することで、メソッドは 1 つのパラメータだけを使用して関数に名前のリストを渡すことができます。

JScript の配列には、JScript 配列と型指定された配列の 2 種類があります。これらの配列は似ていますが、いくつかの違いがあります。JScript 配列と型指定された配列は相互運用できます。したがって、JScript の **Array** オブジェクトは、型指定された配列のメソッドとプロパティを呼び出すことができます。また、型指定された配列は、**Array** オブジェクトの多くのメソッドとプロパティを呼び出すことができます。さらに、型指定された配列を受け取る関数は **Array** オブジェクトを受け取ることができ、逆の場合も同様です。詳細については、「Array オブジェクト」を参照してください。

型指定された配列

型指定された配列 (ネイティブ配列) は、C や C++ などの言語で使用される配列に似ています。型指定された配列は、配列の型宣言で指定された型のデータだけを格納することによって、タイプ セーフを実現します。

メモ :
Object 型の型指定された配列を定義すると、任意の型のデータを格納できます。

型指定された配列の要素数は、スクリプトが配列を作成または初期化するときに設定されます。要素の数を変更するには、配列を作成し直す必要があります。型指定された配列に *n* 個の要素がある場合、要素には 0 ～ *n*-1 の番号が付けられます。この範囲外の要素にアクセスしようとすると、エラーが発生します。また、型指定された配列は "密" な配列です。つまり、範囲内のインデックスは、すべて要素を参照しています。

スクリプトは、宣言済みの型指定された配列を変数または定数に代入できます。また、関数、演算子、またはステートメントに配列を渡すこともできます。変数 (または定数) に代入する場合、変数のデータ型が配列のデータ型に一致し、複数の配列の次元が一致するようにします。

型指定された配列は、.NET Framework の **System.Array** オブジェクトのインスタンスです。**System.Array** オブジェクトの静的メンバにアクセスする場合、または **System.Array** オブジェクトを明示的に作成する場合は、完全修飾名 **System.Array** が必要です。この構文により、組み込みの **JScript** オブジェクトの **Array** と区別されます。

JScript の配列

JScript の **Array** オブジェクトは型指定された配列よりも柔軟で、ジェネリック スタックや項目のリストが必要な場合、またはパフォーマンスが重要でない場合に便利です。ただし、型指定された配列はタイプ セーフでパフォーマンスに優れ、他の言語との相互運用性も高いため、開発者は、通常、JScript 配列ではなく型指定された配列を使用します。

JScript 配列には任意の型のデータを格納できます。これにより、型の衝突を気にせずに、配列を使用するスクリプトをすばやく簡単に作成できます。JScript 配列では JScript で提供される厳密な型チェックが省略されるため、JScript 配列を使用する場合は注意が必要です。

JScript 配列の要素は、動的に追加および削除できます。配列要素を追加するには、要素に値を代入します。**delete** 演算子を使用すると要素を削除できます。

JScript 配列は "疎" 配列です。配列に 0、1、および 2 の 3 つの要素がある場合、要素 3 ～ 49 がなくても要素 50 は存在できます。各 JScript 配列には **length** プロパティがあり、要素が追加されると自動的に更新されます。上の例で要素 50 を追加すると、length 変数の値は 4 ではなく 51 になります。

JScript の **Array** オブジェクトと JScript の **Object** は、ほとんど同じです。主な違いは、JScript の **Object** には (既定では) 自動的に更新される length プロパティがない点と、**Array** の持つプロパティとメソッドがない点の 2 つです。詳細については、「JScript の Array オブジェクト」を参照してください。

参照

関連項目

Array オブジェクト

概念

配列データ

JScript の Array オブジェクト

その他の技術情報

JScript の配列

配列の宣言

JScript の他のデータと同様に、配列も変数に格納できます。型の注釈を指定すると、変数が配列オブジェクトまたは型指定された配列を含む必要があることを指定できますが、初期配列は提供されません。変数に配列を格納するには、配列を "宣言" して、これを変数に代入する必要があります。

JScript 配列オブジェクトを宣言すると、新しい **Array** オブジェクトが作成されます。また、型指定された配列を宣言すると、配列のすべての要素を格納できるだけのメモリが予約されます。どちらの配列も、**new** 演算子を使用して明示的に新しい配列を作成するか、または配列リテラルを使用することによって宣言できます。

new 演算子による配列の宣言

新しい JScript **Array** オブジェクトを宣言するには、**new** 演算子と **Array** コンストラクタを使用します。JScript 配列には動的にメンバを追加できるため、配列の初期サイズを指定する必要はありません。次の例では、`a1` に長さ 0 の配列が代入されます。

```
var a1 = new Array();
```

Array コンストラクタを使用して作成した配列に、長さの初期値を割り当てるには、配列のコンストラクタに整数値を渡します。配列の長さは 0 か正の整数である必要があります。次のコードは、`a2` に長さが 10 の配列を代入しています。

```
var a2 = new Array(10);
```

Array コンストラクタに複数のパラメータを渡したり、数値以外のパラメータを 1 つ渡した場合、作成される配列の要素にすべてのパラメータが含まれます。たとえば、次のコードで作成される配列では、要素 0 は数値の 10、要素 1 は文字列 "Hello" になり、要素 2 には現在の日付が含まれます。

```
var a3 = new Array(10, "Hello", Date());
```

new 演算子を使って、型指定された配列を宣言することもできます。型指定された配列の場合、要素が動的に追加されることはないため、宣言で配列のサイズを指定する必要があります。型指定された配列のコンストラクタでは、かっこではなく、角かっこを使って配列サイズを囲みます。たとえば、次のコードでは 5 つの整数から成る配列を宣言します。

```
var i1 : int[] = new int[5];
```

new 演算子を使って、多次元配列を宣言することもできます。次の例では、3 × 4 × 5 の整数の配列を宣言します。

```
var i2 : int[,,] = new int[3,4,5];
```

配列の配列を宣言する場合、基本配列はサブ配列を宣言する前に宣言しておく必要があります。これらは同時に宣言できません。これにより、サブ配列のサイズを柔軟に決定できます。次の例では、1 番目のサブ配列は長さが 1、2 番目のサブ配列は長さが 2、のようになります。

```
// First, declare a typed array of type int[], and initialize it.
var i3 : int[][] = new (int[])[4];
// Second, initialize the subarrays.
for(var i=0; i<4; i++)
   i3[i] = new int[i+1];
```

配列リテラルによる配列の宣言

配列の宣言と初期化を同時に実行する方法として、配列リテラルを使用する方法があります。配列リテラルは JScript **Array** を表します。しかし、JScript 配列は型指定された配列と相互運用されるため、型指定された配列もリテラルを使用して初期化できます。詳細については、「配列データ」を参照してください。

配列リテラルを使用すると、1 次元配列を簡単に初期化できます。型指定された配列に配列リテラルを代入する場合、コンパイラは配列リテラルから正しい型にデータを変換しようとします。次の例では、リテラルの配列を JScript 配列と型指定された配列に代入します。

```
var al1 : Array = [1,2,"3"];
var il1 : int[] = [1,2,"3"];
```

配列リテラルは、配列の配列も初期化できます。次の例では、JScript 配列および型指定された配列の両方で、整数型の 2 つの配列から成る配列を初期化します。

```
var al1 : Array = [[1,2,3],[4,5,6]];
var il1 : int[][] = [[1,2,3],[4,5,6]];
```

配列リテラルでは、多次元の型指定された配列は初期化できません。

参照

関連項目

new 演算子

概念

配列データ

JScript の Array オブジェクト

その他の技術情報

JScript の配列

配列の使用方法

JScript では、さまざまな種類の配列を使用できます。ここでは、いくつかの配列の使用方法と、特定のアプリケーションに対して適切な配列を選択する方法について説明します。

1 次元配列

次の例は、`addressBook` 配列の先頭の要素と最後の要素にアクセスする方法を示します。 `addressBook` の定義と値の代入は、スクリプトの別の部分で実行されていると仮定します。JScript では配列のインデックスが 0 から始まるため、配列の先頭の要素は 0 で、最後の要素は配列の長さから 1 を引いた値になります。

```
var firstAddress = addressBook[0];
var lastAddress = addressBook[addressBook.length-1];
```

配列の配列と多次元配列

複数のインデックスで参照されるデータは、配列の配列または多次元配列に格納できます。どちらの配列にも独自の機能があります。

各サブ配列の長さが異なるアプリケーションでは、配列の配列が便利です。サブ配列は簡単に再編成できるため、配列要素の並べ替えに役立ちます。一般的な使用の例にはカレンダーがあります。`Year` 配列には 12 の `Month` 配列が含まれ、各 `Month` 配列には各月の日数のデータが格納されます。

各次元のサイズが宣言時にわかっているアプリケーションでは、多次元配列が便利です。多次元配列は、速度とメモリ使用の面で配列の配列よりも効率的です。多次元配列は、型指定された配列であることが必要です。一般的な例には、配列サイズが最初から固定されている、数値計算で使用される行列があります。

JScript 配列要素を使ったループ

JScript 配列は疎配列であるため、最初の要素から最後の要素までの間に、未定義の要素が多数含まれている可能性があります。したがって、**for** ループを使用して配列にアクセスする場合は、各要素が **undefined** かどうかを確認する必要があります。

JScript の **for...in** ループを使用すると、JScript 配列の定義された配列だけに簡単にアクセスできます。次の例では、JScript の疎配列を定義し、**for** と **for...in** ループを使用して要素を表示します。

```
var a : Array = new Array;
a[5] = "first element";
a[100] = "middle element";
a[100000] = "last element";
print("Using a for loop. This is very inefficient.")
for(var i = 0; i<a.length; i++)
   if(a[i]!=undefined)
      print("a[" + i + "] = " + a[i]);
print("Using a for...in loop. This is much more efficient.");
for(var i in  a)
   print("a[" + i + "] = " + a[i]);
```

このプログラムの出力は次のようになります。

```
Using a for loop. This is very inefficient.
a[5] = first element
a[100] = middle element
a[100000] = last element
Using a for...in loop. This is much more efficient.
a[5] = first element
a[100] = middle element
a[100000] = last element
```

参照

関連項目

for...in ステートメント

概念

配列の配列

多次元配列 (Jscript)

その他の技術情報

JScript の配列

配列の配列

作成した配列に他の配列を取り込むことができます。基本配列は、JScript 配列でも型指定された配列でもかまいません。型指定された配列には不適切な型のデータを格納できませんが、JScript 配列の場合は格納するデータをより柔軟に選択できます。

各サブ配列の長さが異なるアプリケーションでは、配列の配列が便利です。各サブ配列の長さが同じである場合は、多次元配列の方が便利です。詳細については、「多次元配列」を参照してください。

配列の配列 (型指定された配列の場合)

次の例では、文字列配列の配列にペットの名前を格納しています。各サブ配列の要素数は他のサブ配列の要素数とは別個に設定されているため (猫の名前の数と犬の名前の数は違っている場合があります)、多次元配列の代わりに配列の配列を使用しています。

```
// Create two arrays, one for cats and one for dogs.
// The first element of each array identifies the species of pet.
var cats : String[] = ["Cat","Beansprout", "Pumpkin", "Max"];
var dogs : String[] = ["Dog","Oly","Sib"];

// Create a typed array of String arrays, and initialze it.
var pets : String[][] = [cats, dogs];

// Loop over all the types of pets.
for(var i=0; i<pets.length; i++)
   // Loop over the pet names, but skip the first element of the list.
   // The first element identifies the species.
   for(var j=1; j<pets[i].length; j++)
      print(pets[i][0]+": "+pets[i][j]);
```

このプログラムの出力は次のようになります。

```
Cat: Beansprout
Cat: Pumpkin
Cat: Max
Dog: Oly
Dog: Sib
```

Object 型の型指定された配列を使用して配列を格納することもできます。

配列の配列 (JScript 配列の場合)

JScript 配列を基本配列として使用すると、格納されるサブ配列の型を柔軟に指定できます。たとえば、次のコードでは、文字列と整数を含む JScript 配列を格納する JScript 配列を作成します。

```
// Declare and initialize the JScript array of arrays.
var timecard : Array;
timecard = [ ["Monday", 8],
             ["Tuesday", 8],
             ["Wednesday", 7],
             ["Thursday", 9],
             ["Friday", 8] ];
// Display the contents of timecard.
for(var i=0; i<timecard.length; i++)
   print("Worked " + timecard[i][1] + " hours on " + timecard[i][0] + ".");
```

この例の出力は次のようになります。

```
Worked 8 hours on Monday.
Worked 8 hours on Tuesday.
Worked 7 hours on Wednesday.
Worked 9 hours on Thursday.
Worked 8 hours on Friday.
```

参照

概念

配列データ

多次元配列 (Jscript)

その他の技術情報

JScript の配列

多次元配列 (Jscript)

JScript では、多次元の型指定された配列を作成できます。多次元配列では、複数のインデックスを使用してデータにアクセスします。スクリプトで配列を宣言するときに、各インデックスの範囲を設定します。多次元配列は配列の配列に似ていますが、配列の配列では各サブ配列の長さが異なる場合もあります。詳細については、「配列の配列」を参照してください。

説明

データ型名の後に角かっこ (**[]**) を指定すると、1 次元配列のデータ型が定義されます。同様の手順で、かっこの間にコンマを指定して多次元配列のデータ型を指定します。配列の次元は、コンマの数に 1 を加えた数に等しくなります。次の例では、1 次元配列と多次元配列の定義の違いを示します。

```
// Define a one-dimensional array of integers. No commas are used.
var oneDim : int[];
// Define a three-dimensional array of integers.
// Two commas are used to produce a three dimensional array.
var threeDim : int[,,];
```

次の例では、文字型の 2 次元配列を使用して、三目並べの升目を格納しています。

```
// Declare a variable to store two-dimensional game board.
var gameboard : char[,];
// Create a three-by-three array.
gameboard = new char[3,3];
// Initialize the board.
for(var i=0; i<3; i++)
   for(var j=0; j<3; j++)
      gameboard[i,j] = " ";
// Simulate a game. 'X' goes first.
gameboard[1,1] = "X"; // center
gameboard[0,0] = "O"; // upper-left
gameboard[1,0] = "X"; // center-left
gameboard[2,2] = "O"; // lower-right
gameboard[1,2] = "X"; // center-right, 'X" wins!
// Display the board.
var str : String;
for(var i=0; i<3; i++) {
   str = "";
   for(var j=0; j<3; j++) {
      if(j!=0) str += "|";
      str += gameboard[i,j];
   }
   if(i!=0)
      print("-+-+-");
   print(str);
}
```

このプログラムの出力は次のようになります。

```
O| |
-+-+-
X|X|X
-+-+-
 | |O
```

Object 型の多次元の型指定された配列を使用すると、任意の型のデータを格納できます。

参照

概念

配列データ

配列の配列

その他の技術情報

JScript の代入と等価比較

JScript では、代入演算子を使用して変数に値を代入します。等値演算子は 2 つの値を比較します。

代入

多くのプログラミング言語と同様に、JScript は等号 (=) を使用して変数に値を代入します。等号は代入演算子です。= 演算子の左のオペランドは "左辺値" であることが必要です。つまり、変数、配列要素、またはオブジェクト プロパティを指定する必要があります。

= 演算子の右側のオペランドは "右辺値" です。右辺値には、式の結果を示す値を含む、あらゆる型の任意の値を指定できます。JScript の代入ステートメントの例を次に示します。

```
anInteger = 3;
```

JScript は、このステートメントを次のように解釈します。

"値 3 を変数 `anInteger` に代入する。"

または

"`anInteger` の値は 3 である。"

型の注釈でステートメントの変数を特定のデータ型にバインドしていない場合、代入は常に成功します。バインドしている場合、コンパイラは左辺値を右辺値のデータ型に変換しようとします。変換が成功しない場合、コンパイラはエラーを生成します。特定の値でだけ変換が成功する場合、コンパイラはコードの実行時に変換に失敗する可能性があることを示す警告を生成します。

次の例では、変数 `i` に格納された整数値を変数 `x` に代入すると、整数値は double 値に変換されます。

```
var i : int = 29;
var x : double = i;
```

詳細については、「型変換」を参照してください。

等価比較

他の一部の言語と異なり、JScript では等号を比較演算子として使用しません。等号は、代入演算子としてだけ使用されます。2 つの値を比較する場合は、等値演算子 (==) または厳密等価演算子 (===) を使用します。

等値演算子は、プリミティブな文字列、数値、および Boolean を値渡しで比較します。必要に応じて型変換を行った後、2 つの変数が同じ値である場合、等値演算子は **true** を返します。オブジェクト (**Array**、**Function**、**String**、**Number**、**Boolean**、**Error**、**Date**、および **RegExp** オブジェクト) は、参照渡しで比較されます。2 つのオブジェクト変数の値が同じであっても、**true** が返されるのは同一のオブジェクトを参照している場合だけです。

厳密等価演算子は 2 つの式の値と型の両方を比較します。2 つの式が等値演算子で等価であり、両方のオペランドのデータ型が同じ場合にだけ **true** を返します。

メモ :
厳密等価演算子は、数値データ型を区別しません。代入演算子、等値演算子、および厳密等価演算子の違いを理解してください。

スクリプト内で行った比較の結果は、常に Boolean 値となります。JScript のコード例を次に示します。

```
y = (x == 2000);
```

変数 `x` の値が数値 2000 と等しいかどうかが評価されます。変数の値が 2000 である場合、比較の結果は Boolean 値の **true** になり、これが変数 `y` に代入されます。`x` が 2000 に等しくない場合、比較の結果は Boolean 値の **false** になり、`y` に代入されます。

等値演算子は、型変換を行って、値が等しいかどうかをチェックします。次の JScript コードでは、リテラル文字列 "42" は、比較の前に数値の 42 に変換されます。結果は **true** になります。

```
42 == "42";
```

オブジェクトを比較するときの規則は異なります。等値演算子の動作は、オブジェクトの型に応じて決まります。オブジェクトが、等値演算子を使って定義されたクラスのインスタンスである場合、戻り値は等値演算子の実装に応じて決まります。等値演算子を提供するクラスは、JScript では定義できません。ただし、他の .NET Framework 言語では、このようなクラス定義も許可されます。

JScript の **Object** オブジェクトを基にしたオブジェクトや JScript クラスのインスタンスなど、等値演算子が定義されていないオブジェクトを比較する場合、それらのオブジェクトは同じオブジェクトを参照している場合にだけ等しくなります。つまり、同じデータを含む 2 つの異なるオブジェクトは、異なるものとして比較されます。この動作を次の例に示します。

```
// A primitive string.
var string1 = "Hello";
// Two distinct String objects with the same value.
var StringObject1 = new String(string1);
var StringObject2 = new String(string1);

// An object converts to a primitive when
// comparing an object and a primitive.
print(string1 == StringObject1);   // Prints true.

// Two distinct objects compare as different.
print(StringObject1 == StringObject2);   // Prints false.

// Use the toString() or valueOf() methods to compare object values.
print(StringObject1.valueOf() == StringObject2);   // Prints true.
```

等値演算子は、制御構造の条件ステートメントで特に有効です。等値演算子とそれを使用するステートメントを組み合わせてみます。JScript コードの例を示します。

```
if (x == 2000)
   z = z + 1;
else
   x = x + 1;
```

JScript の **if...else** ステートメントは、`x` の値が 2000 の場合に特定の動作 (この場合は、`z = z + 1`) を実行し、`x` の値が 2000 でない場合は別の動作 (`x = x + 1`) を実行します。詳細については、「JScript の条件構造」を参照してください。

厳密等価演算子 (===) は、数値データ型でだけ型変換を行います。したがって、整数値の 42 と倍精度浮動小数点値の 42 は等価と見なされますが、どちらも文字列の "42" とは等価になりません。この動作を次の JScript コードで示します。

```
var a : int = 42;
var b : double = 42.00;
var c : String = "42";
print(a===b); // Displays "true".
print(a===c); // Displays "false".
print(b===c); // Displays "false".
```

参照

概念

Boolean データ

型変換

その他の技術情報

JScript 言語の紹介

JScript の条件構造

JScript のコメント

JScript の単一行コメントは、2 つのスラッシュ (//) で始まります。

コード内のコメント

次に単一行コメントの例を示します。単一行コメントの次の行はコードです。

```
// This is a single-line comment.
aGoodIdea = "Comment your code for clarity.";
```

複数行コメントは、スラッシュとアスタリスク (/*) で始まり、この 2 つを逆に組み合わせた記号 (*/) で終わります。

```
/*
This is a multiline comment that explains the preceding code statement.
The statement assigns a value to the aGoodIdea variable. The value,
which is contained between the quote marks, is called a literal. A
literal explicitly and directly contains information; it does not
refer to the information indirectly. The quote marks are not part
of the literal.
*/
```

複数行コメントを他の複数行コメント内に単純に埋め込んだ場合、JScript では正しく解釈されません。埋め込んだ複数行コメントの終わりを示す */ が、複数行コメント全体の終わりとして解釈されてしまうからです。埋め込まれた複数行コメントの後のテキストは JScript コードとして解釈され、構文エラーを生成する可能性があります。

次の例では、内側の */ がコメントの終わりと解釈されるため、3 行目のテキストは JScript コードとして解釈されます。

```
/* This is the outer-most comment
/* And this is the inner-most comment */
...Unfortunately, JScript will try to treat all of this as code. */
```

コメントを記述する場合は、単一行コメントのブロックとして記述することをお勧めします。これにより、複数行コメントを使って、コードの大きなセグメントを後からコメント アウトできます。

```
// This is another multiline comment, written as a series of single-line comments.
// After the statement is executed, you can refer to the content of the aGoodIdea
// variable by using its name, as in the next statement, in which a string literal is
// appended to the aGoodIdea variable by concatenation to create a new variable.
var extendedIdea = aGoodIdea + " You never know when you'll have to figure out what it does
.";
```

また、条件付きコンパイルを使うと、コードの大きなセグメントを効果的にコメント アウトできます。

参照

その他の技術情報

JScript リファレンス
JScript 言語の紹介
条件付きコンパイル

JScript の式

JScript の式は、キーワード、演算子、変数、および値を表すリテラルの組み合わせです。式を使用すると、計算、データ操作、関数呼び出し、データの評価などを行うことができます。

式の使用

最も簡潔な式はリテラルです。次に、JScript のリテラルの式の例を示します。詳細については、「JScript のデータ」を参照してください。

```
3.9                              // numeric literal
"Hello!"                         // string literal
false                            // Boolean literal
null                             // literal null value
[1,2,3]                          // Array literal
var o = {x:1, y:2}               // Object literal
var f = function(x){return x*x;} // function literal
```

より複雑な式では、変数、関数呼び出し、およびその他の式を含めることができます。演算子を使って式を組み合わせて、複雑な式を作成できます。演算子の使用例を次に示します。

```
4 + 5        // additon
x += 1       // addition assignment
10 / 2       // division
a & b        // bitwise AND
```

JScript の複雑な式の例を次に示します。

```
radius = 10;
anExpression = 3 * (4 / 5) + 6;
aSecondExpression = Math.PI * radius * radius;
aThirdExpression = "Area is " + aSecondExpression + ".";
myArray = new Array("hello", Math.PI, 42);
myPi = myArray[1];
```

参照

その他の技術情報

JScript リファレンス
JScript 言語の紹介
JScript の演算子
JScript の変数と定数

JScript の識別子

JScript では、次のような目的で識別子を使用できます。

- 変数、定数、関数、クラス、インターフェイス、および列挙の名前
- ループのラベル (ほとんど使用されません)

識別子の使用

JScript では、大文字と小文字は区別されます。したがって、`myCounter` という変数と `MyCounter` という変数は別のものです。変数名の長さに制限はありません。有効な変数名を作成するための規則を次に示します。

- 先頭文字は、Unicode 文字 (大文字でも小文字でもかまいません) またはアンダースコア (_) 文字にする必要があります。先頭文字として数値を使用することはできません。
- 2 文字目以降の文字には、英字、数字、アンダースコア (_) を使用できます。
- 予約語は変数名に使用できません。

有効な変数名の例を次に示します。

```
_pagecount
Part9
Number_Items
```

次の変数名は無効です。

```
99Balloons // Cannot begin with a number.
Smith&Wesson // The ampersand (&) character is not a valid character for variable names.
```

識別子を選択する場合は、JScript の予約語や、組み込みオブジェクト名や関数名として既に使用されている語を使わないようにしてください。たとえば、**String** や **parseInt** などは使用しないでください。

参照

概念

JScript の予約語

その他の技術情報

JScript の変数と定数
JScript の関数
JScript オブジェクト

JScript のステートメント

JScript プログラムは、ステートメントのコレクションです。JScript ステートメントは、自然言語の文章に相当し、単一のタスクを実行する式を組み合わせて構成します。

ステートメントの使用

ステートメントには、1 つ以上の式、キーワード、または演算子 (記号) が含まれます。通常は 1 行に 1 つのステートメントが含まれますが、各ステートメントがセミコロンで区切られている場合は、2 つ以上のステートメントが同じ行に含まれることもあります。また、ほとんどのステートメントは複数行にまたがって使用できます。例外は、次のとおりです。

- 後置インクリメント演算子と後置デクリメント演算子は、引数と同じ行で使用する必要があります。たとえば、`x++` や `i--` などです。
- **continue** キーワードと **break** キーワードは、ラベルと同じ行で使用する必要があります。たとえば、`continue label1` や `break label2` などです。
- **return** キーワードと **throw** キーワードは、式と同じ行で使用する必要があります。たとえば、`return (x+y)` や `throw "Error 42"` などです。
- カスタム属性は、先頭に修飾子が付いていない限り、修飾する宣言と同じ行で使用する必要があります。たとえば、`myattribute class myClass` などです。

行の最後で明示的にステートメントを終了する必要はありませんが、ここで説明する JScript の例のほとんどでは、わかりやすくするために式を明示的に終了しています。ステートメントを明示的に終了するには、ステートメントの最後にセミコロン (;) を記述します。JScript では、セミコロン (;) は終了文字を表します。次に JScript ステートメントの例を 2 つ示します。

```
var aBird = "Robin"; // Assign the text "Robin" to the variable aBird.
var today = new Date(); // Assign today's date to the variable today.
```

複数の JScript ステートメントを中かっこで囲むと、かっこ内の一連のステートメントがまとめられてブロックとなります。ブロック内のステートメントは、通常、1 つのステートメントとして扱われます。つまり、単独のステートメントを記述できる場所であれば、ほとんどの場合、代わりにブロックを使用できます。ただし、**for** ループや **while** ループのヘッダーは例外です。次のコードは、**for** ループの使用例です。

```
var i : int = 0;
var x : double = 2;
var a = new Array(4);
for (i = 0; i < 4; i++) {
   x *= x;
   a[i] = x;
}
```

ブロック内の各ステートメントはセミコロンで終了しますが、ブロックそのものはセミコロンで終了しません。

通常、関数、条件処理、およびクラスではブロックを使用します。C++ やその他のほとんどの言語とは異なり、JScript ではブロックは新しいスコープとは見なされません。新しいスコープを作成するのは、関数、クラス、静的初期化子、および catch ブロックだけです。

次のコード例では、最初のステートメントが関数定義の開始行であり、関数は **if...else** の構文の 3 つのステートメントで構成されています。ブロックの後には、関数ブロックのかっこで囲まれていないステートメントがあります。したがって、最後のステートメントは関数定義には含まれません。

```
function FeetToMiles(feet, cnvType) {
   if (cnvType == "NM")
      return( (feet / 6080) + " nautical miles");
   else if (cnvType == "M")
      return( (feet / 5280) + " statute miles");
   else
      return ("Invalid unit of measure");
}
var mradius = FeetToMiles(52800, "M");
```

参照

関連項目

class ステートメント
function ステートメント
if...else ステートメント
static ステートメント

その他の技術情報

JScript リファレンス
JScript 言語の紹介

JScript のデータ型

JScript では、13 種類のプリミティブ データ型と、13 種類の参照データ型が用意されています。これらに加えて、新しいデータ型を宣言したり、共通言語仕様 (CLS: Common Language Specification) 準拠の .NET Framework データ型を使用したりできます。ここでは、組み込みデータ型の情報、これらの型を拡張する方法、独自のデータ型を定義する方法、データを入力する方法、およびデータ型を変換する方法を説明します。

このセクションの内容

JScript のデータ

配列、Boolean 値、数値、文字列、およびオブジェクト データを入力する方法を説明するトピックへのリンクを示します。

データ型の概要

JScript のプリミティブ型と参照型、および対応する .NET Framework のクラスを表に示します。

ユーザー定義データ型

class ステートメントを使って新しいデータ型を定義する方法を説明します。

型指定された配列

型指定された配列を定義、初期化、および使用する方法を説明します。

型変換

型変換の概念と、暗黙の型変換および明示的な型変換の違いを説明します。

関連するセクション

JScript オブジェクト

オブジェクトの概要について簡単に説明し、JScript でオブジェクトを作成および使用する方法を説明するトピックへのリンクを示します。

データ型

組み込みデータ型と、それらに関連するプロパティとメソッドを説明するトピックへのリンクを示します。

JScript のデータ

ほとんどの言語と同様に、JScript ではいくつかの基本的なデータを使用します。これらのデータには、数値データと文字列データがあります。文字列はテキストのブロックです。このデータを JScript プログラムに入力するにはいくつかの方法があります。多くの場合、データはリテラル式で入力されます。

このセクションの内容

配列データ

JScript の配列の概念と、スクリプトで配列リテラルを使って配列データを入力する方法を説明します。

Boolean データ

Boolean データの概念と、JScript コードで 2 つのリテラル値を使用する方法を説明します。

数値データ

整数データと浮動小数点データの違いと、スクリプトで数値データを入力する方法を説明します。

文字列データ

文字列データの概念、構文、およびエスケープ文字の使用方法を説明します。

オブジェクト データ

Jscript のオブジェクト データの概念、初期化、および使用方法を説明します。

関連するセクション

JScript のデータ型

JScript のプリミティブ データ型、参照データ型、および .NET Framework データ型の使用方法について説明するトピックへのリンクを示します。

データ型

組み込みデータ型と、それらに関連するプロパティとメソッドを説明するトピックへのリンクを示します。

配列データ

配列リテラルは、JScript の配列を初期化できます。配列リテラルは JScript の **Array** オブジェクトを表し、角かっこ ([]) で囲まれたコンマ区切りのリストで表されます。リストの各要素は、有効な JScript 式か空 (2 つの連続したコンマ) になります。配列リテラル リストの 1 番目の要素のインデックス番号は 0 です。リストの 2 番目以降の要素は、配列の 2 番目以降の要素に対応します。配列リテラル リストの要素が空の場合、JScript **Array** の対応する要素は初期化されず、JScript **Array** は疎な配列になります

配列データの使用

次の例では、変数 `arr` は 3 つの要素を持つ配列として初期化されます。

```
var arr = [1,2,3];
```

Array リテラル リストに空の要素を使用して、疎な配列を作成できます。たとえば、次の配列リテラルは要素 0 と要素 4 だけが定義された配列を表します。

```
var arr = [1,,,,5];
```

配列リテラルには、他の配列を含む、任意の型のデータを指定できます。次の配列の配列では、2 番目のサブ配列には文字列と数値データの両方があります。

```
var cats = [ ["Names", "Beansprout", "Pumpkin", "Max"], ["Ages", 6, 5, 4] ];
```

JScript の **Array** オブジェクトは型指定された配列と相互運用されるため、配列リテラルで型指定された配列も初期化できます。ただし、いくつか制限があります。配列リテラルのデータは、型指定された配列のデータ型に変換できる必要があります。配列リテラルは多次元の型指定された配列を初期化できませんが、型指定された配列の型指定された配列は初期化できます。配列リテラルで型指定された配列を初期化する場合は、2 段階の処理が行われます。まず、配列リテラルは型指定された配列に変換され、型指定された配列の初期化に使用されます。この変換では、配列リテラルの空の要素は **undefined** として解釈され、続いてリテラルの要素が型指定された配列の適切なデータ型に変換されます。次の例では、同一の配列リテラルを使用して、JScript 配列、整数型の配列、および double 型の配列を初期化しています。

```
var arr = [1,,3];
var arrI : int[] = [1,,3];
var arrD : double[] = [1,,3];
print(arr);   // Displays  1,,3.
print(arrI);  // Displays  1,0,3.
print(arrD);  // Displays  1,NaN,3.
```

配列リテラルの空の要素は **undefined** に割り当てられるため、整数型の配列では 0 で表され、double 型の配列では **NaN** で表されます。

参照

関連項目

Array オブジェクト

概念

JScript の式

型変換

その他の技術情報

JScript のデータ

データ型 (JScript)

JScript の配列

組み込みオブジェクト

Boolean データ

文字列型および数値型では、ほぼ無限とも言える種類の値を利用できるのに対し、JScript の **Boolean** 型で使用できる数値は 2 種類だけです。リテラルの **true** および **false** です。Boolean 値は、条件の有効性 (条件が **true** か **false** か) を表します。

Boolean 値の使用

制御構造では、リテラルの Boolean 値 (**true** または **false**) を条件ステートメントとして使用できます。たとえば、**while** ステートメントの条件に **true** を使用して無限ループを作成できます。

```
var s1 : String = "Sam W.";
var s2 : String = "";
while (true) {
   if(s2.Length<s1.Length)
      s2 = s2 + "*";
   else
      break;
}
print(s1);   // Prints Sam W.
print(s2);   // Prints ******
```

無限ループの終了条件をループの制御部分に移して、明示的に有限ループとすることもできます。ただし、無限ループを使用することで、ループを簡単に記述できる場合もあります。

if...else ステートメントに Boolean リテラルを使用すると、簡単にステートメントを挿入したり、プログラム内の複数のステートメントのいずれかを選択したりできます。この方法はプログラム開発では有効です。ただし、**if** ステートメントを使用せずに直接ステートメントを挿入したり、コメントを使用してステートメントを挿入しないようにしたりする方がより効果的です。

詳細については、「JScript の条件構造」を参照してください。

参照

関連項目

true リテラル
false リテラル
boolean 型 (JScript)
Boolean オブジェクト

概念

JScript の式

その他の技術情報

JScript のデータ
JScript の条件構造

数値データ

JScript の数値データに整数データと浮動小数点データのどちらを使用するかは、データを使用する状況によって決まります。整数データと浮動小数点データを表す方法もいくつかあります。

正の整数、負の整数、および 0 は整数型です。これらの数値は、基数 10 (10 進数)、基数 8 (8 進数)、および基数 16 (16 進数) で表現できます。JScript では、ほとんどの数値は 10 進数で表記されます。8 進数の整数であることを示すには、先頭に 0 を付けます。各桁には 0 ～ 7 の数字だけを使用します。先頭が 0 でも、8 や 9 が含まれている場合は 10 進数として処理されます。一般的には、8 進数の使用はお勧めしません。

16 進数 (hex) の数値であることを示すには、先頭に "0x" (x は大文字でも小文字でも可) を付けます。各桁には 0 ～ 9 の数字および A ～ F の英字 (大文字でも小文字でも可) を使用します。A ～ F の英字は、1 桁の数値として、10 進数の 10 ～ 15 に相当します。たとえば、0xF は 15、0x10 は 16 に相当します。

8 進数および 16 進数は負の値である場合もあります。小数部分はなく、指数表記は使用できません。

浮動小数点数は、整数に小数部分が付加されます。浮動小数点数は、整数型の表記に、小数点以下の数値を指定して表現します。また、指数表記も使用できます。10 の累乗の指数を表すには、大文字または小文字の e が使用されます。1 つの 0 から始まり小数点を含む数値は、10 進数の浮動小数点リテラルとして解釈され、8 進数リテラルには解釈されません。

また、JScript の浮動小数点数は、整数型では表現できない特殊な数値を表すことができます。これらの数値は、次のとおりです。

- **NaN** (非数)。数値演算が不適切なデータ (文字列や未定義値) で実行された場合に使用されます。
- **Infinity**。正の数値が JScript で表記するには大きすぎる場合に使用されます。
- -**Infinity** (負の Infinity)。負の数値が JScript で表現するには小さすぎる場合に使用されます。
- 正の 0 および負の 0。JScript では、状況に応じて正の 0 と負の 0 が区別されます。

JScript の数値の例を次に示します。数値が "0x" で始まり小数点がある場合は、エラーが発生します。

数値	説明	10 進数での表記
.0001、0.0001、1e-4、1.0e-4	この 4 つの浮動小数点数は、すべて同じ値です。	0.0001
3.45e2	浮動小数点数。	345
42	整数です。	42
0378	整数です。"0" で始まるため 8 進数のように見えますが、8 は 8 進数では使用できない数値であるため、10 進数として処理されます。この表記では、レベル 1 の警告が生成されます。	378
0377	8 進数です。上記の数値より 1 だけ小さい数値に見えますが、実際にはまったく異なる数値です。	255
0.0001, 00.0001	浮動小数点数です。"0" で始まりますが小数点があるため 8 進数ではありません。	0.0001
0Xff	16 進数の整数です。	255
0x37CF	16 進数の整数です。	14287
0x3e7	16 進数の整数です。e は累乗として処理されません。	999
0x3.45e2	エラーが発生します。16 進数は小数部を持つことができません。	N/A (コンパイラ エラー)

整数型の変数は、有限の数値範囲だけを表現できます。整数型に対して大きすぎる、または小さすぎる数値リテラルを代入しようとすると、コンパイル時に型の不一致エラーが発生します。詳細については、「データ型の概要」を参照してください。

リテラルのデータ型

ほとんどの場合、JScript が数値リテラルを解釈するときのデータ型は、あまり意識する必要はありません。ただし、数値が非常に大きい場合や正確な場合、データ型は重要になります。

JScript の整数リテラルは、サイズと使用方法に応じて、**int**、**long**、**ulong**、**decimal**、または **double** 型のデータを表します。**int** 型の範囲 (-2147483648 ～ 2147483647) にあるリテラルは、**int** 型として解釈されます。int 型の範囲外で **long** 型の範囲 (-9223372036854775808 ～ 9223372036854775807) にあるリテラルは、**long** 型として解釈されます。long 型の範囲外で **ulong** 型の範囲 (9223372036854775807 ～ 18446744073709551615) にあるリテラルは、**ulong** 型として解釈されます。その他の整数リテラルはすべて **double** 型として解釈されるため、精度の情報は失われます。例外的に、リテラルがすぐに **decimal** 型の変数または定数に格納される場合、または **decimal** を受け取る関数に渡される場合は、**decimal** として解釈されます。

JScript の浮動小数点リテラルは **double** 型に解釈されます。ただし、整数型と同様に、すぐに **decimal** として使用される場合は **decimal** として解釈されます。**decimal** 型では、**NaN**、正の **Infinity**、または負の **Infinity** を表現できません。

参照

関連項目

NaN プロパティ (Global)

Infinity プロパティ

概念

JScript の式

その他の技術情報

JScript のデータ

データ型 (JScript)

オブジェクト データ

オブジェクト リテラルは、JScript の **Object** オブジェクトを初期化できます。オブジェクト リテラルは、中かっこ ({}) で囲まれたコンマ区切りのリストで表します。リストの各要素は、プロパティの名前とその値をコロンで区切って指定します。値には、有効な JScript の式を指定できます。

オブジェクト データの使用

次の例では、変数 `obj` が `x` と `y` の 2 つのプロパティを持つオブジェクトに初期化されています。`x` と `y` のそれぞれの値は 1 と 2 です。

```
var obj = { x:1, y:2 };
```

オブジェクト リテラルは入れ子にできます。次の例では、識別子 `cylinder` は `height`、`radius`、および `sectionAreas` の 3 つのプロパティを持つオブジェクトを参照しています。`sectionAreas` プロパティは、`top`、`bottom`、および `side` という独自のプロパティを持ちます。

```
var r = 3;
var h = 2;
var cylinder = { height : h, radius : r,
                 sectionAreas : { top : 4*Math.PI*r*r,
                                  bottom : 4*Math.PI*r*r,
                                  side : 2*Math.PI*r*h } };
```

メモ :

オブジェクト リテラルを使用して、クラス ベースのオブジェクトのインスタンスを初期化することはできません。初期化するには、適切なコンストラクタ関数を使用する必要があります。詳細については、「クラス ベースのオブジェクト」を参照してください。

参照

関連項目

Object オブジェクト

概念

JScript の式

その他の技術情報

JScript のデータ

組み込みオブジェクト

文字列データ

文字列値は、0 個以上の Unicode 文字 (英字、数字、および区切り記号) をつなぎ合わせたものです。文字列型は JScript のテキストを表します。スクリプトでリテラル文字列を使用するには、単一引用符 (') または二重引用符 (") で囲みます。単一引用符で囲んだ文字列の内側に二重引用符のペアを使用したり、その逆の組み合わせで 2 組の引用符のペアを入れ子にして使用したりできます。文字列の例を次に示します。

文字列データの使用

```
"The earth is round."
'"Come here, Watson. I need you." said Alexander.'
"42"
"15th"
'c'
```

JScript にはエスケープ シーケンスが用意されており、直接入力できない文字を文字列として使用できます。エスケープ シーケンスは、円記号 (\) で始まります。この円記号は、次に続く文字が特殊文字であることを JScript のインタープリタに知らせるエスケープ文字です。

エスケープ シーケンス	説明
\b	バックスペース。
\f	フォーム フィード (あまり使用されません)。
\n	ライン フィード (改行)。
\r	キャリッジ リターン。ライン フィードと組み合わせて (\r\n)、出力の書式を指定します。
\t	水平タブ。
\v	垂直タブ。ECMAScript 標準には非準拠で、Microsoft Internet Explorer 6.0 とは互換性がありません。
\'	単一引用符 (')。
\"	二重引用符 (")。
\\	円記号 (\)。
n	8 進数の *n* で表される ASCII 文字。*n* の値は、0 ～ 377 (8 進数) にする必要があります。
\x*hh*	2 桁の 16 進数 *hh* で表される ASCII 文字。
\u*hhhh*	4 桁の 16 進数 *hhhh* で表される Unicode 文字。

この表にないエスケープ シーケンスは、単に円記号の後に続く文字を表します。たとえば、"\a" は "a" と解釈されます。

円記号自体は、エスケープ シーケンスの開始文字を表すため、スクリプトにそのまま文字として入力することはできません。円記号を文字として使うには、円記号を 2 つ続けて (\\) 入力する必要があります。

```
'The image path is C:\\webstuff\\mypage\\gifs\\garden.gif.'
```

単一引用符および二重引用符のエスケープ シーケンスを使用すると、リテラル文字列で引用符を使用できます。次の例は、埋め込まれた引用符を示しています。

```
'The caption reads, \"After the snow of \'97. Grandma\'s house is covered.\"'
```

JScript は、組み込みの **char** データ型を使用して 1 つの文字を表します。1 文字の文字列や単一のエスケープ シーケンスは、文字列自体が

char 型でない場合でも、**char** 型の変数に代入できます。

何も入っていない文字列 ("") は、空の文字列 (長さ 0 の文字列) です。

参照

関連項目

String 型 (JScript)

String オブジェクト

概念

JScript の式

その他の技術情報

JScript のデータ

データ型の概要

JScript には、プログラムで使用できる多くのデータ型が用意されています。これらの型は、値データ型と、JScript オブジェクトと呼ばれる参照データ型の 2 つのカテゴリに分類されます。JScript に型を追加するには、新しいデータ型を含む名前空間またはパッケージをインポートするか、新しいデータ型として使用できる新しいクラスを定義します。

データ型の詳細

次の表は、JScript でサポートされる値データ型を示しています。表の 2 列目は、Microsoft .NET Framework の対応するデータ型です。.NET Framework 型の変数または JScript の値型のどちらの変数を宣言しても同じ結果が得られます。記憶領域のサイズと範囲も型ごとに決まっています。3 列目は、1 つのインスタンスに必要な記憶領域を示しています。4 番目の列は、保持できる値の範囲を示します。

JScript の値型	.NET Framework 型	記憶領域のサイズ	範囲
boolean	Boolean	N/A	**true** または **false**。
char	Char	2 バイト	任意の Unicode 文字。
float (単精度浮動小数点数)	Single	4 バイト	約 -1038 ～ 1038。精度は 7 桁です。10-44 までの数値を表すことができます。
Number、double (倍精度浮動小数点数)	Double	8 バイト	約 -10308 ～ 10308。精度は 15 桁です。10-323 までの数値を表すことができます。
decimal	Decimal	12 バイト (整数部)	約 -1028 ～ 1028。精度は 28 桁です。10-28 までの数値を表すことができます。
byte (符号なし)	Byte	1 バイト	0 ～ 255。
ushort (符号なし短整数)	UInt16	2 バイト	0 ～ 65,535。
uint (符号なし整数)	UInt32	4 バイト	0 ～ 4,294,967,295。
ulong (符号なし長整数)	UInt64	8 バイト	0 ～ 約 1020。
sbyte (符号付き)	SByte	1 バイト	-128 ～ 127。
short (符号付き短整数)	Int16	2 バイト	-32,768 ～ 32,767。
int (符号付き整数)	Int32	4 バイト	-2,147,483,648 ～ 2,147,483,647。
long (符号付き長整数)	Int64	8 バイト	約 -1019 ～ 1019。
void	N/A	N/A	値を返さない関数の戻り値の型として使用されます。

次の表は、JScript に用意されている、型として使用できる参照データ型 (JScript オブジェクト) です。参照型には、定義済みの特定の記憶領域サイズがありません。

JScript の参照型	.NET Framework 型	参照先
ActiveXObject	同等の項目はありません。	オートメーション オブジェクト。
Array	Array および型指定された配列と相互運用。	任意の型の配列。

Boolean	**Boolean** と相互運用。	**true** または **false** の Boolean 値。
Date	DateTime と相互運用。	日付は JScript の **Date** オブジェクトを使って実装されます。範囲は、1970 年 1 月 1 日の前後およそ 285,616 年です。
Enumerator	同等の項目はありません。	コレクション内の項目の列挙。下位互換性を維持するために用意されています。
Error	同等の項目はありません。	**Error** オブジェクト。
Function	同等の項目はありません。	**Function** オブジェクト。
Number	**Double** と相互運用。	約 -10308 ～ 10308 の範囲の数値。精度は約 15 桁です。10-323 までの数値を表すことができます。
Object	Object と相互運用。	**Object** 参照。
RegExp	Regex と相互運用。	正規表現オブジェクト。
String 型 (可変長)	String	0 個 ～ 約 20 億個の Unicode 文字各文字は 16 ビット (2 バイト) です。
String オブジェクト (可変長)	**String** と相互運用。	0 個 ～ 約 20 億個の Unicode 文字各文字は 16 ビット (2 バイト) です。
VBArray	同等の項目はありません。	読み取り専用の Visual Basic 配列。下位互換性を維持するために用意されています。

参照

関連項目

import ステートメント
package ステートメント
class ステートメント

概念

ユーザー定義データ型
データのコピー、受け渡し、および比較

その他の技術情報

データ型 (JScript)
オブジェクト (JScript)
JScript オブジェクト

ユーザー定義データ型

JScript には用意されていないデータ型が必要になる場合があります。この場合、新しいクラスを定義するパッケージをインポートするか、**class** ステートメントを使用して独自のデータ型を作成できます。クラスは、JScript の定義済みのデータ型と同じ方法で、型の注釈や型指定された配列の作成に使用できます。

データ型の定義

次の例では、**class** ステートメントを使用して、新しいデータ型の `myIntVector` を定義しています。関数宣言で新しい型を使用して、関数のパラメータの型を示しています。また、変数に新しい型の注釈を指定しています。

```
// Define a class that stores a vector in the x-y plane.
class myIntVector {
   var x : int;
   var y : int;
   function myIntVector(xIn : int, yIn : int) {
      x = xIn;
      y = yIn;
   }
}

// Define a function to compute the magnitude of the vector.
// Passing the parameter as a user defined data type.
function magnitude(xy : myIntVector) : double {
   return( Math.sqrt( xy.x*xy.x + xy.y*xy.y ) );
}

// Declare a variable of the user defined data type.
var point : myIntVector = new myIntVector(3,4);
print(magnitude(point));
```

このコードの出力は次のようになります。

```
5
```

参照

関連項目

class ステートメント
package ステートメント

概念

型の注釈

その他の技術情報

データ型 (JScript)
JScript オブジェクト

型指定された配列

型指定された配列は、変数、定数、関数、およびパラメータに、組み込みデータ型のように注釈を付けることができるデータ型です。型指定された配列には基本データ型があり、配列の各要素はその基本型になります。基本型自身が配列型となることができ、配列の配列を構成できます。

型指定された配列の使用

データ型の後に角かっこを指定すると、1 次元の型指定された配列が定義されます。*n* 次元の配列を定義するには、基本データ型の後の角かっこの間に *n*-1 のコンマを指定します。

型指定された配列の変数には、最初は記憶領域が割り当てられず、初期値は **undefined** になります。配列変数を初期化するには、**new** 演算子、配列リテラル、配列コンストラクタ、または他の配列を使用します。初期化は、他の型の変数と同様、型指定された配列変数が宣言されたときか、後で実行できます。変数やパラメータの次元が、変数に代入された配列またはパラメータに渡された配列の次元 (または型) と一致しない場合は、型の不一致エラーが発生します。

配列コンストラクタを使用すると、指定したネイティブ型の配列を、指定した (固定の) サイズで作成できます。各引数は、正の整数に評価される式であることが必要です。各引数の値は、各次元の配列のサイズを決定します。引数の数は、配列の次元数を決定します。

次の例は、簡単な配列の宣言を示しています。

```
// Simple array of strings; initially empty. The variable 'names' itself
// will be null until something is assigned to it
var names : String[];

// Create an array of 50 objects; the variable 'things' won't be null,
// but each element of the array will be until they are assigned values.
var things : Object[] = new Object[50];
// Put the current date and time in element 42.
things[42] = new Date();

// An array of arrays of integers; initially it is null.
var matrix : int[][];
// Initialize the array of arrays.
matrix = new (int[])[5];
// Initialize each array in the array of arrays.
for(var i = 0; i<5; i++)
   matrix[i] = new int[5];
// Put some values into the matrix.
matrix[2][3] = 6;
matrix[2][4] = 7;

// A three-dimensional array
var multidim : double[,,] = new double[5,4,3];
// Put some values into the matrix.
multidim[1,3,0] = Math.PI*5.;
```

参照

関連項目

var ステートメント
new 演算子
function ステートメント

概念

型の注釈

その他の技術情報

データ型 (JScript)

型変換

値の型を変更する処理は型変換と呼ばれます。たとえば、文字列の "1234" は数値に変換できます。また、任意の型のデータを **String** 型に変換できます。成功しない型変換もあります。たとえば、**Date** オブジェクトは **ActiveXObject** オブジェクトに変換することはできません。

型変換は、"拡大変換" と "縮小変換" のいずれかです。拡大変換はオーバーフローせず、常に成功しますが、縮小変換は情報の損失を伴う可能性があり、変換に失敗する場合があります。

どちらの変換も、データ型の識別子を使用して明示的に行うか、データ型の識別子を使用せずに暗黙的に実行します。有効な明示的な変換は、情報の損失が生じる場合でも常に成功します。暗黙の型変換は、データが失われない場合にだけ成功します。失われる場合は失敗して、コンパイル エラーかランタイム エラーが生成されます。

変換による損失は、変換元のデータと変換先のデータ型に類似性がない場合に発生します。たとえば、文字列の "Fred" は数値には変換できません。この場合、型変換関数からは既定値が返されます。**Number** 型の場合、既定値は **NaN** です。**int** 型の場合、既定値は 0 です。

文字列から数値への変換など、変換に時間のかかる型もあります。プログラムでは、変換を使用しないほど効率的です。

暗黙の型変換

変数に値を代入する場合など、ほとんどの型変換は自動的に実行されます。変数のデータ型により、式の変換先のデータ型が決まります。

次の例は、暗黙の型変換において、**int** 値、**String** 値、および **double** 値のデータがどのように変換されるかを示しています。

```
var i : int;
var d : double;
var s : String;
i = 5;
s = i;  // Widening: the int value 5 coverted to the String "5".
d = i;  // Widening: the int value 5 coverted to the double 5.
s = d;  // Widening: the double value 5 coverted to the String "5".
i = d;  // Narrowing: the double value 5 coverted to the int 5.
i = s;  // Narrowing: the String value "5" coverted to the int 5.
d = s;  // Narrowing: the String value "5" coverted to the double 5.
```

このコードをコンパイルすると、縮小変換に失敗する可能性がある、または縮小変換は低速であることを示す、コンパイル時の警告が発生します。

変換によって情報の損失が生じる場合、暗黙の縮小変換は動作しません。たとえば、次の例は動作しません。

```
var i : int;
var f : float;
var s : String;
f = 3.14;
i = f;  // Run-time error. The number 3.14 cannot be represented with an int.
s = "apple";
i = s;  // Run-time error. The string "apple" cannot be converted to an int.
```

明示的な変換

式を明示的に特定のデータ型に変換するには、データ型の識別子の後に、変換する式をかっこで囲んで指定します。明示的な変換では、プログラマは暗黙の型変換を使用する場合よりも多くの入力が必要となりますが、より確実な結果が得られます。また、明示的な変換では損失を伴う変換も処理できます。

次の例は、明示的な型変換において、**int** 値、**String** 値、および **double** 値のデータがどのように変換されるかを示しています。

```
var i : int;
var d : double;
var s : String;
i = 5;
s = String(i);  // Widening: the int value 5 coverted to the String "5".
d = double(i);  // Widening: the int value 5 coverted to the double 5.
s = String(d);  // Widening: the double value 5 coverted to the String "5".
i = int(d);     // Narrowing: the double value 5 coverted to the int 5.
```

```
i = int(s);     // Narrowing: the String value "5" coverted to the int 5.
d = double(s);  // Narrowing: the String value "5" coverted to the double 5.
```

明示的な縮小変換は、変換によって情報の損失が生じる場合でも動作します。明示的な型変換を使用しても、互換性のないデータ型は変換できません。たとえば、**Date** データを **RegExp** データには変換できません。また、変換される実際的な値がない場合も変換はできません。たとえば、double 値の **NaN** を明示的に **decimal** に変換しようとした場合は、エラーがスローされます。**NaN** に対応する **decimal** 値がないためです。

次の例では、小数部を持つ数値を整数値に変換し、文字列を整数に変換しています。

```
var i : int;
var d : double;
var s : String;
d = 3.14;
i = int(d);
print(i);
s = "apple";
i = int(s);
print(i);
```

出力は次のようになります。

```
3
0
```

明示的な変換の動作は、変換元と変換先の両方のデータ型に依存します。

参照

関連項目

undefined プロパティ

概念

型の注釈

その他の技術情報

JScript のデータ型

JScript の変数と定数

プログラミング言語では、データは情報を表します。たとえば、次のリテラル文字列には質問が含まれています。

```
'How old am I?'
```

変数と定数は、データを格納します。スクリプトは、変数の名前と定数の名前を使って簡単にデータを参照できます。変数に格納されたデータは、プログラムの実行中に変化しますが、定数に格納されたデータは変化しません。変数を使用するスクリプトは、実際に変数が表すデータにアクセスします。次の例では、`NumberOfDaysLeft` という変数に、`EndDate` と `TodaysDate` の差を代入しています。

```
NumberOfDaysLeft = EndDate - TodaysDate;
```

変数を使用すると、スクリプト内で使用されている値を保存、取得、および操作できます。定数は変化しないデータを参照します。変数名には、変数の目的がわかるような、また、他のユーザーがスクリプトの機能を判断できるような、意味のある名前を使用してください。

このセクションの内容

JScript の変数と定数の型

変数の適切なデータ型を選択する方法、および適切なデータ型を選ぶことの利点を説明します。

JScript の変数と定数の宣言

型指定された、または型指定されていない変数と定数を宣言する方法、およびこれらを初期化する方法を説明します。

変数と定数のスコープ

JScript のグローバル スコープとローカル スコープの違い、およびローカル スコープがグローバル スコープを隠ぺいするしくみを説明します。

undefined 値

未定義の値の概念、変数やプロパティが定義されているかどうかを判断する方法、および変数やプロパティを未定義にする方法を説明します。

関連するセクション

JScript の識別子

JScript で識別子に有効な名前を付ける方法を説明します。

JScript のデータ型

JScript のプリミティブ データ型、参照データ型、および .NET Framework データ型の使用方法について説明するトピックへのリンクを示します。

JScript の代入と等価比較

JScript が変数、配列要素、およびプロパティ要素に値を代入するしくみ、および JScript で使用される等値構文について説明します。

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

JScript の変数と定数の型

JScript には多くのデータ型があり、コードを記述するときに役立ちます。データ型を効果的に使用することで、既定の JScript データ型を使用するプログラムよりも、読み込みと実行を高速に行うことができます。また、コンパイラは、型の誤用に関する有用なエラー メッセージや警告を提供できます。

説明

たとえば、常に 1,000,000 未満の整数が格納される変数に、8 バイトの **double** を使用しても意味はありません。このデータに対して最も効果的なデータ型は **int** です。**int** は 4 バイトの整数型で、–2,147,483,648 ～ 2,147,483,647 の範囲の値を格納します。

型の注釈を指定して宣言した変数または定数には、適切な型のデータだけが格納されるようになります。JScript では、他にも多くのデータ型を型の注釈で指定できます。詳細については、「データ型の概要」を参照してください。JScript に型を追加するには、型を含むアセンブリをインポートするか、ユーザー定義型 (クラス) を宣言します。

参照

概念

JScript の変数と定数の宣言

その他の技術情報

JScript の変数と定数

JScript のデータ型

JScript の変数と定数の宣言

JScript プログラムでは、使用する変数の名前を指定する必要があります。また、変数ごとに格納するデータ型を指定できます。これらのタスクはどちらも **var** ステートメントで実行します。

型指定された変数および定数の宣言

JScript では、変数を宣言するときに、型の注釈を指定してその型を宣言できます。変数 `count` が **int** 型 (整数型) として宣言されている例を次に示します。初期値が指定されていないため、`count` の値は **int** 型の既定値である 0 となります。

```
var count : int; // An integer variable.
```

変数に初期値を代入することもできます。

```
var count : int = 1; // An initialized integer variable.
```

定数は変数と同じように宣言しますが、初期化する必要があります。定義した定数値は変更できません。次に例を示します。

```
const daysInWeek : int = 7;            // An integer constant.
const favoriteDay : String = "Friday"; // A string constant.
const maxDaysInMonth : int = 31, maxMonthsInYear : int = 12
```

変数に型を指定して宣言した場合、その型で意味のある値を代入する必要があります。たとえば、整数型の変数に文字列を代入しても意味はありません。この場合、**TypeError** 例外がスローされ、コードに型の不一致があることが示されます。**TypeError** は、スクリプトの実行中に発生する例外 (エラー) です。catch ブロックは、JScript プログラムでスローされた例外をキャッチできます。詳細については、「try...catch...finally ステートメント」を参照してください。

各宣言を 1 行ずつ記述した方が読みやすいコードになりますが、複数の変数の型と初期値を同時に宣言できます。次に示すのは、読みにくいコードの例です。

```
var count : int = 1; amount : int = 12, level : double = 5346.9009
```

次のように宣言すると、コードは読みやすくなります。

```
var count : int = 1;
var amount : int = 12;
var level : double = 5346.9009;
```

複数の変数を 1 行で宣言する場合、型の注釈が適用されるのは直前の変数だけです。次のコードでは、`x` には型が指定されていません。既定の型は **Object** であるため、`x` は **Object** です。一方、*y* は **int** です。

```
var x, y : int;
```

型指定されていない変数および定数の宣言

型指定された変数を使うことは必須ではありませんが、型指定されていない変数を使ったプログラムは低速で、エラーが発生しやすくなります。

次の例では、`count` という名前の変数を宣言しています。

```
var count;  // Declare a single declaration.
```

データ型を指定しない場合、変数または定数の既定の型は **Object** になります。値を代入しない場合、変数の既定値は **undefined** になります。次のコードは、コマンド ライン プログラムで既定の型と既定値を表示する例です。

```
var count; // Declare a single declaration using default type and value.
print(count); //Print the value of count.
print(typeof(count)); // Prints undefined.
```

型を宣言しなくても、変数に初期値を指定できます。

```
var count = 1; // An initialized variable.
```

次の例では、1 つの **var** ステートメントで、複数の変数を宣言しています。

```
var count, amount, level;  // multiple declarations with a single var keyword.
```

特定の値を代入せずに変数を宣言および初期化するには、変数に JScript 値の **null** を代入します。次に例を示します。

```
var bestAge = null;
```

値を代入せずに宣言した変数は、存在していますが、値は JScript 値の **undefined** になります。次に例を示します。

```
var currentCount;
var finalCount = 1 * currentCount; // finalCount has the value NaN since currentCount is un
defined.
```

JScript における **null** と **undefined** の主な違いは、**null** が (0 ではありませんが) 0 に変換されるのに対して、**undefined** は特殊な値 **NaN** (非数) に変換される点です。等値演算子 (==) を使用する場合、**null** 値と **undefined** 値は常に等しい値として扱われます。

型指定されていない定数を宣言する手順は、変数を宣言する手順と同様ですが、初期値を指定する必要があります。次に例を示します。

```
const daysInWeek  = 7;
const favoriteDay  = "Friday";
const maxDaysInMonth  = 31, maxMonthsInYear = 12
```

var を使用しない変数の宣言

宣言内で **var** キーワードを使用せずに変数を宣言して、これに値を代入できます。これは "暗黙の宣言" であり、望ましくありません。暗黙の宣言では、指定された名前でグローバル オブジェクトのプロパティが作成されます。作成されたプロパティは、グローバル スコープの参照可能範囲を持つ変数のように機能します。プロシージャ レベルで変数を宣言する場合、一般的に参照可能範囲がグローバル スコープとなることは望ましくありません。この場合は、変数の宣言で **var** キーワードを使用する必要があります。

```
noStringAtAll = ""; // The variable noStringAtAll is declared implicitly.
```

宣言されていない変数を参照することはできません。

```
var volume = length * width; // Error - length and width do not yet exist.
```

メモ :

var キーワードを指定せずに変数を宣言すると、高速モードで実行するときにコンパイル エラーが発生します。JScript では、既定のモードは高速モードです。**var** キーワードを使用しないプログラムをコマンド ラインからコンパイルするには、**/fast-** を使用して fast オプションをオフにする必要があります。ASP.NET で fast オプションをオフにするのは安全ではありません。スレッドに関する問題が発生する場合があります。

参照

処理手順

方法 : コマンド ラインで JScript コードをコンパイルする

概念

JScript の識別子

その他の技術情報

JScript の変数と定数

JScript のデータ型

変数と定数のスコープ

JScript には、グローバル、ローカル、およびクラスの 3 つのスコープがあります。関数やクラス定義の外部で宣言した変数または定数はグローバル変数になり、プログラム内のどこからでも値を参照したり変更したりできます。関数定義の中で変数を宣言した場合、その変数はローカル変数になります。ローカル変数は、関数が実行されるたびに生成され、関数の実行完了と同時に破棄されます。その関数の外部からは、変数にアクセスできません。クラス定義の内部で宣言した変数は、クラス内部で利用できるようになり、グローバル スコープからは直接アクセスできません。詳細については、「クラス ベースのオブジェクト」を参照してください。

説明

C++ などの言語には "ブロック スコープ" の機能があり、中かっこ ({}) で囲んで新しいスコープを定義できます。JScript ではサポートされていません。

ローカル変数には、グローバル変数と同じ名前を付けることができます。同じ名前を付けても、2 つの変数は完全に区別され、別の変数として処理されます。したがって、片方の値を変更しても、他方にはまったく影響はありません。ローカル変数が宣言されている関数の中では、ローカル変数だけが意味を持ちます。これを "参照可能範囲" と呼びます。

```
// Define two global variables.
var name : String = "Frank";
var age : int = "34";

function georgeNameAge() {
   var name : String; // Define a local variable.
   name = "George";   // Modify the local variable.
   age = 42;          // Modify the global variable.
   print(name + " is " + age + " years old.");
}

print(name + " is " + age + " years old.");
georgeNameAge();
print(name + " is " + age + " years old.");
```

このプログラムの出力は、グローバル変数の値を変更せずに、ローカル変数の値を変更できることを示しています。関数内部から行ったグローバル変数の変更は、グローバル スコープの値に影響します。

```
Frank is 34 years old.
George is 42 years old.
Frank is 42 years old.
```

JScript では、変数と定数の宣言はすべてコードの実行前に処理されます。したがって、宣言が条件ブロックや他の構造の内部にあるかどうかは処理には影響しません。変数および定数をすべて検出した後、関数内のコードが実行されます。つまり、ローカル定数の値は、定数の宣言ステートメントが実行されるまで未定義です。また、関数で変数への代入が実行されるまでは、変数も未定義です。

この動作により、予期しない結果となる場合があります。例を次に示します。

```
var aNumber = 100;
var anotherNumber = 200;
function tweak() {
   var s = "aNumber is " + aNumber + " and ";
   s += "anotherNumber is " + anotherNumber + "\n";
   return s;
   if (false)  {
      var aNumber;                // This statement is never executed.
      aNumber = 123;              // This statement is never executed.
      const anotherNumber = 42;   // This statement is never executed.
   } // End of the conditional.
} // End of the function definition.

print(tweak());
```

このプログラムの出力は次のようになります。

```
aNumber is undefined and anotherNumber is undefined
```

`aNumber` は 100 または 123 となり、`anotherNumber` は 200 または 42 となると思えますが、実際にはどちらの値も **undefined** になります。`aNumber` および `anotherNumber` はどちらもローカル スコープで定義されるため、同じ名前のグローバル変数およびグローバル定数を隠ぺいします。ローカル変数およびローカル定数を初期化するコードが実行されていないため、これらの値は **undefined** になります。

高速モードでは、明示的な変数宣言が必要です。高速モードがオフの場合は、暗黙的な変数宣言が必要です。関数内部で暗黙的に宣言された変数、つまり var キーワードを使用せずに代入式の左辺に指定された変数はグローバル変数です。

参照

概念

undefined 値

その他の技術情報

JScript の変数と定数

undefined 値

JScript では、値を代入せずに変数を宣言できます。型の注釈を指定した変数の値は、指定した型の既定値となります。たとえば、数値型の既定値は 0 で、**String** 型の既定値は空の文字列です。ただし、データ型を指定していない変数の初期値は **undefined** になり、データ型は **undefined** になります。同様に、存在しない機能拡張オブジェクト プロパティや配列要素にアクセスするコードは、**undefined** 値を返します。

undefined 値の使用

変数またはオブジェクト プロパティが存在するかどうかを判断するには、**undefined** キーワードと比較するか (宣言された変数またはプロパティでも機能します)、型が "undefined" かどうかを確認します (宣言されていない変数またはプロパティでも機能します)。変数 `x` が宣言されているかどうかを調べるコード例を次に示します。

```
// One method to test if an identifier (x) is undefined.
// This will always work, even if x has not been declared.
if (typeof(x) == "undefined"){
   // Do something.
}
// Another method to test if an identifier (x) is undefined.
// This gives a compile-time error if x is not declared.
if (x == undefined){
   // Do something.
}
```

変数またはオブジェクト プロパティが undefined かどうかを確認するには、値を **null** と比較する方法もあります。**null** を含む変数は、"値がない" か "オブジェクトがない" という状態を示します。つまり、このような変数には有効な数値、文字列、Boolean 値、配列、またはオブジェクトは含まれません。**null** 値を代入することによって、変数自体は削除せずに、変数の内容を消去できます。**undefined** および **null** 値は、等値演算子 (==) を使用して比較します。

メモ :

JScript の等値演算子では、**null** と 0 は等しくありません。C や C++ などの言語とは異なります。

次の例では、オブジェクト `obj` にプロパティ `prop` があるかどうかを確認しています。

```
// A third method to test if an identifier (obj.prop) is undefined.
if (obj.prop == null){
   // Do something.
}
```

次の場合は、比較結果が **true** になります。

- プロパティ `obj.prop` に **null** 値が含まれている場合。
- プロパティ `obj.prop` が存在しない場合。

オブジェクト プロパティが存在するかどうかは、別の方法でも確認できます。**in** 演算子は、指定したプロパティがオブジェクトに存在している場合に **true** を返します。次のコードでは、プロパティ `prop` がオブジェクト `obj` に存在している場合に **true** を返します。

```
if ("prop" in someObject)
// someObject has the property 'prop'
```

オブジェクトからプロパティを削除するには、**delete** 演算子を使用します。

参照

関連項目

null リテラル
undefined プロパティ
in 演算子
delete 演算子

その他の技術情報

JScript の変数と定数
JScript のデータ
JScript のデータ型
データ型 (JScript)

JScript オブジェクト

JScript オブジェクトは、データと機能をカプセル化したものです。オブジェクトは、プロパティ (値) とメソッド (関数) で構成されます。プロパティはオブジェクトのデータ コンポーネントで、メソッドはデータやオブジェクトを操作する機能を提供します。JScript では、組み込みオブジェクト、プロトタイプ ベースのオブジェクト、クラス ベースのオブジェクト、ホスト オブジェクト (ASP.NET の **Response** などのホストによって提供されます)、および .NET Framework クラス (外部コンポーネント) の 5 種類のオブジェクトをサポートしています。

new 演算子と選択したオブジェクトのコンストラクタ関数を組み合わせて使用すると、オブジェクトのインスタンスが作成および初期化されます。コンストラクタの使用例を次に示します。

```
var myObject = new Object();           // Creates a generic object.
var birthday = new Date(1961, 5, 10);  // Creates a Date object.
var myCar : Car = new Car("Pinto");    // Creates a user-defined object.
```

JScript では、2 種類のユーザー定義オブジェクト (クラス ベースのオブジェクトとプロトタイプ ベースのオブジェクト) がサポートされます。どちらにも、長所と短所があります。プロトタイプ ベースのオブジェクトは動的に拡張できますが、処理速度が遅く、他の .NET Framework 言語のオブジェクトと効率よく相互運用できません。クラス ベースのオブジェクトは、既存の .NET Framework クラスを拡張でき、タイプ セーフで、効率よく処理できます。クラス ベースのオブジェクトは、**expando** 修飾子を指定して定義することで、プロトタイプ ベースのオブジェクトのように動的に拡張できます。

このセクションの内容

組み込みオブジェクト

JScript スクリプトで使用される一般的なオブジェクトと、その使用方法を説明する情報へのリンクの一覧を示します。

クラス ベースのオブジェクト

JScript のクラス ベースのオブジェクト モデルの使用方法を説明します。また、クラスを (メソッド、フィールド、およびプロパティと共に) 定義する方法、他のクラスから継承するクラスを定義する方法、および expando クラスを定義する方法についても説明します。

プロトタイプ ベースのオブジェクト

JScript のプロトタイプ ベースのオブジェクト モデルの使用方法を説明します。また、カスタム コンストラクタ関数とプロトタイプ ベースのオブジェクトの継承に関する情報へのリンクを示します。

関連するセクション

JScript のデータ型

JScript のプリミティブ データ型、参照データ型、および .NET Framework データ型の使用方法について説明するトピックへのリンクを示します。

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

ASP.NET の概要

ASP.NET について紹介します。JScript を含む .NET 互換言語での ASP.NET の使用方法と、エンタープライズ クラスの Web アプリケーションの作成方法を説明し、参照情報へのリンクを示します。

.NET Framework クラス ライブラリの概要

.NET Framework クラス ライブラリについて紹介します。名前付け規則とシステムの名前空間を説明し、参照情報へのリンクを示します。

組み込みオブジェクト

JScript には、言語仕様の一部として 16 の組み込みオブジェクトが用意されています。各組み込みオブジェクトには、関連するメソッドとプロパティがあります。メソッドとプロパティの詳細については、言語リファレンスを参照してください。ここでは、一般的に使用されるオブジェクトのいくつかを説明し、組み込みオブジェクトの基本構文と使用方法を示します。

このセクションの内容

JScript の Array オブジェクト

配列オブジェクトの使用方法、配列オブジェクトの expando プロパティの利用方法、および型指定された配列との比較について説明します。

JScript の Date オブジェクト

指定できる日付の範囲と、現在の日付と時刻または任意の日付と時刻でオブジェクトを作成する方法を説明します。

JScript の Math オブジェクト

メソッドとプロパティを使用して数値データを操作する方法を説明します。

JScript の Number オブジェクト

Number オブジェクトの目的と、そのプロパティの意味を説明します。

JScript の Object オブジェクト

オブジェクトに expando プロパティおよびメソッドを追加する方法を説明します。また、ドット演算子と添字演算子によるオブジェクト メンバへのアクセスの違いを説明します。

JScript の String オブジェクト

文字列オブジェクトの目的を説明します。また、リテラル文字列で、**String** オブジェクトのメソッドを使用する方法についても説明します。

関連するセクション

JScript オブジェクト

JScript の組み込みオブジェクトの構文と使用方法を説明するトピックへのリンクを示します。

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

オブジェクト

JScript 言語で用意されているオブジェクトの一覧と、各オブジェクトの正しい使用方法と構文を説明する言語リファレンス情報へのリンクを示します。

JScript の Array オブジェクト

Array オブジェクトは、関連するデータをグループ化する変数です。インデックスまたは添字と呼ばれる一意な数字で、配列内の各データを参照します。配列に格納されたデータにアクセスするには、配列識別子とインデックスを添字演算子の "[]" で組み合わせます。たとえば `theMonths[0]` のように指定します。

配列の作成

新しい配列を作成する場合は、**new** 演算子と Array コンストラクタを使用します。次の例では、配列コンストラクタを使用して長さが 12 の配列を作成し、作成した配列にデータを入力しています。

```
var theMonths = new Array(12);
theMonths[0] = "Jan";
theMonths[1] = "Feb";
theMonths[2] = "Mar";
theMonths[3] = "Apr";
theMonths[4] = "May";
theMonths[5] = "Jun";
theMonths[6] = "Jul";
theMonths[7] = "Aug";
theMonths[8] = "Sep";
theMonths[9] = "Oct";
theMonths[10] = "Nov";
theMonths[11] = "Dec";
```

Array キーワードを使用して配列を作成すると、JScript ではその配列に **length** プロパティが割り当てられます。数値を指定しない場合、長さは 0 に設定され、配列にはエントリが含まれません。数値を指定すると、長さはその数値に設定されます。複数のパラメータを指定すると、パラメータは配列のエントリとして使用されます。また、パラメータの数が **length** プロパティに割り当てられます。次に例を示します。これは上記の例と同じ内容です。

```
var theMonths = new Array("Jan", "Feb", "Mar", "Apr", "May", "Jun",
"Jul", "Aug", "Sep", "Oct", "Nov", "Dec");
```

配列リテラルを使用して配列にデータを入力することもできます。詳細については、「配列データ」を参照してください。

Array オブジェクトは "疎" 配列です。配列に 0、1、および 2 の 3 つの要素がある場合、要素 3 ～ 49 がなくても要素 50 が存在できます。**Array** オブジェクトに要素を追加すると、**length** プロパティの値は自動的に変更されます。JScript の配列のインデックスは、常に 1 ではなく 0 から開始されます。そのため、**length** プロパティは、常に配列の最大のインデックスより 1 だけ大きい数値になります。

配列の expando プロパティの使用方法

配列オブジェクトは、JScript **Object** オブジェクトに基づく他のオブジェクトと同様に、expando プロパティをサポートします。expando プロパティは、配列に対して動的に追加および削除できる、配列のインデックスに似た新しいプロパティです。配列のインデックスは整数ですが、expando プロパティは文字列です。また、expando プロパティを追加または削除しても、**length** プロパティは変化しません。

次に例を示します。

```
// Initialize an array with three elements.
var myArray = new Array("Hello", 42, new Date(2000,1,1));
print(myArray.length); // Prints 3.
// Add some expando properties. They will not change the length.
myArray.expando = "JScript";
myArray["another Expando"] = "Windows";
print(myArray.length); // Still prints 3.
```

型指定された配列

上で示した `theMonths` 配列をより速く作成するには、型指定された (ネイティブな) 配列を作成します。この例では、文字列型の配列になります。

```
var theMonths : String[] = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep",
"Oct", "Nov", "Dec"];
```

型指定された配列の要素には、JScript の配列オブジェクトの要素よりも速くアクセスできます。型指定された配列は、他の .NET Framework 言語の配列と互換性を持つ、タイプ セーフな配列です。

JScript の **Array** オブジェクトは非常に柔軟で、リスト、キュー、スタックなどの用途に適しています。ネイティブな配列は、サイズが固定された同一の型の項目を格納する場合に適しています。一般的に、**Array** オブジェクトの特殊な機能 (動的なサイズ変更など) が必要でない場合は、型指定された配列を使用してください。

すべての非破壊的な JScript の **Array** メソッド (長さを変更しないメソッド) は、型指定された配列で呼び出すことができます。

参照

関連項目

Array オブジェクト

その他の技術情報

組み込みオブジェクト

JScript の Date オブジェクト

JScript の **Date** オブジェクトは、任意の日付や時間を表したり、現在のシステムの日付を取得したり、日付の差を計算したりするときに使用できます。このオブジェクトには、定義済みのプロパティとメソッドがあります。**Date** オブジェクトは、曜日、年、月、日、時、分、秒、およびミリ秒を格納します。この情報は、世界協定時刻 (UTC) を基準にしており、1970 年 1 月 1 日 00:00:00.000 から経過したミリ秒の数値を基に算出されます。世界協定時刻は、これまでグリニッジ標準時と呼ばれていました。JScript では、紀元前 250,000 年から西暦 255,000 年までの日付を処理できます。ただし、一部の書式機能は、西暦 0 ～ 9999 年の範囲の日付でしかサポートされていません。

Date オブジェクトの作成

新しい **Date** オブジェクトを作成するには、**new** 演算子を使用します。現在の年の経過日数、および残りの日数を計算する例を次に示します。

```
// Get the current date and read the year.
var today : Date = new Date();
// The getYear method should not be used. Always use getFullYear.
var thisYear : int = today.getFullYear();

// Create two new dates, one for January first of the current year,
// and one for January first of next year. The months are numbered
// starting with zero.
var firstOfYear : Date = new Date(thisYear,0,1);
var firstOfNextYear : Date = new Date(thisYear+1,0,1);

// Calculate the time difference (in milliseconds) and
// convert the differnce to days.
const millisecondsToDays = 1/(1000*60*60*24);
var daysPast : double = (today - firstOfYear)*millisecondsToDays;
var daysToGo : double = (firstOfNextYear - today)*millisecondsToDays;

// Display the information.
print("Today is: "+today+".");
print("Days since first of the year: "+Math.floor(daysPast));
print("Days until the end of the year: "+Math.ceil(daysToGo));
```

このプログラムの出力は次のようになります。

```
Today is: Sun Apr 1 09:00:00 PDT 2001.
Days since first of the year: 90
Days until the end of the year: 275
```

参照

関連項目

Date オブジェクト

その他の技術情報

組み込みオブジェクト

JScript の Math オブジェクト

Math オブジェクトには、組み込みのプロパティとメソッドが多数用意されています。プロパティは特定の数値です。

Math オブジェクトの使用

これらの特定の数値の 1 つに、π (約 3.14159...) があります。これは **Math.PI** プロパティの値で、次のように使用されます。

```
// A radius variable is declared and assigned a numeric value.
var radius = 4;
var area = Math.PI * radius * radius;
// Note capitalization of Math and PI.
```

Math オブジェクトの組み込みメソッドに、指数演算メソッドの **pow** があります。これは、数値を指定した回数だけ累乗します。次の例では、π と指数演算の両方を使用しています。

```
// This formula calculates the volume of a sphere with the given radius.
var volume = (4/3)*(Math.PI*Math.pow(radius,3));
```

別の例を示します。

```
var x = Math.floor( Math.random()*10 ) + 1;
```

Math オブジェクトは明示的に作成できませんが、常にプログラムで利用できます。

参照

関連項目

Math オブジェクト

その他の技術情報

組み込みオブジェクト

JScript の Number オブジェクト

Number オブジェクトの主な目的は、既定の数値型である **Number** 型で使用されるプロパティおよびメソッドをまとめておくことです。**Number** オブジェクトのプロパティで提供される数値定数を次の表に示します。

Number オブジェクトのプロパティ

プロパティ	説明
MAX_VALUE	正または負の最大の数は、約 1.79E+308 となります。値はシステムによって異なります。
MIN_VALUE	正または負の最小の数は、約 2.22E-308 となります。値はシステムによって異なります。
NaN	特殊な非数値 (非数) です。
POSITIVE_INFINITY	表現できる最も大きい正の値 (Number.MAX_VALUE) よりも大きい値は、自動的にこの値に変換されます。infinity と表示されます。
NEGATIVE_INFINITY	表現できる最も小さい負の値 (-Number.MAX_VALUE) よりも小さい値は、自動的にこの値に変換されます。-infinity と表示されます。

Number.NaN は、"非数" として定義される特殊なプロパティです。数値コンテキストに数値として表現できない式が使用されている場合は、**Number.NaN** が返されます。たとえば、文字列 "Hello" や、0/0 (0 割る 0) が数値として使用されている場合は、**NaN** が返されます。**NaN** を任意の数値および自身と比較すると、等しくないと評価されます。**NaN** を確認するには、**Number.NaN** と比較するのではなく、**Global** オブジェクトの **isNaN** メソッドを使用します。

Number オブジェクトの **toLocaleString** メソッドは、数値を表す文字列値を、ホスト環境の現在のロケールに対応する形式で生成します。使用される書式によって、整数部の桁が (ロケール基づく文字で) 位取りされ、大きな数字が読みやすくなります。詳細については、「toLocaleString メソッド」を参照してください。

参照

関連項目

Number オブジェクト
toLocaleString メソッド

その他の技術情報

組み込みオブジェクト

JScript の Object オブジェクト

Object オブジェクトを基本とする JScript のオブジェクトはすべて、expando プロパティ、またはプログラムの実行中に追加および削除できるプロパティをサポートします。

Object オブジェクトの使用

これらのプロパティには、数値を含む任意の名前を指定できます。単純な識別子であるプロパティの名前は、オブジェクト名の後ろにピリオドに続いて記述できます。次に例を示します。

```
var myObj = new Object();
// Add two expando properties, 'name' and 'age'
myObj.name = "Fred";
myObj.age = 53;
```

また、添字演算子 **[]** を使用してオブジェクトのプロパティにアクセスすることもできます。プロパティの名前が単純な識別子でない場合や、プロパティの名前がスクリプトの作成時にわからない場合は、この方法を使用します。単純な識別子を含め、角かっこ内の任意の式では、プロパティをインデックス指定できます。JScript のすべての expando プロパティの名前は、オブジェクトに追加される前に文字列に変換されます。

添字演算子を使用する場合、オブジェクトは "連想配列" として扱われます。連想配列は、任意のデータ値を任意の文字列に動的に関連付けるデータ構造です。次の例では、単純な識別子を持たない expando プロパティが追加されています。

```
var myObj = new Object();
// This identifier contains spaces.
myObj["not a simple identifier"] = "This is the property value";
// This identifier is a number.
myObj[100] = "100";
```

添字演算子は、通常、配列の要素にアクセスするために使用されます。オブジェクトで使用された場合、かっこで囲まれたインデックス部分は、常にリテラル文字列として表現されたプロパティ名になります。

Array オブジェクトには、新しい要素が追加されたときに変更される **length** という特殊なプロパティがあります。通常のオブジェクトには、添え字演算子を使用してプロパティが追加された場合に変更される length プロパティはありません。

オブジェクトのプロパティにアクセスする 2 つの方法には、次の表に示すように重要な違いがあります。

演算子	プロパティ名の扱い	プロパティ名の操作上の差異
ピリオド (.)	識別子	データとして操作できません。
インデックス ([])	リテラル文字列	データとして操作できます。

ユーザーの入力データによってオブジェクトを作成するなど、実行時までプロパティ名がわからない場合は、上の表で説明した違いを有効に利用してください。連想配列からすべてのプロパティを取得するには、**for ... in** ループを使用する必要があります。

参照

関連項目

Object オブジェクト

その他の技術情報

組み込みオブジェクト

JScript の String オブジェクト

JScript の **String** オブジェクトは、単語や文などのテキスト データを表します。通常、**new** 演算子を使って String オブジェクトを明示的に作成することはありません。変数にリテラル文字列を代入することで、暗黙的に作成されます。詳細については、「String オブジェクト」を参照してください。

String オブジェクトの使用

String オブジェクトには、多くの組み込みメソッドがあります。**substring** メソッドは組み込みメソッドの 1 つで、文字列の一部を返します。引数として 2 つの数値を指定します。最初の引数は、部分文字列の開始位置を表す、0 から始まるインデックス番号です。2 番目の引数は、部分文字列の終端を表します。

```
var aString : String = "0123456789";
var aChunk : String = aString.substring(4, 7);  // Sets aChunk to "456".
```

String オブジェクトにも、**length** プロパティがあります。このプロパティには、文字列内の文字数が含まれます (空の文字列の場合は 0)。これは、計算でそのまま使用できる数値です。次の例では、リテラル文字列の長さを取得します。

```
var howLong : int = "Hello World".length  // Sets the howLong variable to 11.
```

参照

関連項目

String オブジェクト

概念

文字列データ

その他の技術情報

組み込みオブジェクト

クラス ベースのオブジェクト

JScript は、クラス ベースのオブジェクト指向プログラミング言語です。したがって、他のクラスから継承できるクラスを定義できます。定義したクラスは、メソッド、フィールド、プロパティ、およびサブクラスを持つことができます。継承を利用すると、既存のクラスを基にしてクラスを作成し、選択した基本クラスのメソッドとプロパティをオーバーライドできます。JScript のクラスは C++ および C# のクラスに似ており、プロトタイプ ベースのオブジェクトとは大きく異なります。

このセクションの内容

独自のクラスの作成

クラスとフィールド、メソッド、およびコンストラクタを定義する方法を説明します。

高度なクラス作成

クラスとプロパティを定義する方法、クラスから継承する方法、および expando プロパティをサポートするクラスを作成する方法を説明します。

関連するセクション

JScript オブジェクト

JScript の組み込みオブジェクトの構文と使用方法を説明するトピックへのリンクを示します。

JScript の修飾子

クラス メンバの参照可能範囲の制御に使用する修飾子、クラスの継承、およびクラスの動作について説明します。

プロトタイプ ベースのオブジェクト

JScript のプロトタイプ ベースのオブジェクト モデルの使用方法を説明します。また、カスタム コンストラクタ関数とプロトタイプ ベースのオブジェクトの継承に関する情報へのリンクを示します。

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

独自のクラスの作成

class ステートメントはクラスを定義します。既定では、クラス メンバにはパブリックにアクセスできます。つまり、クラスにアクセスできるコードからは、クラス メンバにもアクセスできます。詳細については、「JScript の修飾子」を参照してください。

クラスとフィールド

フィールドはオブジェクトで使用されるデータを定義し、プロトタイプ ベースのオブジェクトのプロパティに似ています。2 つのフィールドを持つ簡単なクラスの例を次に示します。クラスのインスタンスは、**new** 演算子で作成されます。

```
class myClass {
   const answer : int = 42; // Constant field.
   var distance : double;   // Variable field.
}

var c : myClass = new myClass;
c.distance = 5.2;
print("The answer is " + c.answer);
print("The distance is " + c.distance);
```

このプログラムの出力は次のようになります。

```
The answer is 42
The distance is 5.2
```

クラスとメソッド

クラスにはメソッドを含めることもできます。メソッドは、クラスに含まれる関数です。メソッドは、オブジェクトのデータを操作するための機能を定義します。上で定義した `myClass` クラスを次のように再定義して、メソッドを含めます。

```
class myClass {
   const answer : int = 42;         // Constant field.
   var distance : double;           // Variable field.
   function sayHello() :String {    // Method.
      return "Hello";
   }
}

var c : myClass = new myClass;
c.distance = 5.2;
print(c.sayHello() + ", the answer is " + c.answer);
```

このプログラムの出力は次のようになります。

```
Hello, the answer is 42
```

クラスとコンストラクタ

クラスにはコンストラクタを定義できます。コンストラクタはクラスと同じ名前を持つメソッドで、**new** 演算子でクラスを作成するときに実行されます。コンストラクタには戻り値の型を指定しません。次の例では、`myClass` クラスにコンストラクタを追加しています。

```
class myClass {
   const answer : int = 42;         // Constant field.
   var distance : double;           // Variable field.
   function sayHello() :String {    // Method.
      return "Hello";
   }
   // This is the constructor.
   function myClass(distance : double) {
      this.distance = distance;
   }
```

```
}

var c : myClass = new myClass(8.5);
print("The distance is " + c.distance);
```

このプログラムの出力は次のようになります。

```
The distance is 8.5
```

参照

概念

高度なクラス作成

その他の技術情報

クラス ベースのオブジェクト

JScript オブジェクト

高度なクラス作成

JScript のクラスを定義するときに、プロパティを割り当てることができます。また、定義したクラスは、続いて他のクラスからも継承できます。フィールドに似たクラス メンバであるプロパティを使用すると、データへのアクセス方法を詳細に制御できます。継承を使用することにより、クラスは他のクラスを拡張 (または、他のクラスに動作を追加) できます。

クラスの定義で、クラスのインスタンスに expando プロパティをサポートさせることができます。つまり、クラス ベースのオブジェクトは、オブジェクトに動的に追加されるプロパティとメソッドを持つことができます。クラス ベースの expando オブジェクトは、プロトタイプ ベースのオブジェクトと同様の機能をいくつか提供します。

クラスとプロパティ

JScript では、**function get** ステートメントおよび **function set** ステートメントを使用してプロパティを指定します。いずれかのアクセサまたは両方のアクセサを指定して、読み取り専用、書き込み専用、または読み取り/書き込みのプロパティを作成できます。ただし、書き込み専用のプロパティはあまり使用されず、クラスのデザインに問題があることを示している場合があります。

呼び出し元のプログラムは、フィールドにアクセスする場合と同じ方法でプロパティにアクセスします。主な違いは、プロパティではアクセスの実行に取得関数と設定関数を使用するのに対し、フィールドには直接アクセスすることです。クラスでプロパティを利用すると、有効な情報だけが入力されていることを確認したり、プロパティの読み取りや設定が実行された回数を追跡したりできます。また、動的な情報を返すこともできます。

通常、プロパティはクラスのプライベート フィールドまたはプロテクト フィールドにアクセスするために使用されます。プライベート フィールドには **private** 修飾子が指定され、このフィールドには同じクラスの他のメンバしかアクセスできません。プロテクト フィールドには **protected** 修飾子が指定され、このフィールドには同じクラスまたは派生クラスのメンバしかアクセスできません。詳細については、「JScript の修飾子」を参照してください。

次の例では、プロパティを使用してプロテクト フィールドにアクセスしています。フィールドは外部コードによって値が変更されないように保護されますが、派生クラスからはアクセスできます。

```
class Person {
   // The name of a person.
   // It is protected so derived classes can access it.
   protected var name : String;

   // Define a getter for the property.
   function get Name() : String {
      return this.name;
   }

   // Define a setter for the property which makes sure that
   // a blank name is never stored in the name field.
   function set Name(newName : String) {
      if (newName == "")
         throw "You can't have a blank name!";
      this.name = newName;
   }
   function sayHello() {
      return this.name + " says 'Hello!'";
   }
}

// Create an object and store the name Fred.
var fred : Person = new Person();
fred.Name = "Fred";
print(fred.sayHello());
```

このコードの出力は次のようになります。

```
Fred says 'Hello!'
```

`Name` プロパティに空白の名前を代入すると、エラーが発生します。

クラスからの継承

他のクラスを基にクラスを定義する場合は、**extends** キーワードを使用します。JScript では、ほとんどの共通言語仕様 (CLS: Common

Language Specification) 準拠のクラスを拡張できます。**extends** キーワードを使用して定義されたクラスは派生クラスと呼ばれます。また、拡張元のクラスは基本クラスと呼ばれます。

次の例では、新しい `Student` クラスを定義しています。このクラスは、前の例の `Person` クラスを拡張しています。Student クラスでは、基本クラスで定義されている `Name` プロパティを再利用しています。また、新しい `sayHello` メソッドを定義して、基本クラスの `sayHello` メソッドをオーバーライドします。

```
// The Person class is defined in the code above.
class Student extends Person {
   // Override a base-class method.
   function sayHello() {
      return this.name + " is studying for finals.";
   }
}

var mary : Person = new Student;
mary.Name = "Mary";
print(mary.sayHello());
```

このコードの出力は次のようになります。

```
Mary is studying for finals.
```

派生クラスでメソッドを再定義しても、基本クラスの対応するメソッドは変更されません。

expando オブジェクト

汎用オブジェクトを expando にするだけの場合は、**Object** コンストラクタを使用します。

```
// A JScript Object object, which is expando.
var o = new Object();
o.expando = "This is an expando property.";
print(o.expando);  // Prints This is an expando property.
```

クラスの 1 つを expando にする場合は、**expando** 修飾子を指定してクラスを定義します。expando メンバには、インデックス (**[]**) 表記を使用してアクセスできます。ドット (.) 表記ではアクセスできません。

```
// An expando class.
expando class MyExpandoClass {
   function dump() {
      // print all the expando properties
      for (var x : String in this)
         print(x + " = " + this[x]);
   }
}
// Create an instance of the object and add some expando properties.
var e : MyExpandoClass = new MyExpandoClass();
e["answer"] = 42;
e["greeting"] = "hello";
e["new year"] = new Date(2000,0,1);
print("The contents of e are...");
// Display all the expando properites.
e.dump();
```

このプログラムの出力は次のようになります。

```
The contents of e are...
answer = 42
greeting = hello
new year = Sat Jan 1 00:00:00 PST 2000
```

参照

概念

JScript の修飾子
独自のクラスの作成

その他の技術情報

クラス ベースのオブジェクト
JScript オブジェクト

プロトタイプ ベースのオブジェクト

JScript はオブジェクト指向のプログラミング言語であるため、カスタム コンストラクタ関数と継承をサポートしています。コンストラクタ関数 (コンストラクタ) を使用すると、独自のプロトタイプ ベースのオブジェクトをデザインおよび実装できます。継承を利用すると、プロトタイプ ベースのオブジェクトで、動的に追加および削除できる共通のプロパティとメソッドのセットを共有できます。

通常は、プロトタイプ ベースのオブジェクトではなく、クラス ベースのオブジェクトを使用してください。クラス ベースのオブジェクトは、他の .NET Framework 言語で記述されたメソッドに渡すことができます。また、クラス ベースのオブジェクトはタイプ セーフであり、効率的なコードを生成します。

このセクションの内容

コンストラクタ関数による独自のオブジェクトの作成

コンストラクタ関数を使用してオブジェクトとそのプロパティおよびメソッドを作成する方法を説明します。

高度なオブジェクト作成 (JScript)

継承を使用して、指定したコンストラクタ関数で作成されたオブジェクトに、共通のプロパティとメソッドのセットを追加する方法を説明します。

関連するセクション

JScript オブジェクト

JScript の組み込みオブジェクトの構文と使用方法を説明するトピックへのリンクを示します。

クラス ベースのオブジェクト

JScript のクラス ベースのオブジェクト モデルの使用方法を説明します。また、クラスを (メソッド、フィールド、およびプロパティと共に) 定義する方法、他のクラスから継承するクラスを定義する方法、および expando クラスを定義する方法についても説明します。

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

コンストラクタ関数による独自のオブジェクトの作成

JScript には、コンストラクタ関数を定義して、スクリプト内で使用できるプロトタイプ ベースのカスタム オブジェクトを作成する機能があります。プロトタイプ ベースのオブジェクトのインスタンスを作成するには、最初にコンストラクタ関数を定義する必要があります。これによって、新しいオブジェクトが作成され初期化されます。つまり、プロパティが作成され、初期値が代入されます。初期化が終了すると、作成したオブジェクトへの参照がコンストラクタから返されます。コンストラクタ内部では、作成したオブジェクトは **this** ステートメントで参照されます。

コンストラクタとプロパティ

次の例では、`pasta` オブジェクトのコンストラクタ関数を定義しています。コンストラクタは、**this** ステートメントを使ってオブジェクトを初期化します。

```
// pasta is a constructor that takes four parameters.
function pasta(grain, width, shape, hasEgg) {
   this.grain = grain;    // What grain is it made of?
   this.width = width;    // How many centimeters wide is it?
   this.shape = shape;    // What is the cross-section?
   this.hasEgg = hasEgg;  // Does it have egg yolk as a binder?
}
```

オブジェクトのコンストラクタを定義したら、**new** 演算子を使用してそのオブジェクトのインスタンスを作成できます。次の例では、`pasta` コンストラクタを使用して `spaghetti` オブジェクトと `linguine` オブジェクトを作成します。

```
var spaghetti = new pasta("wheat", 0.2, "circle", true);
var linguine = new pasta("wheat", 0.3, "oval", true);
```

オブジェクトのインスタンスにプロパティを動的に追加できますが、変更はそのインスタンスにしか影響を与えません。

```
// Additional properties for spaghetti. The properties are not added
// to any other pasta objects.
spaghetti.color = "pale straw";
spaghetti.drycook = 7;
spaghetti.freshcook = 0.5;
```

コンストラクタ関数を変更せずに、オブジェクトのすべてのインスタンスにプロパティを追加する場合は、コンストラクタのプロトタイプ オブジェクトにプロパティを追加します。詳細については、「高度なオブジェクト作成 (JScript)」を参照してください。

```
// Additional property for all pasta objects.
pasta.prototype.foodgroup = "carbohydrates";
```

コンストラクタとメソッド

オブジェクトの定義にメソッド (関数) を含めることができます。オブジェクトの定義にメソッドを含める方法の 1 つは、他の場所で定義された関数を参照するプロパティをコンストラクタ関数に追加することです。これらの関数は、コンストラクタ関数と同様に、現在のオブジェクトを **this** ステートメントで参照します。

前の例で定義した `pasta` コンストラクタ関数を拡張して、関数がオブジェクトの値を表示するときに呼び出される **toString** メソッドを追加します。通常は、文字列が必要な状況でオブジェクトを使用すると、JScript がオブジェクトの **toString** メソッドを使用します。**toString** メソッドを明示的に呼び出す必要はほとんどありません。

```
// pasta is a constructor that takes four parameters.
// The properties are the same as above.
function pasta(grain, width, shape, hasEgg) {
   this.grain = grain;    // What grain is it made of?
   this.width = width;    // How many centimeters wide is it?
   this.shape = shape;    // What is the cross-section?
   this.hasEgg = hasEgg;  // Does it have egg yolk as a binder?
   // Add the toString method (defined below).
   // Note that the function name is not followed with parentheses;
   // this is a reference to the function itself, not a function call.
```

```
   this.toString = pastaToString;
}

// The function to display the contents of a pasta object.
function pastaToString() {
   return "Grain: " + this.grain + "\n" +
          "Width: " + this.width + " cm\n" +
          "Shape: " + this.shape + "\n" +
          "Egg?:  " + Boolean(this.hasEgg);
}

var spaghetti = new pasta("wheat", 0.2, "circle", true);
// Call the method explicitly.
print(spaghetti.toString());
// The print statement takes a string as input, so it
//  uses the toString() method to display the properties
// of the spaghetti object.
print(spaghetti);
```

出力は次のようになります。

```
Grain: wheat
Width: 0.2 cm
Shape: circle
Egg?:  true
Grain: wheat
Width: 0.2 cm
Shape: circle
Egg?:  true
```

参照

その他の技術情報

プロトタイプ ベースのオブジェクト

JScript オブジェクト

高度なオブジェクト作成 (JScript)

JScript では、プロトタイプ ベースのオブジェクトで継承がサポートされています。プロトタイプ ベースのオブジェクトは、継承を利用して、動的に追加および削除できる共通のプロパティとメソッドのセットを共有できます。また、各オブジェクトは既定の動作をオーバーライドできます。

プロトタイプ ベースのオブジェクトの作成

プロトタイプ ベースのオブジェクトのインスタンスを作成するには、最初にコンストラクタ関数を定義する必要があります。詳細については、「コンストラクタ関数による独自のオブジェクトの作成」を参照してください。コンストラクタを記述したら、プロトタイプ オブジェクトを使用して、継承されたプロパティと共有メソッドを作成できます。プロトタイプ オブジェクトは、それ自体が各コンストラクタのプロパティになります。コンストラクタはインスタンス固有の情報をオブジェクトに提供しますが、プロトタイプ オブジェクトはオブジェクトに固有の情報とメソッドをオブジェクトに提供します。

メモ :

オブジェクトのすべてのインスタンスに影響を与えるには、コンストラクタの **prototype** オブジェクトに変更を加える必要があります。オブジェクトのいずれかのインスタンスのプロトタイプ プロパティを変更しても、同じオブジェクトの他のインスタンスには影響がありません。

プロトタイプ オブジェクトのプロパティおよびメソッドは、オブジェクトの各インスタンスに参照渡しでコピーされます。したがって、すべてのインスタンスが同じ情報にアクセスします。1 つのインスタンスでプロトタイプ プロパティの値を既定値から変更しても、他のインスタンスは変更の影響を受けません。カスタム コンストラクタの `Circle` を使用する例を次に示します。**this** ステートメントは、メソッドでオブジェクトのメンバにアクセスできるようにします。

```
// Define the constructor and add instance specific information.
function Circle (radius) {
    this.r = radius;  // The radius of the circle.
}
// Add a property the Circle prototype.
Circle.prototype.pi = Math.PI;
function ACirclesArea () {
   // The formula for the area of a circle is pi*r^2.
   return this.pi * this.r * this.r;
}
// Add a method the Circle prototype.
Circle.prototype.area = ACirclesArea;
// This is how you would invoke the area function on a Circle object.
var ACircle = new Circle(2);
var a = ACircle.area();
```

この方法を使って、既存の (プロトタイプ オブジェクトを持つ) コンストラクタ関数に対して、追加のプロパティを定義できます。この操作は高速モードがオフの場合にだけ有効です。詳細については、「/fast」を参照してください。

たとえば、Visual Basic の **Trim** 関数のように、文字列の先頭または末尾のスペースを削除する場合は、**String** プロトタイプ オブジェクトに独自のメソッドを作成して、スクリプト内のすべての文字列がこのメソッドを自動的に継承するようにできます。この例では、正規表現リテラルを使用して空白を削除しています。詳細については、「Regular Expression オブジェクト」を参照してください。

```
// Add a function called trim as a method of the prototype
// object of the String constructor.
String.prototype.trim = function() {
   // Use a regular expression to replace leading and trailing
   // spaces with the empty string
   return this.replace(/(^\s*)|(\s*$)/g, "");
}

// A string with spaces in it
var s = "    leading and trailing spaces    ";
print(s + " (" + s.length + ")");

// Remove the leading and trailing spaces
s = s.trim();
print(s + " (" + s.length + ")");
```

/fast- フラグを指定してこのプログラムをコンパイルすると、出力は次のようになります。

```
    leading and trailing spaces     (35)
leading and trailing spaces (27)
```

参照

その他の技術情報

プロトタイプ ベースのオブジェクト
JScript オブジェクト

JScript の修飾子

JScript の修飾子は、クラス、インターフェイス、またはクラスやインターフェイスのメンバの、動作および参照可能範囲を変更します。修飾子はクラスやインターフェイスを定義するときに指定しますが、通常は必須ではありません。

参照可能範囲の修飾子

参照可能範囲の修飾子は、外部コードによるクラス、インターフェイス、およびクラスやインターフェイスのメンバへのアクセスを制限します。参照可能範囲を制限することで、特殊な内部メソッドやフィールドの呼び出しを禁止し、適切なオブジェクト指向プログラミングを促進できます。

既定では、クラスにアクセスできるコードは、そのクラスのメンバにもアクセスできます。参照可能範囲の修飾子を使用すると、外部コードから特定のクラス メンバへのアクセスを選択的に禁止して、同じパッケージのクラスだけにメンバへのアクセスを許可したり、派生クラスだけにクラス メンバへのアクセスを許可したりできます。

参照可能範囲の修飾子は、グローバル関数やグローバル変数には適用できません。同時に指定できる参照可能範囲の修飾子は、**protected** と **internal** だけです。

参照可能範囲の修飾子	対象	説明
public	クラス、クラスのメンバ、インターフェイス、インターフェイスのメンバ、または列挙型	クラスにアクセスできるコードは、参照可能範囲の制限を受けることなく、クラスのメンバを参照できます。JScript では、クラス、インターフェイス、およびクラスとインターフェイスのメンバは既定でパブリックです。
private	クラスのメンバ	クラスのメンバは、宣言されたクラス内だけで参照可能です。派生クラスからは参照できません。現在のクラス以外のコードからは、プライベート メンバにアクセスできません。
protected	クラスのメンバ	クラスのメンバは、宣言されたクラス内と、そのクラスの派生クラスだけで参照可能です。protected 修飾子は、パッケージ スコープのクラスには使用できませんが、入れ子になったクラスでは使用できます。
internal	クラス、クラスのメンバ、列挙型	クラス、クラスのメンバ、または列挙型は、宣言されたパッケージ内のどこからでも参照できます。パッケージの外部からは参照できません。

継承の修飾子

継承の修飾子は、派生クラスのメソッドおよびプロパティが、基本クラスのメソッドおよびプロパティをオーバーライドする方法を制御します。この修飾子を使用すると、作成するクラスが派生クラスのメソッドによってオーバーライドされるかどうかを管理できます。

既定では、派生クラスでバージョン セーフ属性の **hide** が使用されていない場合、基本クラスのメソッドは派生クラスのメソッドによってオーバーライドされます。この属性はオーバーライドを妨げます。継承の修飾子を使用すると、特定のメソッドがオーバーライドされるかどうかを制御できます。

場合によっては、基本クラスのメソッドがオーバーライドされないようにする必要があります。たとえば、パッケージでクラスを定義する場合、**final** 修飾子を使用して、派生クラスがクラスのメソッドやプロパティを変更しないように設定できます。

クラスの特定のメソッドをオーバーライドする必要がある場合もあります。たとえば、基本的な機能を提供し、一部のメソッドに **abstract** 修飾子を使用するクラスを作成できます。抽象メソッドの実装は、派生クラスの作成者が行います。

バージョン セーフ修飾子もオーバーライドを管理しますが、この修飾子は基本クラス側からではなく、派生クラス側からオーバーライドを管理します。バージョン セーフ修飾子は、オーバーライドしている基本クラスのメソッドに、継承の修飾子が指定されていない場合にだけ効果があります。

継承の修飾子を 2 つ組み合わせたり、継承の修飾子を **static** 修飾子と組み合わせたりすることはできません。

継承の修飾子	対象	説明
abstract	クラス、メソッド、またはプロパティ	メソッドまたはプロパティの場合は、メンバの実装がないことを示します。クラスの場合は、実装されていないメソッドが 1 つ以上あることを示します。抽象クラス、または抽象メンバを含むクラスは、**new** キーワードを使ってインスタンス化できませんが、基本クラスとして使用できます。

final	クラス、メソッド、またはプロパティ	拡張されないクラス、またはオーバーライドされないメソッドに使用します。**final** 修飾子を使用すると、派生クラスが重要な関数をオーバーライドできず、クラスの動作は変更されません。**final** 修飾子を指定したメソッドを隠ぺいまたはオーバーロードすることはできますが、オーバーライドすることはできません。

バージョン セーフ修飾子

バージョン セーフ修飾子は、基本クラスのメソッドをオーバーライドする派生クラスのメソッドを制御します。この修飾子を使用すると、作成するクラスが基本クラスのメソッドをオーバーライドするかどうかを管理できます。

既定では、派生クラスのメソッドは基本クラスのメソッドをオーバーライドします。ただし、基本クラスの定義に継承の修飾子を指定すると、オーバーライドを妨げることができます。バージョン セーフ修飾子を使用すると、特定のメソッドがオーバーライドされるかどうかを制御できます。

場合によっては、基本クラスのメソッドがオーバーライドされないようにする必要があります。たとえば、クラスを拡張して基本クラスのメソッドの動作を変更する場合があります。基本クラスのメソッドがオーバーライドされないようにする場合は、メソッドの宣言で **hide** 修飾子を使用します。

基本クラスの特定のメソッドをオーバーライドする場合もあります。たとえば、クラスを変更せずに、クラスのメソッドを変更する場合などです。クラスを拡張し、メソッドの宣言に **override** 修飾子を使用すると、新しいメソッドで基本クラスをオーバーライドできます。

バージョン セーフ修飾子の使用が適切かどうかは、基本クラスのメソッドの宣言で継承の修飾子が使用されているかどうかに依存します。**final** 修飾子を指定した基本クラスのメソッドは、オーバーライドされません。**abstract** 修飾子を指定した基本クラスのメソッドは、基本クラスの抽象メソッドを明示的に実装しない限り隠ぺいされません。

バージョン セーフ修飾子を 2 つ組み合わせたり、バージョン セーフ修飾子を **static** 修飾子と組み合わせたりすることはできません。バージョン セーフ モードで実行している場合、バージョン セーフ修飾子は、基本クラスのメソッドをオーバーライドする各メソッドに 1 つしか使用できません。

バージョン セーフ修飾子	対象	説明
hide	メソッドまたはプロパティ	基本クラスの同じ名前のメンバをオーバーライドしません。
override	メソッドまたはプロパティ	既定では、基本クラスの同じ名前のメンバをオーバーライドします。

expando 修飾子

expando 修飾子を使用すると、クラス ベースのオブジェクトが JScript オブジェクトと同様に動作します。メソッドとプロパティは、expando オブジェクトに動的に追加できます。詳細については、「プロトタイプ ベースのオブジェクト」を参照してください。

expando 修飾子は、他の修飾子と関係なく使用できます。

修飾子	対象	説明
expando	クラスまたはメソッド	クラスの場合、動的なプロパティ (expando) を格納および取得できる、既定のインデックス付きプロパティが与えられます。メソッドの場合、expando オブジェクトのコンストラクタであることを示します。

static 修飾子

static 修飾子は、クラスのメンバが、クラスのインスタンスではなくクラス自身に属していることを示します。したがって、クラスに固有のデータとメソッドは、特定のインスタンスに関連付けられません。

static 修飾子は、バージョン セーフ修飾子または継承の修飾子と組み合わせることができません。

修飾子	対象	説明
static	メソッド、プロパティ、フィールド、またはクラス	メソッドの場合、クラスのインスタンスを必要とせずに呼び出せることを示します。プロパティおよびフィールドの場合、すべてのインスタンスで 1 つのコピーが共有されることを表します。**static** 修飾子と、クラスを初期化するコードを表す **static** ステートメントを混同しないようにしてください。

参照

関連項目

class ステートメント
interface ステートメント
function ステートメント
function get ステートメント
function set ステートメント

var ステートメント
const ステートメント
static ステートメント
その他の技術情報
修飾子

JScript の演算子

JScript には、算術演算子、論理演算子、ビット処理演算子、代入演算子など、さまざまな演算子が用意されています。演算子は、単純な式を組み合わせてより複雑な式を形成します。

このセクションの内容

演算子の一覧

JScript の演算子を種類によってグループ分けして表に示します。

演算子の優先順位

演算子の一覧と、対応する優先順位およびその動作を表に示します。

ビット処理演算子による型の強制変換

ビット処理演算子のオペランドに関する変換規則について説明します。オペランドどうし、およびオペランドとビット処理演算子の間でバイナリ形式の互換性が保たれるように、オペランドの型変換が必要です。

関連するセクション

JScript の代入と等価比較

代入演算子、等値演算子、および厳密等価演算子の使用方法について説明します。

JScript における型の強制変換

型の強制変換の概念、使用方法、および制限について説明します。

演算子 (JScript)

JScript の演算子に関するトピックへのリンクを示します。

演算子の一覧

JScript で使用される演算子を次の表に示します。説明に記載されている算術演算子の名前は、各演算子の正しい構文と使用方法を説明するトピックにリンクされています。

算術演算子

説明	記号
加算	+
デクリメント	--
除算	/
インクリメント	++
剰余	%
乗算	*
減算	-
単項マイナス符号	-

すべての算術演算子は、数値データに対して演算を実行します。加算演算子は、両方のオペランドが文字列の場合、文字列を連結します。

論理演算子

説明	記号
等値	==
以上	>=
より大きい	>
厳密等価	===
In	in
非等値	!=
以下	<=
より小さい	<
論理 AND	&&
論理 NOT	!
論理 OR	||
厳密非等価	!==

論理演算子はブール値を返します。この値は、演算子に応じて、比較、テスト、または組み合わせの結果を表します。

ビット処理演算子

説明	記号
ビットごとの AND	&
ビットごとの左シフト	<<
ビットごとの NOT	~
ビットごとの OR	|
ビットごとの右シフト	>>
ビットごとの XOR	^
符号なしの右シフト	>>>

ビット処理演算子は、オペランドのバイナリ表現を操作します。2 つのオペランドに互換性がない場合は、適切な型に強制的に変換されます。詳細については、「ビット処理演算子による型の強制変換」を参照してください。

代入演算子

説明	記号
代入	=
加算代入	+=
ビットごとの AND 代入	&=
ビットごとの OR 代入	|=
ビットごとの XOR 代入	^=
除算代入	/=
左シフト代入	<<=
剰余代入	%=
乗算代入	*=
右シフト代入	>>=
減算代入	-=
符号なし右シフト代入	>>>=

代入演算子はすべて、左のオペランドに代入された値を返します。

その他の演算子

説明	記号
コンマ	,
条件 (三項)	?:

delete	delete
instanceof	instanceof
new	new
typeof	typeof
Void	void

参照

概念

演算子の優先順位

その他の技術情報

JScript の演算子

演算子の優先順位

演算子の優先順位は、式を評価するときにコンパイラが実行する演算の順序を制御する、JScript における規則セットです。優先順位の高い演算は、優先順位の低い演算より先に実行されます。たとえば、乗算は加算より先に実行されます。

優先順位表

次の表では、JScript の演算子を優先順位の高い順に上から並べています。

優先順位	評価順序	演算子	説明
15	左から右	., [], ()	フィールド アクセス、配列のインデックス付け、関数の呼び出し、および式のグループ化
14	右から左	++、--、-、~、!、delete、new、typeof、void	単項演算子、戻り値の型、オブジェクトの作成、未定義の値
13	左から右	*, /, %	乗算、除算、剰余
12	左から右	+, -	加算および文字列の連結、減算
11	左から右	<<, >>, >>>	ビット シフト
10	左から右	<、<=、>、>=、instanceof	小なり、以下、大なり、以上、instanceof
9	左から右	==, !=, ===, !==	等値、非等値、厳密等価、厳密非等価
8	左から右	&	ビットごとの AND
7	左から右	^	ビットごとの XOR
6	左から右	|	ビットごとの OR
5	左から右	&&	論理 AND
4	左から右	||	論理 OR
3	右から左	?:	条件
2	右から左	=, *OP*=	代入、複合代入
1	左から右	, (コンマ)	複数の評価

式でかっこを使用すると、演算子の優先順位と関係なく、演算の実行順序を変更できます。つまり、かっこの内側の演算は、かっこの外側の演算よりも先に実行されます。ただし、かっこの内側の演算子には通常の優先順位が適用されます。

次に例を示します。

```
z = 78 * (96 - 3 + 45)
```

この式には、=、*、()、-、および + の 5 つの演算子があります。演算の優先順位の規則に従うと、これらの演算子は ()、-、+、*、= の順に実行されます。

1. 最初に、かっこの内側の演算子が評価され、演算が実行されます。かっこ内には、加算演算子と減算演算子があります。どちらの演算子も優先順位が同じなので、左から右の順に演算が実行されます。まず 96 から 3 を引いて、結果は 93 になります。次に、93 に 45 を足して、結果は 139 になります。

2. 次に、かっこの外側の乗算が実行されます。78 に 139 が掛けられ、演算結果は 10764 になります。
3. 最後に代入が実行されます。z に 10764 が代入されます。

参照

概念

演算子の一覧

その他の技術情報

JScript の演算子

ビット処理演算子による型の強制変換

このバージョンの JScript のビット処理演算子は、以前のバージョンの JScript のビット処理演算子と完全な互換性を維持しています。また、JScript の演算子は新しい数値データ型に対しても使用できます。ビット処理演算子の動作は、データのバイナリ表現に依存します。したがって、演算子がデータの型を強制変換する方法を理解することが重要です。

ビット処理演算子には、事前バインディングされた変数、遅延バインディングされた変数、およびリテラル データの、3 種類の引数を渡すことができます。事前バインディングされた変数は、明示的な型の注釈を使って定義された変数です。遅延バインディングされた変数は、数値データを含む **Object** 型の変数です。

ビットごとの AND (&)、OR (|)、および XOR (^) 演算子

いずれかのオペランドが遅延バインディングされている場合、または両方のオペランドがリテラルの場合は、どちらのオペランドも **int** (**System.Int32**) に強制変換され、演算の戻り値は **int** になります。

両方のオペランドが事前バインディングされている場合、または一方のオペランドがリテラルで他方が事前バインディングされている場合は、さらに手順が必要です。どちらのオペランドも、2 つの条件で決定される型に強制変換されます。

- どちらのオペランドも整数でない場合は、両方とも **int** に強制変換されます。
- いずれか一方のオペランドだけが整数の場合、整数でないオペランドは、整数型と **int** のうち長い方の型に強制変換されます。
- いずれかのオペランドの方が長い場合、オペランドの変換先の型は、長い方のオペランドと同じ長さになります。
- どちらのオペランドも符号なしである場合、オペランドは符号なしの型に強制変換されます。それ以外の場合は、符号付きの型に変換されます。

次に、オペランドは適切な型に強制変換され、ビット処理演算子が実行されて結果が返されます。演算結果のデータ型は、強制変換されたオペランドのデータ型と同じになります。

整数リテラルをビット処理演算子および事前バインディングされた変数と使用する場合、リテラルは、**int**、**long**、**ulong**、または **double** として解釈されます。解釈は、数値を表すことができる最小の型によって決まります。リテラルの **decimal** 値は、**double** に強制変換されます。リテラルのデータ型は、上記の規則に従って、さらに強制変換されることがあります。

ビットごとの NOT 演算子 (~)

オペランドが遅延バインディングされている場合、浮動小数点数である場合、またはリテラルである場合、オペランドは **int** (**System.Int32**) に変換され、NOT 演算の戻り値は **int** になります。

オペランドが事前バインディングされた整数型である場合、NOT 演算の戻り値の型はオペランドの型と同じになります。

ビットごとの左シフト演算子 (<<) および右シフト演算子 (>>)

左オペランドが遅延バインディングされている場合、浮動小数点数である場合、またはリテラルである場合、オペランドは **int** (**System.Int32**) に強制変換されます。それ以外の場合、左オペランドは事前バインディングされた整数型であり、型の強制変換は実行されません。右オペランドは、常に整数型に強制変換されます。次に、強制変換された値に対してシフト演算が実行されます。戻り値の型は、左オペランドと同じか (事前バインディングされている場合)、または **int** 型になります。

符号なし右シフト演算子 (>>>)

左オペランドが遅延バインディングされている場合、浮動小数点数である場合、またはリテラルである場合、オペランドは **uint** (**System.UInt32**) に強制変換されます。それ以外の場合、左オペランドは事前バインディングされた整数型であり、符号なしの同じ型に強制変換されます。たとえば、**int** は **uint** に強制変換されます。右オペランドは、常に整数型に強制変換されます。次に、強制変換された値に対してシフト演算が実行されます。戻り値の型は、強制変換された左オペランドと同じか (事前バインディングされている場合)、または **uint** 型になります。

符号なし右シフト演算の結果は、常に、オーバーフローすることなく、戻り値の型の符号付きの型に格納できます。

参照

概念

演算子の優先順位
型変換
JScript における型の強制変換
数値データ

その他の技術情報

JScript の演算子

JScript の関数

JScript の関数は、アクションを実行したり、値を返したりできます。たとえば、現在の時刻を表示したり、その時刻を表す文字列を返したりできます。関数は、グローバル メソッドとも呼ばれます。

関数は、いくつかの演算を組み合わせて名前を付けたものです。関数によってコードは簡素化され、再利用できるようになります。一連のステートメントを記述してそれに名前を付けておくと、名前を呼び出して必要な情報を渡すことで、一連のステートメントをまとめて実行できます。

関数に情報を渡すには、渡す情報を関数名の後ろにかっこで囲んで指定します。関数に渡される情報は、引数またはパラメータと呼ばれます。関数には、引数を受け取らないものと、1 つ以上の引数を必要とするものがあります。用途に応じて引数の数が変わる関数もあります。

JScript では、あらかじめ言語に組み込まれている関数と独自に作成する関数の、2 種類の関数がサポートされています。

このセクションの内容

型の注釈

型の注釈の概念と、関数定義で型の注釈を使用してデータ型の入出力を制御する方法について説明します。

ユーザー定義の JScript 関数

JScript で新しい関数を定義して使用する方法について説明します。

再帰

再帰の概念と、再帰関数を作成する方法について説明します。

関連するセクション

JScript の演算子

算術演算子、論理演算子、ビット処理演算子、代入演算子、およびその他の演算子の一覧を示し、これらの演算子の効果的な使用方法について説明する情報へのリンクを示します。

JScript のデータ型

JScript のプリミティブ データ型、参照データ型、および .NET Framework データ型の使用方法について説明するトピックへのリンクを示します。

JScript における型の強制変換

型の強制変換の概念、使用方法、および制限について説明します。

function ステートメント

関数宣言の構文について説明します。

型の注釈

型の注釈は、関数の引数で必要な型、戻り値データで必要な型、または両方で必要な型を指定します。関数のパラメータに型の注釈を指定しない場合、パラメータは **Object** 型になります。同様に、関数の戻り値を指定しない場合は、コンパイラが適切な戻り値の型を判断します。

型の注釈の使用

関数のパラメータに型の注釈を使用すると、関数が処理できるデータ型だけを受け取ることが保証されます。関数の戻り値を明示的に宣言すると、関数が返すデータの型が明確になるため、コードが読みやすくなります。

次の例では、関数のパラメータと戻り値の型の両方に型の注釈を指定します。

```
// Declare a function that takes an int and returns a String.
function Ordinal(num : int) : String{
   switch(num % 10) {
   case 1: return num + "st";
   case 2: return num + "nd";
   case 3: return num + "rd";
   default: return num + "th";
   }
}

// Test the function.
print(Ordinal(42));
print(Ordinal(1));
```

このプログラムの出力は次のようになります。

```
42nd
1st
```

整数型に強制変換できない引数を `Ordinal` 関数に渡した場合、型の不一致を示すエラーが発生します。たとえば、`Ordinal(3.14159)` と指定するとエラーになります。

参照

関連項目

function ステートメント

その他の技術情報

JScript の関数

データ型 (JScript)

ユーザー定義の JScript 関数

JScript には多数の組み込み関数が用意されていますが、独自の関数も作成できます。関数の定義は、1 つの関数ステートメントと、JScript ステートメントのブロックで構成されます。

独自の関数の定義

次の例の `checkTriplet` 関数は、三角形の 3 つの辺の長さを引数として受け取ります。指定した数値がピタゴラスの三平方の定理 (直角三角形の斜辺の長さの 2 乗は、他の 2 つの辺の長さの 2 乗の和に等しい) を満たしているかどうかを調べることにより、その三角形が直角三角形かどうかを評価します。`checkTriplet` 関数は、直角三角形かどうかを評価するときに、他の 2 つの関数のいずれかを呼び出します。

浮動小数点数バージョンの評価では、評価変数として非常に小さい数値 (`epsilon`) を使用している点に注目してください。浮動小数点数の計算では、丸め誤差によって正しい結果が得られない場合があります。したがって、指定された 3 つの数値がすべて整数でない限り、ピタゴラスの三平方の定理を満たしているかどうかを直接調べることは、実際的ではありません。ただし、直接比較した方がより正確に直角三角形であることを確認できるため、この例のコードでは、まず直接比較することが適切かどうかを調べ、適切である場合は直接比較する方法を使用しています。

3 つの関数を定義するときに、型の注釈は使用されていません。このアプリケーションでは、`checkTriplet` 関数で整数と浮動小数点数の両方のデータ型を受け取るようにした方が便利です。

```
const epsilon = 0.00000000001; // Some very small number to test against.

// Type annotate the function parameters and return type.
function integerCheck(a : int, b : int, c : int) : boolean {
   // The test function for integers.
   // Return true if a Pythagorean triplet.
   return ( ((a*a) + (b*b)) == (c*c) );
} // End of the integer checking function.

function floatCheck(a : double, b : double, c : double) : boolean {
   // The test function for floating-point numbers.
   // delta should be zero for a Pythagorean triplet.
   var delta = Math.abs( ((a*a) + (b*b) - (c*c)) * 100 / (c*c));
   // Return true if a Pythagorean triplet (if delta is small enough).
   return (delta < epsilon);
} // End of the floating-poing check function.

// Type annotation is not used for parameters here. This allows
// the function to accept both integer and floating-point values
// without coercing either type.
function checkTriplet(a, b, c) : boolean {
   // The main triplet checker function.
   // First, move the longest side to position c.
   var d = 0; // Create a temporary variable for swapping values
   if (b > c) { // Swap b and c.
      d = c;
      c = b;
      b = d;
   }
   if (a > c) { // Swap a and c.
      d = c;
      c = a;
      a = d;
   }

   // Test all 3 values. Are they integers?
   if ((int(a) == a) && (int(b) == b) && (int(c) == c)) { // If so, use the precise check.
      return integerCheck(a, b, c);
   } else { // If not, get as close as is reasonably possible.
      return floatCheck(a, b, c);
   }
} // End of the triplet check function.

// Test the function with several triplets and print the results.
// Call with a Pythagorean triplet of integers.
```

```
print(checkTriplet(3,4,5));
// Call with a Pythagorean triplet of floating-point numbers.
print(checkTriplet(5.0,Math.sqrt(50.0),5.0));
// Call with three integers that do not form a Pythagorean triplet.
print(checkTriplet(5,5,5));
```

このプログラムの出力は次のようになります。

```
true
true
false
```

参照

関連項目

function ステートメント

その他の技術情報

JScript の関数

JScript のデータ型

再帰

再帰は、関数で関数自身を呼び出す、重要なプログラミング手法です。再帰法を使用した簡単な例として、階乗の計算があります。0 の階乗は明確に 1 であると定義されます。*n* を 0 より大きい整数とすると、*n* の階乗は 1 ～ *n* の範囲にある整数すべての積になります。

再帰の使用

次の段落は、階乗を求める関数の定義です。

"指定された数字が 0 より小さい場合は、計算しません。整数以外は計算しません。階乗を求める数字が 0 の場合、階乗は 1 です。階乗を求める数字が 1 以上の場合は、その数字にその数字より 1 だけ小さい値の階乗を掛けます。"

0 より大きい数値の階乗を計算する場合、他の数値の階乗を少なくとも 1 つ計算する必要があります。関数を現在の数値に対して実行する前に、1 つ小さい数値に対して自身を呼び出す必要があります。これが、再帰の一例です。

再帰と反復計算 (ループ) には強い関連があります。再帰と反復計算のどちらでも、関数は同じ結果を返すことができます。通常、どちらの処理方法が適しているかは、実行する計算によって決まります。プログラマは、その計算に対して最も無理のない方法、または最も適した方法を選択するだけです。

再帰は便利ですが、結果を返さず、終了しない再帰関数を簡単に作成してしまう可能性があります。このような再帰は "無限" ループになります。たとえば、階乗の計算の定義から最初の規則 (負の値に関する規則) を省略して関数を作成し、この関数を使用して負の値の階乗を求めようとしたとします。この場合、たとえば -24 の階乗を計算しようとすると、-25 の階乗の計算が必要となります。-25 の階乗を計算するためには、さらに -26 の階乗の計算が必要となります。このように処理が行われていくと、再帰呼び出しは終了しません。

再帰で発生する別の問題は、再帰関数が利用可能なリソース (システム メモリ、スタック領域など) をすべて使用してしまう可能性があることです。再帰関数が自身を (または、元の関数を呼び出す別の関数を) 呼び出すごとに、いくらかのリソースが使用されます。これらのリソースは再帰関数が終了するときに解放されますが、再帰のレベルが多すぎる関数は、利用可能なすべてのリソースを使用する可能性があります。利用可能なリソースがすべて使用されると、例外がスローされます。

このように、再帰関数は注意してデザインする必要があります。再帰が過度に (または無限に) なる可能性がある場合は、自身を呼び出した回数をカウントし、呼び出し回数を制限するように関数をデザインします。関数が自身を呼び出した回数がしきい値を超えたら、関数を自動的に終了します。最適となる最大の反復処理回数は、再帰関数によって異なります。

次に、階乗を求める関数の定義を JScript のコードで示します。型の注釈を指定して、関数が整数だけを受け取るようにします。無効な数値 (0 より小さい数) が渡された場合は、throw ステートメントがエラーを生成します。それ以外の場合は、再帰関数を使用して階乗を計算します。再帰関数は 2 つの引数を受け取ります。1 つは階乗の引数で、もう 1 つは現在の再帰レベルを追跡するためのカウンタです。カウンタが最大の再帰レベルに達しなかった場合は、元の数の階乗が返されます。

```
function factorialWork(aNumber : int, recursNumber : int ) : double {
   // recursNumber keeps track of the number of iterations so far.
   if (aNumber == 0) {  // If the number is 0, its factorial is 1.
      return 1.;
   } else {
      if(recursNumber > 100) {
         throw("Too many levels of recursion.");
      } else {  // Otherwise, recurse again.
         return (aNumber * factorialWork(aNumber - 1, recursNumber + 1));
      }
   }
}

function factorial(aNumber : int) : double {
   // Use type annotation to only accept numbers coercible to integers.
   // double is used for the return type to allow very large numbers to be returned.
   if(aNumber < 0) {
      throw("Cannot take the factorial of a negative number.");
   } else {  // Call the recursive function.
      return  factorialWork(aNumber, 0);
   }
}

// Call the factorial function for two values.
print(factorial(5));
print(factorial(80));
```

このプログラムの出力は次のようになります。

```
120
7.156945704626378e+118
```

参照

概念

型の注釈

その他の技術情報

JScript の関数

JScript における型の強制変換

JScript では、コンパイラで例外を生成することなく、異なる型の値に対して演算を実行できます。JScript コンパイラは、演算を実行する前に、一方のデータの型をもう一方のデータ型に自動的に変更します。他の言語には、強制変換を制御するより厳密な規則があります。

Coercion Details

強制変換が常に失敗することが検証されない限り、コンパイラはすべての強制変換を許可します。失敗する可能性のある強制変換に対しては、コンパイル時に警告が生成されます。強制変換に失敗すると、ほとんどの場合はランタイム エラーが生成されます。次に例を示します。

演算	結果
数値と文字列の加算	数値型が文字列型に変換されます。
ブール値と文字列の加算	ブール型が文字列型に変換されます。
数値とブール値の加算	ブール型が数値型に変換されます。

例を次に示します。

```
var x = 2000;      // A number.
var y = "Hello";   // A string.
x = x + y;         // the number is coerced into a string.
print(x);          // Outputs 2000Hello.
```

文字列を明示的に整数に変換するには、**parseInt** メソッドを使用できます。詳細については、「parseInt メソッド」を参照してください。文字列を明示的に数値に変換するには、**parseFloat** メソッドを使用できます。詳細については、「parseFloat メソッド」を参照してください。文字列は、比較の場合は自動的に相当する数値に変換されますが、加算 (連結) の場合は文字列のままで処理されます。

JScript も厳密に型指定される言語であるため、別の型変換機構を利用できます。新しい変換機構は、変換先の型名を、変換する式を引数として受け入れる関数のように使用します。この方法は、すべての JScript プリミティブ型、JScript 参照型、および .NET Framework 型で機能します。

たとえば、次のコードは整数値をブール値に変換します。

```
var i : int = 23;
var b : Boolean;
b = i;
b = Boolean(i);
```

`i` の値が 0 でないため、`b` は **true** になります。

新しい型変換機構は、多くのユーザー定義型でも機能します。ただし、ユーザー定義型に対する変換が動作しない場合もあります。類似していない型を変換する場合、JScript はユーザーの意図を正しく解釈できない可能性があります。変換元の型が複数の値で構成されている場合などがその例です。たとえば、次のコードでは 2 つのクラス (型) が作成されます。一方のクラスには、整数型の変数 `i` が 1 つだけ含まれます。もう一方のクラスには 3 つの変数 (`s`、`f`、および `d`) が含まれ、それぞれの型が異なっています。JScript は、最後のステートメントで、最初の型の変数を 2 番目の型に変換する方法を判断できません。

```
class myClass {
   var i : int = 42;
}
class yourClass {
   var s : String = "Hello";
   var f : float = 3.142;
   var d : Date = new Date();
}
// Define a variable of each user-defined type.
var mine : myClass = new myClass();
var yours : yourClass;

// This fails because there is no obvious way to convert
// from myClass to yourClass
```

```
  yours = yourClass(mine);
```

参照

概念

型変換

ビット処理演算子による型の強制変換

その他の技術情報

JScript の関数

JScript の条件構造

通常、JScript のステートメントはスクリプトに記述されている順に実行されます。これはシーケンシャル実行と呼ばれ、プログラム フローの既定の方法です。

シーケンシャル実行の代わりに、処理される条件の結果に応じて、プログラム フローがスクリプトの別の部分に移動される場合があります。つまり、シーケンスに従って次のステートメントを実行する代わりに、他のステートメントが実行されます。また、"反復処理" では、同じ一連のステートメントを繰り返し実行します。多くの場合、反復処理ではループが使用されます。

このセクションの内容

条件付きステートメント

条件付きステートメントの概念と、条件付きステートメントとして使用される一般的な式について説明します。

制御構造

JScript に用意されている 2 種類の制御構造について説明します。制御構造には、選択制御構造と反復制御構造があり、両方を組み合わせて使用する場合もあります。

条件付きステートメントの使用方法

if ステートメントと **do**...**while** ループの例を示して、条件付きステートメントの使用方法について説明します。

条件演算子

条件演算子の使用方法と、条件演算子と **if**...**else** ステートメントの関係について説明します。

JScript のループ

JScript で使用されるループの概念を紹介し、JScript コードでのループ構造の使用方法について説明する情報へのリンクを示します。

関連するセクション

JScript のデータ型

JScript のプリミティブ データ型、参照データ型、および .NET Framework データ型の使用方法について説明するトピックへのリンクを示します。

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

条件付きステートメント

JScript コードの命令は、既定ではシーケンシャルに実行されます。ただし、論理的なシーケンスを変更し、特定の条件に応じてコードの別の場所に制御を移すと便利な場合があります。制御構造は、条件付きステートメントの結果が true と false のどちらであるかに応じて、プログラムの制御を 2 つの場所のどちらかに移します。条件付きステートメントには、ブール型に変換できる式を使用できます。ここでは一般的な条件ステートメントを説明します。

等値演算子と厳密等価演算子

等値演算子 (==) を条件付きステートメントに使用すると、2 つの引数が同じ値かどうかを確認できます。必要な場合は型変換が実行されます。厳密等価演算子 (===) は、2 つの式の値と型の両方を比較し、2 つのオペランドの値とデータ型が同じである場合だけ **true** を返します。厳密等価演算子は、数値データ型を区別しません。

次の JScript コードでは、等値演算子とそれを使用する **if** ステートメントを組み合わせています。詳細については、「制御構造」を参照してください。

```
function is2000(x) : String {
   // Check if the value of x can be converted to 2000.
   if (x == 2000) {
      // Check is the value of x is strictly equal to 2000.
      if(x === 2000)
         print("The argument is number 2000.");
      else
         print("The argument can be converted to 2000.");
   } else {
      print("The argument is not 2000.");
   }
}
// Check several values to see if they are 2000.
print("Check the number 2000.");
is2000(2000);
print("Check the string \"2000\".");
is2000("2000")
print("Check the number 2001.");
is2000(2001);
```

このコードの出力は次のようになります。

```
Check the number 2000.
The argument is number 2000.
Check the string "2000".
The argument can be converted to 2000.
Check the number 2001.
The argument is not 2000.
```

非等値演算子と厳密非等価演算子

非等値演算子 (**!=**) は、等値演算子の反対の結果を返します。オペランドの値が同じである場合は **false** を返し、それ以外の場合は **true** を返します。同様に、厳密非等価演算子 (**!==**) は厳密等価演算子の反対の結果を返します。

次の JScript コードのサンプルでは、**while** ループの制御に非等値演算子が使用されています。詳細については、「制御構造」を参照してください。

```
var counter = 1;
// Loop over the print statement while counter is not equal to 5.
while (counter != 5) {
   print(counter++);
}
```

このコードの出力は次のようになります。

```
1
```

```
2
3
4
```

比較演算子

等値演算子および非等値演算子は、データが特定の値である場合は有用です。ただし、値が特定の範囲内にあるかどうかをチェックすることが必要な場合もあります。このような場合は、関係演算子の、小なり (<)、大なり (>)、以下 (<=)、または以上 (>=) を使用できます。

```
if(tempInCelsius < 0)
   print("Water is frozen.")
else if(tempInCelsius > 100)
   print("Water is vapor.");
else
   print("Water is liquid.);
```

ショート サーキット

複数の条件を組み合わせて評価する場合、その中に結果を予測できる条件が含まれるときは、ショート サーキット評価と呼ばれる機能を使用できます。これにより、スクリプトの実行が高速化し、エラーを発生させる可能性のある副作用を避けることができます。JScript で論理式を評価する場合は、結果を得るために必要な最低限の部分式だけが評価されます。

論理 AND 演算子 (**&&**) は、渡された左側の式を最初に評価します。左側の式が **false** に変換される場合、右側の式の値にかかわらず、論理 AND 演算子の結果は **false** になります。したがって、右側の式は評価されません。

たとえば `((x == 123) && (y == 42))` という式の場合、まず `x` が 123 に等しいかどうかが評価されます。x が 123 でない場合、`y` は評価されず、**false** が返されます。

同様に、論理 OR 演算子 (**||**) は、最初に左側の式を評価します。左側の式が **true** に変換される場合、右側の式は評価されません。

ショート サーキットを使用すると、評価する条件に関数の呼び出しや複雑な式が含まれている場合などに便利です。スクリプトを最も効率よく実行するために、論理 OR 演算子では左側の式に **true** になりそうな条件を指定します。論理 AND 演算子の場合は、**false** になりそうな条件から指定してください。

この方法でスクリプトを作成している次のコード例では、**runfirst()** 関数の戻り値が **false** になる場合は、**runsecond()** 関数が実行されません。

```
if ((runfirst() == 0) || (runsecond() == 0)) {
   // some code
}
```

また、次の例のような場合にも便利です。

```
if ((x == 0) && (y/x == 5)) {
   // some code
}
```

その他の式

ブール値に変換できる式は、条件ステートメントとして使用できます。たとえば、次のような式を使用した場合を考えます。

```
if (x = y + z) // This may not do what you expect - see below!
```

上のコード例は、単一の等号 (代入) が使用されているため、`x` が `y + z` に等しいかどうかを調べているのではありません。上記のコードでは、`y + z` の値を変数 `x` に代入し、その結果 (`x` の値) を **true** 値に変換できるかどうかを調べています。`x` が `y + z` に等しいかどうかを調べるには、次のコードを使用します。

```
if (x == y + z) // This is different from the code above!
```

参照

概念

Boolean データ

その他の技術情報

JScript の条件構造
JScript のデータ型
JScript リファレンス
演算子 (JScript)

制御構造

すべての制御構造 (**switch** ステートメントを除く) において、プログラム制御の移動は、真理値ステートメント (戻り値がブール値の **true** または **false**) で表される判断の結果に基づきます。式を作成した後、結果が true かどうかを評価します。プログラムの制御構造には、主に 2 つの種類があります。

選択制御構造

選択制御構造を使用すると、プログラムに分岐点を作成してプログラム フローの流れを変更できます。JScript では、次の 4 つの選択構造を使用できます。

- 単一選択構造 (**if**)
- 分岐選択構造 (**if...else**)
- 複数選択構造 (**switch**)
- インライン条件演算子 (**?:**)

反復制御構造

反復構造では、ある条件が true である間、特定のアクションを繰り返します。制御ステートメントの条件が満たされると (通常は指定した反復処理の後)、制御は反復構造を抜けて次のステートメントに移ります。JScript では、次の 4 つの反復構造を使用できます。

- ループの最初に式が評価される (**while**)
- ループの最後に式が評価される (**do...while**)
- オブジェクトのプロパティまたは配列の要素を処理する (**for...in**)
- カウンタを使って反復を制御する (**for**)

複合制御構造

複雑なスクリプトでは、選択制御構造と反復制御構造を組み合わせて使用します。

例外処理でもプログラム フローを制御できますが、ここでは説明しません。詳細については、「try...catch...finally ステートメント」を参照してください。

参照

その他の技術情報

JScript の条件構造
JScript リファレンス

条件付きステートメントの使用方法

JScript では、条件付きステートメントとして **if** および **if...else** がサポートされています。**if** ステートメントは条件を評価します。条件が true に評価されると、関連する JScript コードが実行されます。**if...else** ステートメントは条件を評価し、条件付きステートメントの結果に応じて、2 つのコード ブロックのどちらかを実行します。単純な **if** ステートメントでは全体を 1 行に収めて記述することもできますが、通常は、**if** ステートメントや **if...else** ステートメントが複数行にわたって記述されます。

条件付きステートメントの例

if ステートメント、および **if...else** ステートメントを使用したコードの例を次に示します。最初の例は、ブール値の評価の最も単純な形式です。かっこで囲まれた部分の評価結果が **true** である (または **true** に変換できる) 場合にだけ、**if** の後のステートメントまたはステートメント ブロックが実行されます。

次の例では、`newUser` の値が true の場合に、`registerUser` 関数が呼び出されます。

```
if (newUser)
   registerUser();
```

次の例では、両方の条件が true でないと、評価が失敗します。

```
if (rind.color == "deep yellow " && rind.texture == "wrinkled") {
   theResponse = ("Is it a Crenshaw melon?");
}
```

次の例では、変数 `quit` が **true** になるまで、**do...while** ループ本体のコードが実行されます。

```
var quit;
do {
   // ...
   quit = getResponse()
}
while (!quit)
```

参照

関連項目

if...else ステートメント

その他の技術情報

JScript の条件構造

JScript リファレンス

条件演算子

JScript では、暗黙的な条件形式である条件演算子もサポートされます。条件演算子は 3 つのオペランドを受け取ります。疑問符で最初の 2 つのオペランドを区切り、2 番目と 3 番目のオペランドはコロンで区切ります。最初のオペランドは条件式です。2 番目のオペランドは、条件式が true に評価された場合に実行されるステートメントです。3 番目のオペランドは、条件が false の場合に実行されます。詳細については、「条件 (三項) 演算子 (?:)」を参照してください。条件演算子は **if...else** ステートメントに似ています。

条件演算子の使用

次の例では、24 時間制の時刻が午前 ("AM") か午後 ("PM") かを条件演算子で判断します。

```
var hours : String = (the24Hour >= 12) ? " PM" : " AM";
```

通常、実行するステートメントを選択するときには、**if ... then ... else** 構造が適切です。また、2 つの式のいずれかを選択するときには、条件演算子 (?) が適切です。条件演算子を使って、3 つ以上の選択肢を指定したり、ステートメントのブロックを実行したりしないでください。このような場合は、**if...then...else** 構造を使用します。

参照

その他の技術情報

JScript の条件構造
JScript リファレンス

JScript のループ

JScript には、1 つのステートメント、または複数のステートメントで構成されるブロックを繰り返し実行する方法がいくつか用意されています。一般に、繰り返し実行は "ループ" または "反復処理" と呼ばれます。反復処理では、ループを 1 回実行します。通常、ループの制御は、ループ内が実行されるたびに値が変化する変数を評価して行います。JScript では、**for**、**for...in**、**while**、および **do...while** の 4 種類のループがサポートされています。

このセクションの内容

for ループ

for ループの動作について説明し、実際的な例を示します。

for...in ループ

for...**in** ループの概念と JScript での使用方法について説明します。

while ループ

2 種類の **while** ループについて説明します。また、**for** ループとの違いについても説明します。

break ステートメントと continue ステートメント

break ステートメントと **continue** ステートメントを使用してループの動作をオーバーライドする方法について説明します。

関連するセクション

JScript の条件構造

JScript がプログラム フローを処理する方法について説明し、プログラムの実行フローの制御方法について説明する情報へのリンクを示します。

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

for ループ

for ステートメントは、カウンタ変数、評価する条件、およびカウンタを更新する処理を指定します。ループが反復処理される前に、条件が評価されます。評価が成功した場合は、ループ内のコードが実行されます。評価が成功しなかった場合、ループ内のコードは実行されず、ループ後の 1 行目のコードの処理が続けられます。カウンタ変数は、ループの実行後、次の反復処理が開始される前に更新されます。

for ループの使用

ループの条件が満たされない場合、ループは一度も実行されません。条件が常に満たされると、無限ループになります。場合によっては、ループが実行されない方が望ましいこともありますが、無限ループが望ましい場合はほとんどありません。したがって、ループ条件を記述するときには注意が必要です。次の例では、**for** ループを使用して、配列の要素を 1 つ前の要素までの合計値で初期化しています。

```
var sum = new Array(10); // Creates an array with 10 elements
sum[0] = 0;              // Define the first element of the array.
var iCount;

// Counts from 0 through one less than the array length.
for(iCount = 0; iCount < sum.length; iCount++) {
   // Skip the assignment if iCount is 0, which avoids
   // the error of reading the -1 element of the array.
   if(iCount!=0)
      // Add the iCount to the previous array element,
      // and assign to the current array element.
      sum[iCount] = sum[iCount-1] + iCount;
   // Print the current array element.
   print(iCount + ": " + sum[iCount]);
}
```

このプログラムの出力は次のようになります。

```
0: 0
1: 1
2: 3
3: 6
4: 10
5: 15
6: 21
7: 28
8: 36
9: 45
```

次の例には、2 つのループがあります。最初のループのコード ブロックは実行されません。2 番目のループは無限ループになります。

```
var iCount;
var sum = 0;
for(iCount = 0; iCount > 10; iCount++) {
   // The code in this block is never executed, since iCount is
   // initially less than 10, but the condition checks if iCount
   // is greater than 10.
   sum += iCount;
}
// This is an infinite loop, since iCount is always greater than 0.
for(iCount = 0; iCount >= 0; iCount++) {
   sum += iCount;
}
```

参照

関連項目

for ステートメント

その他の技術情報

JScript のループ

JScript の条件構造

for...in ループ

JScript には、オブジェクト内のすべてのユーザー定義プロパティ、配列内のすべての要素、またはコレクション内のすべての要素を 1 つずつ処理するための特殊なループが用意されています。**for...in** ループのループ カウンタは、数値ではなく文字列またはオブジェクトです。ループ カウンタには、現在のプロパティの名前、現在の配列要素のインデックス、または現在のコレクション要素が含まれます。

for...in ループの使用

for...in ループの使用例を次に示します。

```
// Create an object with some properties.
var prop, myObject = new Object();
myObject.name = "James";
myObject.age = 22;
myObject.phone = "555 1234";
// Loop through all the properties in the object.
for (prop in myObject){
   print("myObject." + prop + " equals " + myObject[prop]);
}
```

このプログラムの出力は次のようになります。

```
myObject.name equals James
myObject.age equals 22
myObject.phone equals 555 1234
```

JScript の **for...in** ループの動作により、コレクション要素の反復処理に **Enumerator** オブジェクトを使用する必要がなくなりました。

参照

関連項目

for...in ステートメント

その他の技術情報

JScript のループ
JScript の条件構造
JScript リファレンス

while ループ

while ループは、**for** ループに似ていて、ステートメント ブロックを繰り返し実行します。ただし、while ループには組み込みのカウンタ変数がなく、式も更新しません。ステートメントやステートメント ブロックの繰り返し処理を制御する場合に、"あるコードを n 回実行する" というような単純な規則よりも複雑な指定をする必要があるときは、**while** ループを使用します。

while ループの使用

次のコードは、**while** ステートメントの使用例です。

```
var x = 1;
while (x < 100) {
   print(x);
   x *= 2;
}
```

このプログラムの出力は次のようになります。

```
1
2
4
8
16
32
64
```

メモ :

while ループには、明示的に組み込まれたカウンタ変数がありません。このため、他の種類のループと比べ、無限ループに陥る可能性が高くなります。さらに、**while** ループは、ループを制御する条件がいつどこで更新されるのかを見つけにくいため、条件が更新されないループを記述する可能性もあります。このような理由から、**while** ループをデザインするときには注意が必要です。

JScrip には、**while** ループに似た **do...while** ループもあります。**do...while** ループでは、条件がループの先頭ではなく終わりで評価されるため、ループが少なくとも 1 回は実行されます。上のコード例は、次のように記述することもできます。

```
var x = 1;
do {
   print(x);
   x *= 2;
}
while (x < 100)
```

このプログラムの出力は、上の例の出力と同じです。

参照

関連項目

while ステートメント
do...while ステートメント

その他の技術情報

JScript のループ
JScript の条件構造
JScript リファレンス

break ステートメントと continue ステートメント

JScript の **break** ステートメントは、ある条件を満たしたときにループの実行を終了します。**break** は、**switch** ブロックを終了するときにも使用します。**continue** ステートメントを使用すると、コード ブロックの残りの部分を実行せずに、次の反復処理に移ることができます。ループが **for** または **for...in** の場合は、カウンタ変数が更新されます。

break ステートメントと continue ステートメントの使用

ループを制御する **break** ステートメントと **continue** ステートメントの使用例を次に示します。

```
for(var i = 0;i <=10 ;i++) {
   if (i > 7) {
      print("i is greater than 7.");
      break; // Break out of the for loop.
   }
   else {
      print("i = " + i);
      continue; // Start the next iteration of the loop.
      print("This never gets printed.");
   }
}
```

このプログラムの出力は次のようになります。

```
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i is greater than 7.
```

参照

関連項目

break ステートメント
continue ステートメント

その他の技術情報

JScript のループ
JScript の条件構造
JScript リファレンス

JScript の予約語

JScript には多くの予約語があり、これらは JScript の言語構文において特別な意味を持ちます。したがって、これらの予約語を関数、変数、または定数の名前として使うことはお勧めできません。予約語には 3 つのグループがあります。

保護された予約語

保護された予約語は、識別子として使用できません。保護された予約語を識別子として使用すると、スクリプトを読み込んだときにコンパイル エラーが発生します。

break	case	catch	class	const
continue	debugger	default	delete	do
else	export	extends	false	finally
for	function	if	import	in
instanceof	new	null	protected	return
super	switch	this	throw	true
try	typeof	var	while	with

メモ：

"export" は保護された予約語ですが、実際には使用されません。

新しい予約語

JScript には、新しい予約語の一覧も用意されています。保護された予約語と同様、これらのキーワードは現在のバージョンの JScript において特別な意味を持ちます。下位互換性を維持するために、新しい予約語は識別子として利用できます。新しい予約語を識別子として使用すると、スクリプト内ではキーワードとしての意味を失います。新しい予約語を識別子として使用すると混乱する可能性があるため、識別子としては使用しないことをお勧めします。

abstract	boolean	byte	char	decimal
double	enum	final	float	get
implements	int	interface	internal	long
package	private	protected	public	sbyte
set	short	static	uint	ulong
ushort	void			

将来的に使用される予約語

JScript には将来的に使用される予約語も用意されています。これらの予約語は、JScript の今後のバージョンでキーワードとして使用されることになっています。新しい予約語と同様、これらの予約語は現在のバージョンの JScript では識別子として使用できます。ただし、これらの予約語の使用を避けておくと、JScript の将来のバージョンで、スクリプトのアップグレードと新しい機能の利用が簡単になります。

識別子を選択する場合には、JScript の組み込みオブジェクト名や関数名として既に使用されている語を使わないように十分に注意してください。たとえば、**String** や **parseInt** などは選択できません。

assert	ensure	event	goto	invariant
namespace	native	require	synchronized	throws
transient	use	volatile		

参照

その他の技術情報

JScript リファレンス
JScript 言語の紹介

JScript のセキュリティに関する考慮事項

どの言語においても、安全なコードを作成することが重要な課題となります。JScript では、推奨される手順の使用を開発者に強制しないため、気付かないうちに安全でない方法で言語を使用してしまう可能性のある領域がいくつかあります。JScript は、セキュリティの確保も 1 つの目標としてデザインされていますが、最大の目標は、便利なアプリケーションを迅速に開発できるようにすることです。場合によっては、この 2 つの目標が相反することがあります。

セキュリティに影響を与える可能性のある、いくつかの主な問題を以下に示します。これらの問題を認識しておくことによって、セキュリティ上の危険を回避できます。**eval** メソッドを除くこれらの問題は、.NET Framework の新機能が導入されたために考慮することが必要になりました。

eval メソッド

JScript で最も誤用されやすい機能は **eval** メソッドです。このメソッドでは、JScript のソース コードを動的に実行できます。**eval** メソッドを使用する JScript アプリケーションは、プログラムから渡されるどのようなコードでも実行できるため、**eval** メソッドのすべての呼び出しにセキュリティ上の危険性があります。どのようなコードでも実行する柔軟性がアプリケーションで必要とされない限り、アプリケーションが **eval** メソッドに渡すコードを明示的に記述することを検討してください。

eval メソッドの完全な柔軟性を必要とするアプリケーションのセキュリティを向上させるために、**eval** に渡されるコードは、既定では制限付きコンテキスト内で実行されます。制限付きのセキュリティ コンテキストにより、ファイル システム、ネットワーク、ユーザー インターフェイスなどのすべてのシステム リソースへのアクセスを禁止できます。これらのリソースにアクセスしようとすると、セキュリティ例外が生成されます。ただし、**eval** メソッドで実行されるコードは、ローカル変数とグローバル変数を変更することはできます。詳細については、「eval メソッド」を参照してください。

以前のバージョンの JScript で記述されたコードでは、呼び出し元のコードと同じセキュリティ コンテキストでコードを実行するために **eval** メソッドが必要な場合があります。この動作を有効にするために、省略可能な 2 番目のパラメータとして文字列 "unsafe" を **eval** メソッドに渡すことができます。"unsafe" モードでは、コード文字列が呼び出し元のコードと同じアクセス許可で実行されるため、既知のソースから取得したコード文字列だけを実行する必要があります。

セキュリティ属性

.NET Framework のセキュリティ属性を使用して、JScript の既定のセキュリティ設定を明示的にオーバーライドできます。ただし、この操作を確実に理解していない限り、既定のセキュリティ設定は変更しないでください。特に、**AllowPartiallyTrustedCallers** 属性 (APTCA) カスタム属性の使用は避けてください。これは、通常、信頼できない呼び出し元は JScript コードを安全に呼び出すことができないためです。APTCA を使用して信頼できるアセンブリを作成し、そのアセンブリがアプリケーションによって読み込まれた場合は、部分的に信頼されている呼び出し元から、アプリケーションの完全に信頼できるアセンブリにアクセスできてしまいます。詳細については、「安全なコーディングのガイドライン」を参照してください。

部分的に信頼されるコードとホストされる JScript コード

JScript をホストするエンジンでは、呼び出されたすべてのコードによって、グローバル変数、ローカル変数、オブジェクトのプロトタイプ チェインなど、エンジンの一部を変更できます。また、関数は、渡される expando オブジェクトの expando プロパティまたはメソッドを変更できます。このため、JScript アプリケーションが、部分的に信頼されるコードを呼び出す場合や、他のコードで記述されたアプリケーション (Visual Studio for Applications [VSA] ホストからなど) で実行される場合は、アプリケーションの動作が変更される可能性があります。

上記の問題点を考慮すると、アプリケーションまたは **AppDomain** クラスのインスタンス内の JScript コードは、アプリケーション内の他のコードと同等以下の信頼レベルで実行する必要があります。他のコードよりも高い信頼レベルで実行される場合は、他のコードが JScript クラスのエンジンを操作し、それによってエンジンがデータを変更し、アプリケーション内の他のコードに影響を与える可能性があります。詳細については、「_AppDomain」を参照してください。

アセンブリ アクセス

JScript は、厳密な名前とテキストの簡易名の両方を使用してアセンブリを参照できます。厳密な名前参照には、アセンブリのバージョン情報と、アセンブリの整合性および ID を確認する暗号署名が含まれています。アセンブリを参照する場合は簡易名を使用する方が簡単ですが、厳密な名前を使用すると、機能の異なる他のアセンブリが同じ簡易名を持つ場合にコードが保護されます。詳細については、「方法 : 厳密な名前のアセンブリを参照する」を参照してください。

スレッド処理

JScript ランタイムは、スレッド セーフ機能を持つようにはデザインされていません。このため、マルチスレッド JScript コードは、予期しない動作をする場合があります。JScript でアセンブリを開発する場合は、マルチスレッド コンテキストでそのアセンブリが使用される可能性があることに注意してください。**Mutex** クラスなど、**System.Threading** 名前空間にあるクラスを使用して、アセンブリ内の JScript コードが適切な同期をとって実行されるようにする必要があります。

適切な同期コードを記述することはどの言語でも難しいため、必要な同期コードの実装方法をよく理解していない限り、JScript で汎用アセンブリを記述しないでください。詳細については、「System.Threading」を参照してください。

メモ：

ASP.NET では生成するすべてのスレッドの同期を管理するため、JScript で作成された ASP.NET アプリケーションの同期コードを記述する必要はありません。ただし、JScript で作成された Web コントロールはアセンブリのように動作するため、これらのコントロールには同期コードが含まれている必要があります。

ランタイム エラー

JScript は、柔軟に型指定された言語であるため、Visual Basic や Visual C# などの他の一部の言語よりも型の不一致が許容されます。型の不一致によってアプリケーションでランタイム エラーが発生する場合があるため、コードの開発時には、発生する可能性のある型の不一致を検出することが重要です。コマンド ライン コンパイラで **/warnaserror** フラグを使用するか、ASP.NET ページの **@ Page** ディレクティブの **warninglevel** 属性を使用できます。詳細については、「/warnaserror」および「@ Page」を参照してください。

互換モード

互換モードで **/fast-** オプションを使用してコンパイルされたアセンブリは、既定の高速モードでコンパイルされたアセンブリよりも暗号化の度合いが低くなります。**/fast-** オプションを使用すると、既定では使用できない言語機能が有効になります。JScript Version 5.6 以前のバージョン用に記述されたスクリプトとの互換性を実現するためには、これらの機能が必要になります。たとえば、互換モードでは、**String** オブジェクトなどの組み込みオブジェクトに expando プロパティを動的に追加できます。

互換モードは、JScript のレガシ コードからスタンドアロン実行可能ファイルを作成する場合に役立ちます。新しい実行可能ファイルまたはライブラリを開発する場合は、既定のモードを使用してください。既定のモードでは、安全なアプリケーションを作成できるだけでなく、パフォーマンスや他のアセンブリとの互換性も向上します。詳細については、「/fast」を参照してください。

参照

概念

前のバージョンの JScript で作成されたアプリケーションのアップグレード

その他の技術情報

ネイティブ コードと .NET Framework コードのセキュリティ

言語リファレンス

JScript 言語の要素が、アプリケーションおよびスクリプトを開発するための基礎を形成します。

このセクションの内容

関連するセクション

.NET Framework リファレンス

.NET Framework クラス ライブラリおよびその他の基本要素の構文と構造を説明するトピックへのリンクを示します。

データ型 (JScript)

データ型は、変数、定数、または関数に指定できる値の型を指定します。変数、定数、および関数の型の注釈を指定して適切なデータ型を限定することで、プログラミング エラーを減らすことができます。また、型の注釈を指定することで、より高速で効率的なコードが生成されます。

このセクションの内容

関連するセクション

JScript のデータ型

JScript のプリミティブ データ型、参照データ型、および .NET Framework データ型の使用方法について説明するトピックへのリンクを示します。

データ型の概要

JScript でサポートされる値データ型と参照データ型、それらのデータ型に対応する .NET Framework データ型、メモリの記憶領域のサイズ、および値の範囲を表に示します。

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

boolean 型 (JScript)

boolean 型の値は true または false のいずれかであり、型に true と false のどちらのキーワードが代入されたかによって決まります。

対応する .NET Framework データ型は、**System.Boolean** です。**Boolean** 型と **boolean** 型は同じです。

解説

boolean 型のプロパティとメソッドは、**System.Boolean** のプロパティおよびメソッドと同じです。

JScript では、**Boolean** オブジェクトも定義されます。**boolean** 型と **Boolean** オブジェクトは相互運用されます。したがって、**Boolean** オブジェクトは **boolean** 型のメソッドとプロパティを呼び出すことができ、**boolean** 型は **Boolean** オブジェクトのメソッドとプロパティを呼び出すことができます。詳細については、「Boolean オブジェクトのプロパティとメソッド」を参照してください。また、**Boolean** オブジェクトは **boolean** 型を受け取る関数で使用でき、その逆も可能です。

通常は、**Boolean** オブジェクトの代わりに **boolean** 型を使用してください。

プロパティおよびメソッド

AllMembers.T:System.Boolean

必要条件

バージョン .NET

参照

関連項目

Boolean オブジェクト
Boolean Structure
true リテラル
false リテラル

概念

データ型の概要

その他の技術情報

データ型 (JScript)

byte 型 (JScript)

byte 型は、符号なしの 1 バイトとして格納されます。

byte 型は、0 ～ 255 の範囲の整数を表現できます。

対応する .NET Framework データ型は、Byte です。**byte** 型のプロパティとメソッドは、**Byte** のプロパティおよびメソッドと同じです。

プロパティおよびメソッド

AllMembers.T:System.Byte

必要条件

バージョン .NET

参照

概念

データ型の概要

その他の技術情報

データ型 (JScript)

char 型 (JScript)

char 型は、2 バイトの Unicode 文字として格納されます。

char 型は、65,536 種類の Unicode 文字を表現できます。

対応する .NET Framework データ型は、Char です。**char** 型のプロパティとメソッドは、**Char** のプロパティおよびメソッドと同じです。

プロパティおよびメソッド

AllMembers.T:System.Char

必要条件

バージョン .NET

参照

関連項目

String 型 (JScript)

概念

データ型の概要

その他の技術情報

データ型 (JScript)

decimal 型 (JScript)

decimal 型は、12 バイトの整数部、1 ビットの符号、および累乗の指数として格納されます。

decimal 型は、大きくて精度の高い 10 進数を正確に表現できます。**decimal** 型には、精度を失うことなく、有効桁数が 28 の最大 1028 (正または負) の数値を格納できます。この型は、丸め誤差を避ける必要のあるアプリケーション (経理アプリケーションなど) で使用すると便利です。

対応する .NET Framework データ型は、Decimal です。**decimal** 型のプロパティとメソッドは、**Decimal** のプロパティおよびメソッドと同じです。

プロパティおよびメソッド

AllMembers.T:System.Decimal

必要条件

バージョン .NET

参照
概念
データ型の概要
その他の技術情報
データ型 (JScript)

double 型 (JScript)

double 型は、8 バイトの倍精度浮動小数点数として格納されます。倍精度の 64 ビット IEEE 754 値を表します。

double 型は、最大で 10308 (正または負)、最小で 10-323 を 15 桁の精度で表すことができます。また、**double** 型では、非数 (**NaN**)、正または負の無限大、および正または負の 0 を表すことができます。

大きい数値が必要であっても厳密さは要求されないアプリケーションには、この型が便利です。正確な数値が必要な場合は、**Decimal** 型を使用します。

対応する .NET Framework データ型は、Double です。**double** 型は、**Number** 型と同じです。

解説

double 型のプロパティとメソッドは、**System.Double** のプロパティおよびメソッドと同じです。

JScript では、**Number** オブジェクトが定義されます。**double** 型と **Number** オブジェクトは相互運用されます。このため、**Number** オブジェクトは **double** 型のメソッドやプロパティを呼び出すことができ、**double** 型は **Number** オブジェクトのメソッドやプロパティを呼び出すことができます。詳細については、「Number オブジェクトのプロパティとメソッド」を参照してください。また、**Number** オブジェクトは **double** 型を受け取る関数で使用でき、その逆も可能です。

通常は、**double** オブジェクトの代わりに **Number** 型を使用してください。

プロパティおよびメソッド

AllMembers.T:System.Double

必要条件

バージョン .NET

参照

関連項目

Number 型
decimal 型 (JScript)

概念

データ型の概要

その他の技術情報

データ型 (JScript)

float 型

float 型は、4 バイトの単精度浮動小数点数として格納されます。単精度の 32 ビット IEEE 754 値を表します。

float 型は、最大で 1038 (正または負)、最小で 10-44 を 7 桁の精度で表すことができます。また、**float** 型では、非数 (**NaN**)、正または負の無限大、および正または負の 0 を表すことができます。

大きい数値が必要であっても厳密さは要求されないアプリケーションには、この型が便利です。正確な数値が必要な場合は、**Decimal** 型を使用します。

対応する .NET Framework データ型は、Single です。**float** 型のプロパティとメソッドは、**Single** のプロパティおよびメソッドと同じです。

プロパティおよびメソッド

AllMembers.T:System.Single

必要条件

バージョン .NET

参照

関連項目

decimal 型 (JScript)

概念

データ型の概要

その他の技術情報

データ型 (JScript)

int 型

int 型は、4 バイトの整数として格納されます。

int 型は、-2,147,483,648 ～ 2,147,483,647 の範囲の整数を表現できます。

対応する .NET Framework データ型は、Int32 です。**int** 型のプロパティとメソッドは、**Int32** のプロパティおよびメソッドと同じです。

プロパティおよびメソッド

AllMembers.T:System.Int32

必要条件

バージョン .NET

参照

概念

データ型の概要

その他の技術情報

データ型 (JScript)

long 型 (JScript)

long 型は、8 バイトの整数として格納されます。

long 型は、約 -1019 ～ 1019 の範囲の整数を表現できます。

対応する .NET Framework データ型は、Int64 です。**long** 型のプロパティとメソッドは、**Int64** のプロパティおよびメソッドと同じです。

プロパティおよびメソッド

AllMembers.T:System.Int64

必要条件

バージョン .NET

参照

概念

データ型の概要

その他の技術情報

データ型 (JScript)

Number 型

Number 型は、8 バイトの倍精度浮動小数点数として格納されます。倍精度の 64 ビット IEEE 754 値を表します。

Number 型は、最大で 1E+308 (正または負)、最小で 1E-323 を 15 桁の精度で表すことができます。また、**Number** 型では、非数 (**NaN**)、正または負の無限大、および正または負の 0 を表すことができます。

大きい数値が必要であっても厳密さは要求されないアプリケーションには、この型が便利です。正確な数値が必要な場合は、**Decimal** 型を使用します。

対応する .NET Framework データ型は、Double です。**Number** 型は、**double** 型と同じです。

解説

Number 型のプロパティとメソッドは、**Double** のプロパティおよびメソッドと同じです。

JScript では、**Number** オブジェクトも定義されます。**Number** 型と **Number** オブジェクトは相互運用されます。このため、**Number** オブジェクトは **Number** 型のメソッドやプロパティを呼び出すことができ、**Number** 型は **Number** オブジェクトのメソッドやプロパティを呼び出すことができます。詳細については、「Number オブジェクトのプロパティとメソッド」を参照してください。また、**Number** オブジェクトは **Number** 型を受け取る関数で使用でき、その逆も可能です。

通常は、**Number** オブジェクトの代わりに **Number** 型を使用してください。

プロパティおよびメソッド

AllMembers.T:System.Double

必要条件

バージョン .NET

参照

関連項目

double 型 (JScript)
decimal 型 (JScript)
Number オブジェクト

概念

データ型の概要

その他の技術情報

データ型 (JScript)

sbyte 型 (JScript)

sbyte 型は、符号付きの 1 バイトとして格納されます。

sbyte 型は、-128 ～ 127 の範囲の整数を表現できます。

対応する .NET Framework データ型は、SByte です。**sbyte** 型のプロパティとメソッドは、**SByte** のプロパティおよびメソッドと同じです。

プロパティおよびメソッド

AllMembers.T:System.SByte

必要条件

バージョン .NET

参照

概念

データ型の概要

その他の技術情報

データ型 (JScript)

short 型 (JScript)

short 型は、2 バイトの整数として格納されます。

short 型は、-32,768 ～ 32,767 の範囲の整数を表現できます。

対応する .NET Framework データ型は、Int16 です。**short** 型のプロパティとメソッドは、**Int16** のプロパティおよびメソッドと同じです。

プロパティおよびメソッド

AllMembers.T:System.Int16

必要条件

バージョン .NET

参照

概念

データ型の概要

その他の技術情報

データ型 (JScript)

String 型 (JScript)

String の長さは、0 ～ 約 20 億文字です。各文字は 16 ビット Unicode 値です。

対応する .NET Framework データ型は、String です。

解説

String 型のプロパティとメソッドは、**String** のプロパティおよびメソッドと同じです。

JScript では、**String** オブジェクトも定義されます。このデータ型は、**String** 型とは異なるプロパティとメソッドを提供します。**String** 型の変数にプロパティを作成したり、メソッドを追加したりすることはできません。**String** オブジェクトのインスタンスでは、これらの処理が可能です。

String オブジェクトは **String** データと相互運用されます。このため、**String** オブジェクトは **String** 型のメソッドやプロパティを呼び出すことができ、**String** 型は **String** オブジェクトのメソッドやプロパティを呼び出すことができます。詳細については、「String オブジェクトのプロパティとメソッド」を参照してください。また、**String** オブジェクトは **String** 型を受け取る関数で使用でき、その逆も可能です。

リテラル文字列にエスケープ シーケンスを使用すると、改行文字や Unicode 文字など、文字列で直接使用できない特殊文字を表すことができます。スクリプトをコンパイルすると、文字列リテラルの各エスケープ シーケンスは、それぞれが表している文字に変換されます。詳細については、「文字列データ」を参照してください。

JScript は、サロゲート ペアなどの特殊な Unicode 表記を解釈しません。また、比較時に文字列を正規化しません。

メモ：
単一の文字を表し、組み合わせた場合にだけ意味を持つ、対になった Unicode 文字は、サロゲート ペアと呼ばれます。

いくつかの文字は、複数の Unicode 文字の組み合わせによって表されます。正規化された表記が同じ文字を表している場合は、異なる表記も同一であると解釈されます。

プロパティおよびメソッド

AllMembers.T:System.String

必要条件

バージョン .NET

参照

関連項目

String オブジェクト
char 型 (JScript)

概念

データ型の概要
文字列データ

その他の技術情報

データ型 (JScript)

uint 型

uint 型は、4 バイトの符号なし整数として格納されます。

uint 型は、0 ～ 4,294,967,295 の範囲の整数を表現できます。

対応する .NET Framework データ型は、UInt32 です。**uint** 型のプロパティとメソッドは、**UInt32** のプロパティおよびメソッドと同じです。

プロパティおよびメソッド

AllMembers.T:System.UInt32

必要条件

バージョン .NET

参照

概念

データ型の概要

その他の技術情報

データ型 (JScript)

ulong 型 (JScript)

ulong 型は、8 バイトの符号なし整数として格納されます。

ulong 型は、0 ～ 1020 の範囲の整数を表現できます。

対応する .NET Framework データ型は、UInt64 です。**ulong** 型のプロパティとメソッドは、**UInt64** のプロパティおよびメソッドと同じです。

プロパティおよびメソッド

AllMembers.T:System.UInt64

必要条件

バージョン .NET

参照
概念
データ型の概要
その他の技術情報
データ型 (JScript)

ushort 型 (JScript)

ushort 型は、2 バイトの符号なし整数として格納されます。

ushort 型は、0 ～ 65,535 の範囲の整数を表現できます。

対応する .NET Framework データ型は、UInt16 です。**ushort** 型のプロパティとメソッドは、**UInt16** のプロパティおよびメソッドと同じです。

プロパティおよびメソッド

AllMembers.T:System.UInt16

必要条件

バージョン .NET

参照

概念

データ型の概要

その他の技術情報

データ型 (JScript)

ディレクティブ

JScript のディレクティブは、特定のコンパイラ、デバッガ、およびエラー メッセージのオプションを制御します。

このセクションの内容

@debug ディレクティブ

デバッグ シンボルの出力をオンまたはオフにします。

@position ディレクティブ

わかりやすい位置情報をエラー メッセージに表示します。

関連するセクション

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

.NET Framework リファレンス

.NET Framework クラス ライブラリおよびその他の基本要素の構文と構造を説明するトピックへのリンクを示します。

@debug ディレクティブ

デバッグ シンボルの出力をオンまたはオフにします。

```
@set @debug(on | off)
```

引数

on

既定値です。デバッグをオンにします。

off

省略可能です。デバッグをオフにします。

解説

JScript コードの作成者が記述するプログラム コードと、コンパイルされて実行される実際のコードは異なる場合があります。これは、ASP.NET などのホスト環境や開発ツールが独自のコードを生成し、プログラムに追加することがあるためです。一般的に、このようなコードは、コード作成者がデバッグするときには関係ありません。したがって、コードの作成者は、コードをデバッグするときに開発ツールが生成した部分を参照せずに、自分が記述した部分だけを確認しようとします。同様の理由で、パッケージ作成者はデバッグをオフにすることがあります。

コンパイラは、**/debug** オプションを指定してコマンド ラインからコンパイルする場合、または **@page** ディレクティブでデバッグ フラグを設定して ASP.NET ページをコンパイルする場合にだけ、デバッグ シンボルを生成します。このような状況では、**debug** ディレクティブは既定でオンになります。**debug** ディレクティブは、ファイルの終端に到達するまで、または次の **debug** ディレクティブが見つかるまで有効です。

debug ディレクティブがオフの場合、コンパイラはローカル変数 (関数またはメソッド内で定義される変数) に関するデバッグ情報を生成しません。ただし、**debug** ディレクティブが、グローバル変数に関するデバッグ情報の生成を妨げることはありません。

使用例

次のコードは、**/debug** オプションを指定してコマンド ラインからコンパイルされた場合にローカル変数 `debugOnVar` のデバッグ シンボルを生成しますが、`debugOffVar` のデバッグ シンボルは生成しません。

```
function debugDemo() {
   // Turn debugging information off for debugOffVar.
   @set @debug(off)
   var debugOffVar = 42;
   // Turn debugging information on.
   @set @debug(on)

   // debugOnVar has debugging information.
   var debugOnVar = 10;

   // Launch the debugger.
   debugger;
}

// Call the demo.
debugDemo();
```

必要条件

バージョン .NET

参照

関連項目

@set ステートメント
@position ディレクティブ
/debug
debugger ステートメント

その他の技術情報

JScript コードの作成、コンパイル、およびデバッグ

@position ディレクティブ

わかりやすい位置情報をエラー メッセージに表示します。

```
@set @position(end | [file = fname ;] [line = lnum ;] [column = cnum])
```

引数
fname

file がある場合は、必ず指定します。ファイル名を表すリテラル文字列で、ドライブまたはパス情報も指定できます。

lnum

line がある場合は、必ず指定します。コードの行を表す正の整数。

cnum

column がある場合は、必ず指定します。コードの列を表す正の整数。

解説

JScript コードの作成者が記述するプログラム コードと、コンパイルされて実行される実際のコードは異なる場合があります。これは、ASP.NET などのホスト環境や開発ツールが独自のコードを生成し、プログラムに追加することがあるためです。一般的に、このようなコードは作成者に関係ありませんが、エラーが発生したときに混乱を招く可能性があります。

コンパイラが、エラーの発生した行を作成者のコードで正しく識別する代わりに、元のコードに存在しない行を誤って識別する場合があります。これは、生成された追加コードによって、元のコードの相対位置が変更されているためです。

使用例

次の例では、ファイルの行番号が、JScript ホストによって挿入されたコードに合わせて変更されています。左の列の行番号が、開発者に対して表示される元のソースの行番号です。

```
01  ..  // 10 lines of host-inserted code.
..  ..  //...
10  ..  // End of host-inserted code.
11  ..  @set @position(line = 1)
12  01  var i : int = 42;
13  02  var x = ; // Error reported as being on line 2.
14  03  //Remainder of file.
```

必要条件

バージョン .NET

参照
関連項目
@set ステートメント
@debug ディレクティブ

エラー

エラー メッセージは、スクリプトの予期しない結果や動作をトラブルシューティングする場合に役立ちます。ここでは、実行時に発生するエラー、または不正な構文により発生するエラーを解決する方法を説明します。

このセクションの内容

JScript ランタイム エラー

JScript 構文エラー

関連するセクション

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

エラー メッセージ

Visual Studio 統合開発環境 (IDE: Integrated Development Environment) およびその他の開発言語に関連するエラー メッセージへのリンク一覧を示します。

JScript ランタイム エラー

JScript ランタイム エラーは、JScript のスクリプトがシステムで処理できない動作を実行しようとした場合に発生します。JScript ランタイム エラーは、スクリプトの実行中に変数の式が評価され、メモリが動的に割り当てられるときに発生します。

エラーのトラップ

ランタイム エラーは、JScript プログラムでトラップして調べることができます。エラーを生成するコードを **try** ブロックで囲み、スローされたエラーは **catch** ブロックでキャッチできます。JScript がスローするエラーは **Error** オブジェクトです。JScript コードでは、**throw** ステートメントを使用して、**Error** オブジェクトを含む任意のデータ型のカスタム エラーを生成できます。キャッチされた **Error** オブジェクトの番号とメッセージをプログラムで表示して、エラーを識別できます。エラーがキャッチされない場合、スクリプトは終了します。

特定のエラー メッセージについて調べるには、次のような方法があります。

- [JScript ランタイム エラー] ノードの目次で、エラー番号とメッセージを検索します。
- [キーワード] の [検索する文字列] ボックスに、エラー番号を入力します。エラー番号は "JSxxxx" の形式で、xxxx は 4 桁のエラー コードです。
- [検索] の [検索する文字列] ボックスに、エラー メッセージを入力します。エラー メッセージには、単一引用符で囲まれた語が含まれる場合があります。引用符で囲まれた語句はコード中の識別子を表し、エラー メッセージの一部ではありません。検索には、引用符で囲まれた語句は含めないでください。

参照

関連項目

Error オブジェクト
try...catch...finally ステートメント
throw ステートメント

概念

JScript 構文エラー

JS5000: 'this' に割り当てることはできません。

this に値を割り当てています。this は、次のいずれかを参照する JScript のキーワードです。

- 現在メソッドを実行しているオブジェクト
- 現在のメソッドがない場合 (またはメソッドがほかのどのオブジェクトにも属していない場合) はグローバル オブジェクト

メソッドは、オブジェクトによって呼び出された JScript 関数です。メソッド内部では、キーワード **this** は、メソッドの呼び出し元のオブジェクトへの参照です。また、**new** 演算子を使用して呼び出したクラス コンストラクタによって作成されたオブジェクトに対する参照の場合もあります。

メソッド内部では、**this** を使用して現在のオブジェクトを参照することはできますが、新しい値を **this** に割り当てることはできません。

このエラーを解決するには

- **this** に値を割り当てないようにします。インスタンス化されたオブジェクトのプロパティまたはメソッドにアクセスするには、ドット演算子を使用します (circle.radius など)。

メモ :
ユーザーが作成した変数に **this** という名前を付けることはできません。**this** は JScript の予約語です。

参照

関連項目
this ステートメント
概念
JScript スクリプトのトラブルシューティング
その他の技術情報
JScript リファレンス

JS5001: 数字が必要です。

Number.prototype.toString メソッドまたは **Number.prototype.valueOf** メソッドが、**Number** 型以外のオブジェクトから呼び出されました。この場合の呼び出し元のオブジェクトは、**Number** 型である必要があります。

このエラーを解決するには

- **Number.prototype.toString** メソッドまたは **Number.prototype.valueOf** メソッドの呼び出しは、**Number** 型のオブジェクトでだけ行います。

参照

関連項目

Number オブジェクト
toString メソッド
valueOf メソッド
prototype プロパティ

その他の技術情報

オブジェクト (JScript)

JS5002: 関数が必要です。

Function オブジェクト以外のオブジェクトで **Function prototype** メソッドを呼び出そうとしたか、関数呼び出しコンテキストでオブジェクトを使用しました。次のコード例では、**mysample** が関数ではないため、このエラーが発生します。

```
var mysample = new Object();  // Create a new object called "mysample".
var x = mysample();           // Try and call mysample as if it were a function.
```

このエラーを解決するには

1. **Function** オブジェクトだけが、**Function prototype** メソッドを呼び出すようにします。
2. 関数呼び出し演算子 **()** が、関数だけを呼び出すようにします。

参照

関連項目

Function オブジェクト
prototype プロパティ

その他の技術情報

オブジェクト (JScript)

JS5003: 関数の結果に割り当てられません。

関数の戻り値に値を代入しようとしました。関数の戻り値は、変数に代入することはできますが、変数として使用することはできません。関数自体に新しい値を割り当てる場合は、かっこ (関数呼び出し演算子) を省略します。

このエラーを解決するには

1. 関数の戻り値を"代入先" として使用しないようにします。ただし、関数の戻り値を "変数" に代入することはできます。

```
myVar = myFunction(42);
```

2. または、関数の戻り値ではなく、関数自体を変数に割り当てることもできます。

```
myFunction = new Function("return 42;");
```

参照

関連項目

Function オブジェクト

その他の技術情報

JScript 言語の紹介

JScript の関数

メソッド

JS5005: 文字列が必要です。

String 型以外のオブジェクトで、**String.prototype.toString** メソッドまたは **String.prototype.valueOf** メソッドを呼び出しました。この場合の呼び出し元のオブジェクトは、**String** 型である必要があります。

このエラーを解決するには

- **String.prototype.toString** メソッドまたは **String.prototype.valueOf** メソッドの呼び出しは、**String** 型のオブジェクトでだけ行います。

参照

関連項目

String オブジェクト
toString メソッド
valueOf メソッド
prototype プロパティ

その他の技術情報

オブジェクト (JScript)

JS5006: Date オブジェクトが必要です。

Date 型以外のオブジェクトで、**Date.prototype.toString** メソッドまたは **Date.prototype.valueOf** メソッドを呼び出しました。この場合の呼び出し元のオブジェクトは、**Date** 型である必要があります。

このエラーを解決するには

- **Date.prototype.toString** メソッドまたは **Date.prototype.valueOf** メソッドの呼び出しは、**Date** 型のオブジェクトでだけ行います。

参照

関連項目

Date オブジェクト
toString メソッド
valueOf メソッド
prototype プロパティ

その他の技術情報

オブジェクト (JScript)

JS5007: オブジェクトが必要です。

Object 型以外のオブジェクトで、**Object.prototype.toString** メソッドまたは **Object.prototype.valueOf** メソッドを呼び出しました。この場合の呼び出し元のオブジェクトは、**Object** 型である必要があります。

このエラーを解決するには

- **Object.prototype.toString** メソッドまたは **Object.prototype.valueOf** メソッドの呼び出しは、**Object** 型のオブジェクトでだけ行います。

参照

関連項目

Object オブジェクト
toString メソッド
valueOf メソッド
prototype プロパティ

その他の技術情報

オブジェクト (JScript)

JS5008: 無効な代入です。

読み取り専用の識別子に値を代入しようとしました。読み取り専用の識別子には、値を代入できません。たとえば、ホスト定義のオブジェクトや外部 COM オブジェクトは、読み取り専用の識別子です。

このエラーを解決するには

- 読み取り専用の識別子に値を代入しないようにします。

参照

関連項目

代入演算子 (=)

JS5009: 未定義の識別子です。

JScript コンパイラが、識別子を認識できません。参照されている変数が存在しないか、**with** ブロックがアクセスするオブジェクト プロパティが存在しない可能性があります。

このエラーを解決するには

1. **var** ステートメントで変数を宣言します (**var** x; など)。
2. **with** ブロックで、有効なオブジェクト メンバだけを参照していることを確認します。
3. **with** ステートメントを使用せずに、オブジェクトを明示的に参照します。

参照

関連項目

var ステートメント
with ステートメント

概念

変数と定数のスコープ

その他の技術情報

JScript の変数と定数
JScript リファレンス

JS5010: ブール型 (Boolean) が必要です。

Boolean 型以外のオブジェクトで、**Boolean.prototype.toString** メソッドまたは **Boolean.prototype.valueOf** メソッドを呼び出しました。この場合の呼び出し元のオブジェクトは、**Boolean** 型である必要があります。

このエラーを解決するには

- Boolean.prototype.toString メソッドまたは Boolean.prototype.valueOf メソッドの呼び出しは、**Boolean** 型のオブジェクトでだけ行います。

参照

関連項目

Boolean オブジェクト
toString メソッド
valueOf メソッド
prototype プロパティ

その他の技術情報

オブジェクト (JScript)

JS5013: VBArray が必要です。

Visual Basic safeArray のオブジェクトが必要な箇所で、**VBArray** に Visual Basic safeArray ではないオブジェクトが指定されました。Visual Basic safeArrays は、JScript では直接作成できません。既存の ActiveX オブジェクトやほかのオブジェクト、または同じ Web ページの Visual Basic スクリプトから取得して、インポートする必要があります。

このエラーを解決するには

1. **VBArray** コンストラクタには、Visual Basic safeArray オブジェクトだけを渡します。
2. **System.Array** オブジェクトを使用します。これにより、すべての .NET 言語 (JScript および Visual Basic を含む) で、配列にアクセスしたり、配列を修正したりできます。

参照

関連項目

VBArray オブジェクト

概念

配列の使用方法

JS5015: Enumerator オブジェクトが必要です。

Enumerator 型以外のオブジェクトで、**Enumerator.prototype.atEnd**、**Enumerator.prototype.item**、**Enumerator.prototype.moveFirst**、または **Enumerator.prototype.moveNext** の各メソッドを呼び出しました。この場合の呼び出し元のオブジェクトは、**Enumerator** 型である必要があります。

このエラーを解決するには

- **Enumerator.prototype.atEnd**、**Enumerator.prototype.item**、**Enumerator.prototype.moveFirst**、または **Enumerator.prototype.moveNext** の各メソッドの呼び出しは、**Enumerator** 型のオブジェクトでだけ行います。**Enumerator** 型のオブジェクトかどうかを確認するには、次のように記述します。

```
if(x instanceof Enumerator)
```

参照

関連項目

Enumerator オブジェクト
atEnd メソッド
item メソッド (JScript)
moveFirst メソッド
moveNext メソッド
prototype プロパティ

その他の技術情報

オブジェクト (JScript)

JS5016: Regular Expression オブジェクトが必要です。

RegExp 型以外のオブジェクトで、**RegExp.prototype.toString** メソッドまたは **RegExp.prototype.valueOf** メソッドを呼び出しました。この場合の呼び出し元のオブジェクトは、**RegExp** 型である必要があります。

このエラーを解決するには

- **RegExp.prototype.toString** メソッドまたは **RegExp.prototype.valueOf** メソッドの呼び出しは、**RegExp** 型のオブジェクトでだけ行います。

参照

関連項目

Regular Expression オブジェクト
toString メソッド
valueOf メソッド
prototype プロパティ

概念

正規表現の構文

その他の技術情報

オブジェクト (JScript)

JS5017: 正規表現で構文エラーが発生しました。

正規表現の検索文字列の構文が、JScript 正規表現の 1 つ以上の文法規則に違反しています。

このエラーを解決するには

- 正規表現の検索文字列が、JScript 正規表現の構文に準拠するようにします。

参照

関連項目

Regular Expression オブジェクト
compile メソッド (JScript)

概念

正規表現の構文

JS5022: 例外の値がスローされ、キャッチされませんでした。

コードに **throw** ステートメントが記述されていますが、**try** ブロック内にないか、エラーをトラップするための対応する **catch** ブロックがありません。例外は、**try** ブロック内で **throw** ステートメントによってスローされ、**try** ブロックの外側にある **catch** ステートメントによってキャッチされます。

このエラーを解決するには

1. 例外をスローするコードを **try** ブロック内に記述し、対応する **catch** ブロックがあることを確認します。
2. 使用する catch ステートメントが、適切な例外書式に対応するよう記述されていることを確認します。
3. 例外が再スローされる場合には、これに対応する別の catch ステートメントが用意してあることを確認します。

参照

関連項目

Error オブジェクト
throw ステートメント
try...catch...finally ステートメント

JS5023: 関数には、有効なプロトタイプ オブジェクトが存在しません。

instanceof 演算子を使用して、オブジェクトが特定の関数クラスから派生したかどうかを調べようとしましたが、オブジェクトの **prototype** プロパティが **null** または外部オブジェクト型として再定義されていました (両者とも有効な JScript オブジェクトではありません)。外部オブジェクトは、ホスト オブジェクト モデルのオブジェクト (Internet Explorer のドキュメントやウィンドウ オブジェクトなど) または外部 COM オブジェクトのどちらでもかまいません。

このエラーを解決するには

- 関数の **prototype** プロパティが、有効な JScript オブジェクトを参照するようにします。

参照

関連項目

Function オブジェクト
prototype プロパティ

その他の技術情報

オブジェクト (JScript)

JS5024: エンコードする URI は無効な文字を含んでいます。

URI (Uniform Resource Identifier) としてエンコードされた文字列に、無効な文字が含まれています。文字列の大半の文字は正常に URI に変換できますが、一部に無効な Unicode 文字シーケンスがあります。

このエラーを解決するには

- エンコードする文字列に有効な Unicode シーケンスだけが含まれるようにします。

  完全な URI は、構成要素と区切り記号の並びで構成されます。一般的な書式は次のとおりです。

  ```
  <Scheme>:<first>/<second>;<third>?<fourth>
  ```

  山かっこの中の名前は構成要素を表し、":"、"/"、";"、"?" の各文字は区切り記号として予約されています。

参照

関連項目

encodeURI メソッド
encodeURIComponent メソッド

JS5025: デコードする URI は正しくエンコードされていません。

不正な書式の URI (Uniform Resource Identifier) をデコードしようとしました。URI には特別な構文があり、アルファベット以外の大半の文字は、URI で使用するにはかっこで囲む必要があります。**encodeURI** メソッドと **encodeURIComponent** メソッドは、通常の JScript 文字列から URI を作成できません。

完全な URI は、構成要素と区切り記号の並びで構成されます。一般的な書式は次のとおりです。

```
<Scheme>:<first>/<second>;<third>?<fourth>
```

山かっこの中の名前は構成要素を表し、":"、"/"、";"、"?" の各文字は区切り記号として予約されています。

このエラーを解決するには

- 有効な URI だけをデコードするようにします。たとえば、通常の JScript 文字列は無効な文字を含む場合があり、有効な URI でない可能性があります。

参照

関連項目

decodeURI メソッド
decodeURIComponent メソッド

JS5026: 小数の桁数が有効範囲を超えています。

関数 **Number.prototype.toExponential** は、無効な引数を受け取ることができません。関数 **toExponential()** への引数は、0 〜 20 の範囲内である必要があります。

このエラーを解決するには

- **toExponential()** の引数に正しい値を指定します。

参照

関連項目

Number オブジェクト
toExponential メソッド
prototype プロパティ

その他の技術情報

オブジェクト (JScript)

JS5027: 有効桁数の範囲を超えています。

関数 **Number.prototype.toPrecision** は、無効な引数を受け取ることができません。関数 **toPrecision** への引数は、1 ～ 21 の範囲内である必要があります。

このエラーを解決するには

- **toPrecision** の引数に正しい値を指定します。

参照

関連項目

Number オブジェクト
toPrecision メソッド
prototype プロパティ

その他の技術情報

オブジェクト (JScript)

JS5029: 配列の長さは、0 または正の整数でなければなりません。

Array コンストラクタを呼び出す引数が、負または非数 (**NaN**) です。JScript では、小数は自動的に 0 または正の整数に変換されます。

このエラーを解決するには

- **Array** オブジェクトを新規作成するときは、正の整数または 0 だけを使用します。要素が 1 つだけの配列を作成するには、次の 2 つの手順を実行します。まず、要素が 1 つの配列を作成します。次に、1 番目の要素 (array[0]) に値を設定します。

  数値要素 1 つだけの配列を指定する正しい方法を次に示します。

  ```
  var piArray = new Array(1);
  piArray [0] = 3.14159;
  ```

  配列サイズに上限はありませんが、整数値の最大値には制限があります (約 40 億)。

参照

概念

配列の使用方法

その他の技術情報

JScript リファレンス

JS5030: 配列の長さには、正の整数または 0 を割り当てなければなりません。

Array オブジェクトの **length** プロパティに代入された値が、負の数値または非数 (**NaN**) です。JScript では、小数は自動的に 0 または正の整数に変換されます。

このエラーを解決するには

- length プロパティに 0 または正の整数を代入します。**Array** オブジェクトの **length** プロパティを設定する正しい方法を次に示します。

```
var my_array = new Array();
my_array.length = 99;
```

  配列サイズに上限はありませんが、整数値の最大値には制限があります (約 40 億)。

参照

概念

配列の使用方法

その他の技術情報

JScript リファレンス

JS5031: Array オブジェクトの配列が必要です。

Array オブジェクトが必要なコンテキストで、**Array** オブジェクト以外を使用しようとしました。

このエラーを解決するには

- このコンテキストで、**Array** オブジェクトが使用されていることを確認します。

参照

関連項目

Array オブジェクト

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS5032: 指定されたコンストラクタはありません。

識別子に **new** 演算子が適用されていますが、識別子がコンストラクタ関数およびクラス コンストラクタのいずれにも対応していません。

このエラーを解決するには

- new 演算子が、コンストラクタ関数またはクラス コンストラクタに適用されていることを確認します。

参照

概念

JScript スクリプトのトラブルシューティング
コンストラクタ関数による独自のオブジェクトの作成
独自のクラスの作成

その他の技術情報

JScript リファレンス

JS5033: Eval を別名を使って呼び出すことはできません。

eval メソッドと等価の変数セットを持つプログラムが、別名を定義して、別名を関数として使用しています。別名は **eval** メソッドでは使用できません。

このエラーを解決するには

- **eval** メソッドを直接呼び出します。

参照

関連項目

eval メソッド (JScript)

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS5034: 実装されていません。

実装されていない機能を使用しようとしました。

このエラーを解決するには

- 実装されていない機能への参照を削除します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS5035: Null または空の名前付きパラメータ名を指定することはできません。

名前付きパラメータが JScript 関数またはメソッドを呼び出していますが、名前付きパラメータのいずれかの名前が空か null です。すべてのパラメータは null 以外の名前を持ちます。

メモ：

JScript を使用して関数およびメソッドを呼び出す場合は、名前付きパラメータは使用できません。ただし、JScript 関数およびメソッドを、名前付きパラメータをサポートする他の言語 (たとえば Visual Basic) から呼び出すことはできます。詳細については、「位置と名前による引数渡し」を参照してください。

このエラーを解決するには

- それぞれの名前付きパラメータに、パラメータ名を指定します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS5036: 名前付きパラメータ名が重複しています。

名前付きパラメータが JScript 関数またはメソッドを呼び出していますが、名前付きパラメータの名前が重複しています。パラメータの名前は一意である必要があります。

メモ：
JScript を使用して関数およびメソッドを呼び出す場合は、名前付きパラメータは使用できません。ただし、JScript 関数およびメソッドを、名前付きパラメータをサポートする他の言語 (たとえば Visual Basic) から呼び出すことはできます。詳細については、「位置と名前による引数渡し」を参照してください。

このエラーを解決するには

- それぞれの名前付きパラメータに、一意の名前を指定します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS5037: 指定された名前は、パラメータの名前ではありません。

名前付きパラメータが JScript 関数またはメソッドを呼び出していますが、名前付きパラメータのいずれかの名前がパラメータ名に対応していません。名前付きパラメータの名前は、パラメータ名を参照している必要があります。

メモ：

JScript を使用して関数およびメソッドを呼び出す場合は、名前付きパラメータは使用できません。ただし、JScript 関数およびメソッドを、名前付きパラメータをサポートする他の言語 (たとえば Visual Basic) から呼び出すことはできます。詳細については、「位置と名前による引数渡し」を参照してください。

このエラーを解決するには

- それぞれの名前付きパラメータに、パラメータ名を指定します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS5038: 指定された引数が少なすぎます。

指定された引数が少なすぎます。名前付きパラメータ名の数が引き渡された引数の数を超えることはできません。

名前付きパラメータが JScript 関数またはメソッドを呼び出していますが、渡された引数の数が、関数またはメソッドで指定されている引数の数を超えています。渡された引数の少なくとも 1 つが破棄されるため、これは認められません。

メモ：

JScript を使用して関数およびメソッドを呼び出す場合は、名前付きパラメータは使用できません。ただし、JScript 関数およびメソッドを、名前付きパラメータをサポートする他の言語 (たとえば Visual Basic) から呼び出すことはできます。詳細については、「位置と名前による引数渡し」を参照してください。

このエラーを解決するには

- 渡される引数の数が、パラメータ名の数を超えないようにします。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS5039: この式をデバッガで評価できません。

JScript プログラムのデバッグ中に、評価できない式をコマンド ウィンドウに入力しました。

このエラーを解決するには

- コマンド ウィンドウには有効な JScript 式だけを入力します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JScript コードの作成、コンパイル、およびデバッグ

JS5040: 読み取り専用のフィールドまたはプロパティに割り当てられました。

読み取り専用の識別子に値を代入しています。読み取り専用の識別子に値を書き込むことはできません。

const ステートメントは、読み取り専用のフィールドまたは定数を定義します。対応する **function set** ステートメントがない場合、**function get** ステートメントは読み取り専用のプロパティを定義します。

このエラーを解決するには

1. 読み取り専用の識別子に値が代入されないことを確認します。
2. **var** ステートメントを使用してフィールドまたは変数を定義し、代入できるようにします。
3. 対応する **function set** ステートメントをプロパティに追加し、代入できるようにします。

参照

関連項目

const ステートメント
function get ステートメント
function set ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス
クラス ベースのオブジェクト

JS5041: このプロパティは割り当て元としてのみ指定できます。

書き込み専用のプロパティの値を読み取ろうとしました。書き込み専用の識別子から値を読み取ることはできません。

対応する **function get** ステートメントがない場合、**function set** ステートメントは書き込み専用のプロパティを定義します。

このエラーを解決するには

1. 書き込み専用のプロパティから値を読み取らないようにします。
2. 対応する **function get** ステートメントをプロパティに追加し、値を読み取ることができるようにします。

参照

関連項目

function get ステートメント
function set ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス
クラス ベースのオブジェクト

JS5042: インデックスの数が配列の次元と一致しません。

配列の要素にアクセスしていますが、インデックスの数と配列の次元数が一致しません。

このエラーを解決するには

- 配列要素にアクセスするために使用するインデックス番号と、配列を定義するときに指定した次元数が一致することを確認します。

参照

概念

型指定された配列

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS5043: ref パラメータを含むメソッドを、デバッガで呼び出すことはできません。

JScript プログラムのデバッグ中に、参照渡しでパラメータを受け取るメソッドの呼び出しがコマンド ウィンドウに入力されました。参照渡しのパラメータは変数の値を変更できるため、この操作は実行できません。

このエラーを解決するには

- コマンド ウィンドウでは、パラメータを参照渡しで受け取るメソッドを呼び出さないようにします。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JScript コードの作成、コンパイル、およびデバッグ

JS5044: Deny、PermitOnly、および Assert セキュリティ メソッドを遅延バインディングを使って呼び出すことはできません。

遅延バインディングされた **CodeAccessPermission** オブジェクトの、**PermitOnly**、**Assert**、または **Deny** メソッドを呼び出しています。セキュリティ上の理由から、これは許可されていません。**PermitOnly**、**Assert**、および **Deny** メソッドを使用するには、**CodeAccessPermission** オブジェクトを格納する変数を明示的に型指定 (事前バインディング) して、**CodeAccessPermission** オブジェクトだけを格納する必要があります。

このエラーを解決するには

- **CodeAccessPermission** オブジェクトを格納する変数を定義する場合は、型の注釈を使用します。

参照

関連項目

CodeAccessPermission Class

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS5045: JScript は、宣言のセキュリティ属性をサポートしません。

CodeAccessSecurityAttribute から継承するカスタム属性が、メソッド、クラス、またはアセンブリの定義に適用されています。これは認められていません。宣言セキュリティ属性の代わりに動的なセキュリティを使用して、コードへのアクセスを管理する必要があります。

メモ：

.NET Framework 内では、**Assert**、**Deny**、または **PermitOnly** セキュリティ メソッドを呼び出すことができるのは、事前バインディングされたコードだけです。したがって、型の注釈を指定した変数を使用して、アクセス許可オブジェクトを格納する必要があります。型の注釈の使用により、コンパイラは、事前バインディングされたコードを生成できます。また、**eval** メソッドまたは **new** 演算子で生成された **Function** オブジェクトを使って実行時に生成されたコードは、遅延バインディングされたコードです。このコードにより、**Assert**、**Deny**、または **PermitOnly** メソッドへの呼び出しを行うことができなくなります。

次の例では、動的なセキュリティを使用して、メソッドによる特定のファイルへのアクセスを拒否しています。

```
import System;
import System.IO;
import System.Security;
import System.Security.Permissions;
class Alpha{
   function Bravo() {
      var fileioperm : FileIOPermission;
      fileioperm = new FileIOPermission(FileIOPermissionAccess.AllAccess, 'd:\\temp\\myfile
.txt');
      fileioperm.Deny();
      // Any additional code in this method will be
      // denied access to d:\temp\myfile.txt.
   }
}
```

このエラーを解決するには

- 宣言セキュリティの代わりに動的なセキュリティを使用して、安全なメソッドまたはアセンブリを宣言します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JScript 構文エラー

JScript 構文エラーは、JScript ステートメントの構造が JScript スクリプト言語の文法規則に違反した場合に発生します。JScript 構文エラーは、プログラム実行前のコンパイル時に発生します。

エラー メッセージについて調べる方法

特定のエラー メッセージについて調べるには、次のような方法があります。

- [JScript 構文エラー] ノードの目次で、エラー番号とメッセージを検索します。
- [キーワード] の [検索する文字列] ボックスに、エラー番号を入力します。エラー番号は "JSxxxx" の形式で、xxxx は 4 桁のエラー コードです。
- [検索] の [検索する文字列] ボックスに、エラー メッセージを入力します。エラー メッセージには、単一引用符で囲まれた語が含まれる場合があります。引用符で囲まれた語句はコード中の識別子を表し、エラー メッセージの一部ではありません。検索には、引用符で囲まれた語句は含めないでください。

参照

概念

JScript ランタイム エラー

JS0005: プロシージャの呼び出し、または引数が無効です。

プロシージャが呼び出されましたが、呼び出しに対応するプロシージャが定義されていません。

このエラーを解決するには

1. プロシージャに渡すデータ型と、プロシージャ側で受け取るデータ型として定義されたデータ型が一致していることを確認します。
2. プロシージャに渡す引数の数が、プロシージャの受け取る引数の数と一致していることを確認します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS0007: メモリが不足しています。

使用可能なメモリは、すべて使用されています。このエラーは、大量のデータを処理するときに発生する場合があります。エラーを回避するには、使用可能なメモリを効率的に使用するようにします。つまり、プログラムが不要なメモリを配列やオブジェクトなどの形で予約しないようにします。

プログラムが使用するメモリを節約するには、ガベージ コレクション ルーチンを使用してメモリを動的に解放する方法もあります。JScript で使用されるガベージ コレクション ルーチンは、使われていないメモリの解放を処理します。ルーチンは、プログラムからアクセスされなくなったデータを解放します。変数のデータが新しいデータで置き換えられたり、スコープが変化して変数にアクセスできなくなると、データにアクセスできなくなる場合があります。

プログラム内でメモリを解放するには、サイズの大きい配列やオブジェクトなど、メモリを大量に使用する変数が不要になったときに、これらを **null** に設定します。これにより、ガベージ コレクタがメモリを解放します。

このエラーを解決するには

1. コードでメモリが効率的に使用されていることを確認します。
2. メモリを大量に使用するオブジェクトは、必要になる直前に宣言します。
3. 不要になった変数を **null** に設定します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS0013: 型が一致しません。

互換性のないデータ型を使用しようとしています。このエラーは、変数に値を代入するとき、または型の注釈が指定されているパラメータを持つ関数に対して引数を渡すときに発生することがあります。

このエラーを解決するには

- 指定されたデータ型に強制変換できる型のデータを渡します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS0028: スタック領域が不足しています。

使用可能なスタック領域は、すべて使用されています。このエラーは、再帰関数が明示的に終了していない場合に発生します。

このエラーを解決するには

- 再帰関数を明示的に終了させます。

参照

概念

JScript スクリプトのトラブルシューティング

再帰

その他の技術情報

JScript リファレンス

JS0051: 内部エラーです。

スクリプト エンジンで内部エラーが発生しました。

このエラーを解決するには

1. 同等の別の方法で目的の結果が得られるように、コードを再作成します。
2. このページの下部にある [Microsoft Product Support Knowledge Base リンク] をクリックして、代替手段を検索します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS0053: ファイルが見つかりません。

アクセスしようとしたファイルが見つかりません。

このエラーを解決するには

1. パスとファイル名が正しいことを確認します。
2. すべてのファイルが存在していることを確認します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS0424: オブジェクトが必要です。

オブジェクトが必要なコンテキストで、オブジェクト以外のものを使用しようとしました。

このエラーを解決するには

- このコンテキストにオブジェクトが存在することを確認します。

参照

関連項目

Object オブジェクト
ActiveXObject オブジェクト

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS0429: オブジェクトを作成できません。

新しいオブジェクトを作成しようとしましたが作成できません。このエラーは、オブジェクトを提供するアプリケーションが使用できない場合や、アプリケーションが特定のオブジェクトを提供していない場合に発生します。

このエラーを解決するには

1. アプリケーションが使用可能であることを確認します。
2. オブジェクトを提供するために適切なアプリケーションが使用されていることを確認します。

参照

関連項目

ActiveXObject オブジェクト

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS0438: オブジェクトでサポートされていないプロパティまたはメソッドです。

プログラムがアクセスしようとしているプロパティまたはメソッドは、オブジェクトのメンバではありません。

このエラーを解決するには

- アクセスするプロパティやメソッドが有効であることを確認します。

参照

関連項目

ActiveXObject オブジェクト

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS0445: このオブジェクトではサポートされていない操作です。

オブジェクトが、サポートしていないアクションに対して使用されています。

このエラーを解決するには

- オブジェクトで有効なアクションだけが使用されていることを確認します。

参照

関連項目

ActiveXObject オブジェクト

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS0451: オブジェクトはコレクションではありません。

新しい **Enumerator** オブジェクトを作成しようとしましたが、コンストラクタに渡された引数がコレクションではありません。

メモ：
コレクションの要素は、JScript から直接アクセスできます。詳細については、「Enumerator オブジェクト」を参照してください。

このエラーを解決するには

- **Enumerator** オブジェクトの構築にコレクションだけが使用されていることを確認します。

参照

関連項目

Enumerator オブジェクト

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1002: 構文エラーです。

JScript ステートメントが、JScript の 1 つ以上の文法規則に違反しています。

このエラーを解決するには

1. 示されている行番号の構文をプログラムで再度確認します。
2. かっこや中かっこの使用に間違いがないかを確認します。

参照

関連項目

Error オブジェクト

その他の技術情報

JScript リファレンス

JS1003: ':' が必要です。

式で三項条件演算子を使用しましたが、第 2 オペランドと第 3 オペランドの間にコロンがありません。三項条件演算子 (3 つのオペランドを持つ演算子) では、第 2 オペランド (true) と第 3 オペランド (false) の間にコロンを指定する必要があります。

このエラーを解決するには

- 第 2 オペランドと第 3 オペランドの間にコロンを追加します。

参照

関連項目

条件 (三項) 演算子 (?:)

その他の技術情報

演算子 (JScript)

JScript リファレンス

JS1004: ';' が必要です。

1 行に複数のステートメントがありますが、これらのステートメントがセミコロンによって区切られていません。または、**for** ステートメントのヘッダーで、初期化式、条件式、およびインクリメント式の間にセミコロンがありません。

ステートメントを終了するには、セミコロンを使用します。1 行に複数のステートメントを記述できますが、各ステートメントはセミコロンで区切る必要があります。

また、**for** ループのヘッダーにある初期化式、条件式、およびインクリメント式もセミコロンで区切る必要があります。

このエラーを解決するには

1. 各ステートメントをセミコロンで終了するようにします。
2. 関数の呼び出しで、かっこが正しく指定されていることを確認します。
3. **for** ループのヘッダーにセミコロンがあることを確認します。
4. 代入に = があることを確認します。

参照

関連項目

for ステートメント

その他の技術情報

JScript 言語の紹介

JScript リファレンス

JS1005: '(' が必要です。

かっこで囲まれる式に左かっこがありません。一部の式は、左かっこと右かっこの中に記述する必要があります。たとえば、次の **for** ステートメントのかっこの配置は適切です。

```
for (initialize; test; increment) {
   statement;
}
```

このエラーを解決するには

- 対応する左かっこを追加します。

参照

その他の技術情報

JScript リファレンス

JS1006: ')' が必要です。

かっこで囲まれる式に右かっこがありません。一部の式は、左かっこと右かっこの中に記述する必要があります。たとえば、次の **for** ステートメントのかっこの配置は適切です。

```
for (initialize; test; increment) {
   statement;
}
```

このエラーを解決するには

- 対応する右かっこを追加します。

参照

その他の技術情報

JScript リファレンス

JS1007: ']' が必要です。

配列要素の参照に、右角かっこがありません。式で配列要素を参照するには、左角かっこと右角かっこが必要です。

このエラーを解決するには

- 配列要素を参照する式に右角かっこを追加します。

参照

関連項目

Array オブジェクト

概念

配列の使用方法

JS1008: '{' が必要です。

関数本体、クラス メンバ ブロック、インターフェイス メンバ ブロック、または列挙ブロックの開始を示す左中かっこがありません。関数本体のコードは、1 行であっても中かっこで囲む必要があります。

ループでの中かっこの使用は、関数本体やメンバ ブロックの場合ほど厳密ではありません。

このエラーを解決するには

- 関数本体の開始を示す左中かっこを追加します。

参照

関連項目

Function オブジェクト
function ステートメント
class ステートメント
interface ステートメント
enum ステートメント

その他の技術情報

JScript リファレンス

JS1009: '}' が必要です。

関数本体、クラス メンバ ブロック、インターフェイス メンバ ブロック、列挙ブロック、ループ、コードのブロック、またはオブジェクト初期化子の終了を示す右中かっこがありません。たとえば、**for** ループで、ループを開始する左中かっこだけが指定されているときに、このエラーが発生します。

このエラーを解決するには

- 関数、クラス、インターフェイス、列挙、ループ、ブロック、またはオブジェクト初期化子の終了を示す右中かっこを追加します。

参照

関連項目

Function オブジェクト
function ステートメント
class ステートメント
interface ステートメント
enum ステートメント

その他の技術情報

JScript の条件構造
JScript リファレンス

JS1010: 識別子が必要です。

コンテキスト内の必要な場所に識別子がありません。識別子として使用できる項目は、次のとおりです。

- 変数
- プロパティ
- 配列
- 関数名

このエラーを解決するには

- 識別子が等号の左側に表示されるように、式を変更します。

参照

概念

配列の使用方法

その他の技術情報

JScript リファレンス

JS1011: '=' が必要です。

条件付きコンパイル ステートメントで使用される変数で、変数と代入される値の間に等号 (=) がありません。

このエラーを解決するには

- 等号記号を追加します。次に例を示します。

```
@set @myvar1 = 1
```

参照

概念

条件付きコンパイル変数

その他の技術情報

条件付きコンパイル

JS1014: 無効な文字です。

識別子に、JScript コンパイラが有効な文字として識別できない文字が使用されています。次の規則に従う文字が有効です。

- 先頭文字は、ASCII 文字 (大文字でも小文字でもかまいません)、アンダースコア (_)、またはドル記号 ($) にする必要があります。
- 先頭文字に続く文字には、ASCII 文字、数字、アンダースコア、またはドル記号を使用できます。
- 識別子名に予約語は使用できません。

このエラーを解決するには

- JScript 言語定義に含まれていない文字は使用しないようにします。

参照

概念

文字列データ

その他の技術情報

JScript の変数と定数

データ型 (JScript)

JS1015: 未終了の文字列型の定数です。

文字列定数の終わりを表す引用符がありません。文字列定数は、ペアになった引用符で囲む必要があります。

メモ：
引用符には、単一引用符と二重引用符のどちらでも使用できますが、2 種類の引用符を組み合わせてペアにすることはできません。単一引用符で囲んだ文字列の内側に二重引用符のペアを使用したり、その逆の組み合わせで 2 組の引用符のペアを入れ子にして使用したりできます。

このエラーを解決するには

- 文字列の最後に引用符を追加します。

参照

関連項目

String オブジェクト
toString メソッド

JS1016: 未終了のコメントです。

複数行コメント ブロックが適切に終了していません。複数行コメントは "/*" で開始し、これを逆にした "*/" で終了する必要があります。

適切な複数行コメントの使用例を次に示します。

```
/* This is a comment
This is another part of the same comment.*/
```

このエラーを解決するには

- 複数行コメントは、必ず "*/" で終了するようにします。

参照

関連項目

コメント ステートメント

JS1018: 'return' ステートメントが関数の外にあります。

return ステートメントが、コードのグローバル スコープ内にあるか、クラスまたはパッケージの本体にあります。**return** ステートメントは、関数本体に記述する必要があります。

このエラーを解決するには

- **return** ステートメントを削除します。

参照

関連項目

return ステートメント
Function オブジェクト
caller プロパティ

JS1019: 'break' をループの外に設定できません。

break キーワードがループの外側にあります。**break** キーワードは、ループまたは **switch** ステートメントを終了するのに使用します。このキーワードは、ループまたは **switch** ステートメントの中に記述する必要があります。

このエラーを解決するには

- **break** キーワードが、ループまたは switch ステートメントの内側にあることを確認します。

参照

関連項目

break ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript の条件構造

JScript リファレンス

JS1020: 'continue' をループの外に設定できません。

continue ステートメントがループの外側にあります。**continue** ステートメントは、次に示すループの内側でだけ使用できます。

- **do-while** ループ
- **while** ループ
- **for** ループ
- **for/in** ループ

このエラーを解決するには

- **continue** ステートメントが、次に示すループの内側でだけ使用されていることを確認します。
    - **do-while** ループ
    - **while** ループ
    - **for** ループ
    - **for/in** ループ

参照

関連項目

continue ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript の条件構造

JScript リファレンス

JS1023: 16 進数の数字が必要です。

コードに不正な Unicode エスケープ シーケンスがあるか、16 進数リテラルの先頭に 16 進数以外の文字があります。

Unicode エスケープ シーケンスは、\u の後に 4 桁の 16 進数を続けて表します。16 進数リテラルは、0x の後に 16 進数の数字を続けて表します。16 進数には、数字の 0 ～ 9 と、大文字の A ～ F または小文字の a ～ f を使用できます。Unicode エスケープ シーケンスの正しい使用例を次に示します。

```
z = "\u1A5F";
```

次の例は、正しい 16 進数リテラルを示しています。

```
k = 0x3E8;
```

このエラーを解決するには

1. 16 進数には 0 ～ 9、大文字の A ～F、および小文字の a ～ f だけを使用します。
2. Unicode エスケープ シーケンスに 4 桁の 16 進数が含まれるようにします。

メモ：
文字列でリテラル テキスト \u を使用する場合は、円記号を 2 つ使用して (\\u)、\u の円記号をエスケープします。

参照

その他の技術情報

JScript リファレンス
データ型 (JScript)

JS1024: while が必要です。

do ... while ループに、**while** 条件がありません。**do** ステートメントでは、対応する **while** テスト ステートメントをコード ブロックの最後に指定する必要があります。

このエラーを解決するには

- ブロックの終わりを表す右中かっこの後ろに **while** テスト ステートメントを追加します。

参照

関連項目

while ステートメント

その他の技術情報

JScript の条件構造

JScript リファレンス

JS1025: この Label は既に定義されています。

新しいラベルで、既存のラベルの名前が使用されています。特定のスコープ内では、ラベルを一意にする必要があります。

このエラーを解決するには

- すべてのラベル名が、それぞれのスコープ内で一意となるようにします。

参照

関連項目

ラベル付きステートメント
switch ステートメント
break ステートメント
continue ステートメント

JS1026: この Label が定義されていません。

存在しないラベルを参照しています。特定のスコープ内では、ラベルを一意にする必要があります。

このエラーを解決するには

1. ラベル名のスペルを確認します。
2. すべてのラベル参照が、現在のスコープ内で定義されているラベルを参照していることを確認します。これには、スコープ内の前方にある定義も含みます。

参照

関連項目

ラベル付きステートメント
switch ステートメント
break ステートメント
continue ステートメント

JS1027: 'default' は 'switch' ステートメントのなかに、一度のみ表示できます。

switch ステートメントで、**default** case ステートメントが 2 回以上使用されています。default case は、常に switch ステートメントの最後の case ステートメント (式の値と一致しなかった場合に実行される case) である必要があります。

このエラーを解決するには

- switch ステートメントで、default case ステートメントを一度だけ使用するようにします。

参照

関連項目

switch ステートメント

概念

JScript の予約語

その他の技術情報

JScript の条件構造

JS1028: 識別子または文字列が必要です。

オブジェクト リテラルの宣言で、リテラル構文が不正です。オブジェクト リテラルのプロパティは、識別子または文字列である必要があります。オブジェクト リテラル (「オブジェクト初期化子」とも呼びます) は、"プロパティ : 値のペア"の形式で記述し、カンマ区切りのリストにして、全体をかっこで囲んで指定します。次に例を示します。

```
var point = {x:1.2, y:-3.4};
```

このエラーを解決するには

- 正しいリテラル構文を使用します。

参照

関連項目

コンマ演算子 (,)

JS1029: '@end' が必要です。

条件付きコンパイルされるコード ブロックが、**@end** ステートメントで終了していません。JScript ステートメントに条件付きコンパイルを実行するには、**@if/@end** ブロック内に記述する必要があります。

このエラーを解決するには

- 対応する **@end** ステートメントを追加します。

参照

関連項目

@if...@elif...@else...@end ステートメント

概念

条件付きコンパイル変数

その他の技術情報

条件付きコンパイル

JS1030: 条件付きコンパイルは無効になっています。

コードで条件付きコンパイル変数が使用されていますが、条件付きコンパイルが有効になっていません。条件付きコンパイルをオンにすると、JScript コンパイラでは、@ で始まる識別子を条件付きコンパイル変数としてコンパイルします。条件付きコンパイルを指定するには、次に示すステートメントで条件付きコードを開始します。

```
/*@cc_on @*/
```

このエラーを解決するには

- 条件付きコードの先頭に、次のステートメントを追加します。

  ```
  /*@cc_on @*/
  ```

参照

関連項目

@cc_on ステートメント
@if...@elif...@else...@end ステートメント
@set ステートメント

概念

条件付きコンパイル変数

その他の技術情報

条件付きコンパイル

JS1031: 定数が必要です。

@if (条件付きコンパイル) テスト ステートメントに変数があります。条件付きコンパイル テスト ステートメントで使用できるのは、リテラルと条件コンパイル変数 (コンパイル時にはどちらも定数) だけです。

このエラーを解決するには

1. 変数をリテラルに変更します。
2. 変数を条件付きコンパイル変数に変更します。

参照

関連項目

@cc_on ステートメント
@if...@elif...@else...@end ステートメント
@set ステートメント

概念

条件付きコンパイル変数

その他の技術情報

条件付きコンパイル

JS1032: '@' が必要です。

@set ステートメントに、条件付きコンパイル ステートメントで使用する変数が使用されていますが、変数名の先頭に "@" 記号がありません。

このエラーを解決するには

- 変数名の先頭に "@" 記号を追加します。次に例を示します。

```
@set @myvar = 1
```

参照

関連項目

@set ステートメント

概念

条件付きコンパイル変数

その他の技術情報

条件付きコンパイル

JS1033: catch が必要です。

例外処理の **try** ブロックに、対応する **catch** ステートメントがありません。例外処理を正しく機能させるには、エラーが発生する可能性のあるコードと、エラーが発生したときに実行しないコードを **try** ブロックの内側に記述する必要があります。例外は、**try** ブロック内の **throw** ステートメントによってスローされ、**try** ブロックの外側にある 1 つ以上の **catch** ステートメントでキャッチされます。

このエラーを解決するには

1. 対応する **catch** ブロックを追加します。
2. **catch** ブロックの代わりに **finally** ブロックを使用します。

参照

関連項目

try...catch...finally ステートメント
Error オブジェクト

JS1034: 一致しない 'else' です。'if' が定義されていません。

else ステートメントに対応する **if** ステートメントがありません。**if** ステートメントの後には、1 つのステートメントか複合ステートメントを指定します。その後に、**else** ステートメントを必要に応じて指定します。**else** ステートメントはこれ以外の場所では使用しません。

複合ステートメントは、明示的に中かっこで囲みます。コンパイラはタブ規則を無視します。タブは、コードを読みやすくするために使用されています。

このエラーを解決するには

1. **if** ステートメントに続くコードをかっこで囲みます。
2. **else** ステートメントの前に **if** ステートメントを追加します。

参照

関連項目

if...else ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1100: ',' が必要です。

関数宣言において、パラメータの間にカンマがありません。

このエラーを解決するには

- 関数宣言で、各パラメータをカンマで区切ります。

参照

関連項目

function ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1101: 可視性修飾子は既に定義されています。

1 つの式に可視性修飾子を 2 回以上適用しています。修飾子を繰り返し指定しても効果はありません。

このエラーを解決するには

- 適用する可視性修飾子の数は、1 つだけにします。

参照

概念

JScript スクリプトのトラブルシューティング

JScript の修飾子

その他の技術情報

JScript リファレンス

JS1102: 無効な可視性修飾子です。

不適切なコンテキストで可視性修飾子が使用されています。このエラーは、修飾子を使用できない式に修飾子を適用した場合や、既に修飾子が指定されているクラスやインターフェイスのメンバに互換性のない修飾子を使用した場合に発生します。

このエラーを解決するには

1. 可視性修飾子を削除するか、別の修飾子を使用します。
2. クラス メンバに適用されている可視性修飾子が、クラスの可視性と一致していることを確認します。

参照

概念

JScript スクリプトのトラブルシューティング

JScript の修飾子

その他の技術情報

JScript リファレンス

JS1103: 'case' または 'default' ステートメントが見つかりません。

switch ステートメント ブロック内の 1 行目が、**case** または **default** キーワードで開始されていません。

このエラーを解決するには

- **switch** ステートメントの後に、**case** または **default** キーワードを追加します。

参照

関連項目

switch ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1104: '@end' に対応する '@if' が定義されていません。

@end ステートメントに対応する **@if** ステートメントがありません。**@end** ステートメントは **@if**...**@end** ブロックを終了します。したがって、**@if** ステートメントを使用する場合は、対応する **@end** ステートメントが必要です。同様に、**@end** ステートメントには対応する **@if** ステートメントが必要です。

このエラーを解決するには

- 各 **@end** ステートメントの前に **@if ステートメント**があることを確認します。

参照

関連項目

@if...@elif...@else...@end ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1105: '@else' に対応する '@if' が定義されていません。

@else ステートメントに対応する **@if** ステートメントがありません。**@else** ステートメントは、**@if**...**@end** ブロックの内部でだけ使用できます。

このエラーを解決するには

- **@else** ステートメントが、**@if**...**@end** ブロックの内部でだけ使用されていることを確認します。

参照

関連項目

@if...@elif...@else...@end ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1106: '@elif' に対応する '@if' が定義されていません。

@elif ステートメントに対応する **@if** ステートメントがありません。**@elif** ステートメントは、**@if**...**@end** ブロックの内部でだけ使用できます。

このエラーを解決するには

- **@elif** ステートメントが、**@if**...**@end** ブロックの内部でだけ使用されていることを確認します。

参照

関連項目

@if...@elif...@else...@end ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1107: ソース文字が足りません。

コード行の終わりに達しましたが、式は終了していません。このエラーは、式の開始と終了を同じ行で行う必要があるときに、行を 2 行以上に分割した場合に発生します。

このエラーを解決するには

- 式を開始した行と同じ行で、式を終了します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1108: 互換性のない可視性修飾子です。

可視性修飾子が、この修飾子を使用できないクラスやインターフェイス、または互換性のない修飾子が指定されているクラスやインターフェイスのメンバに対して使用されています。

このエラーを解決するには

1. 可視性修飾子を削除するか、別の修飾子を使用します。
2. クラス メンバに適用されている可視性修飾子が、クラスの可視性と一致していることを確認します。

参照

概念

JScript スクリプトのトラブルシューティング
JScript の修飾子

その他の技術情報

JScript リファレンス
クラス ベースのオブジェクト

JS1109: このコンテキストでクラスを定義することはできません。

クラスの宣言が、不適切なコンテキストで使用されています。クラスの宣言は、メイン プログラム ブロック、ほかのクラスの内部、パッケージ内、または関数内だけで許可されています。

このエラーを解決するには

- クラスをメイン プログラム ブロック、ほかのクラスの内部、パッケージの内部、または関数内で定義します。

参照

関連項目

class ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1110: 式はコンパイル時の定数でなければなりません。

コンパイル時の定数以外は指定できないコンテキストで、コンパイル時に未定義の式が使用されています。

このエラーを解決するには

- コンパイル時の定数だけを使用します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1111: 識別子は既に使用中です。

1 つの式で、同じ識別子名が繰り返し使用されています。

このエラーを解決するには

1. どちらのインスタンスも同じ識別子を参照している場合は、識別子の 2 番目の定義を削除します。
2. 識別子が異なる場合は、それぞれに一意の名前を指定します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript の変数と定数

JScript リファレンス

JS1112: 型名が必要です。

型名のスペルが間違っているか、型名が指定されていません。変数、定数、関数、またはパラメータの定義では、コロンの後に型名が必要です。

このエラーを解決するには

- 型名が必要な場所で、有効な型名の識別子が使用されていることを確認します。

参照

概念

型の注釈

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1113: クラス定義内部でのみ有効です。

クラス定義の内部だけで有効なディレクティブまたはキーワードが、クラス定義以外の場所にあります。

このエラーを解決するには

1. ディレクティブまたはキーワードを削除します。
2. コードがクラス定義の内部にあることを確認します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

クラス ベースのオブジェクト

JScript リファレンス

JS1114: 不明な position ディレクティブです。

position ディレクティブに無効な引数が渡されました。有効な引数は、**end**、**file=**、**line=**、および **column=** です。

このエラーを解決するには

- position ディレクティブに、有効な引数だけが使用されていることを確認します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

ディレクティブ

JS1115: 同じ行で、ディレクティブの後にほかのコードを記述することはできません。

ディレクティブの後に別のコードがあります。

このエラーを解決するには

- ディレクティブの後で、行を 2 つに分割します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

ディレクティブ

JS1118: デバッグ ディレクティブが無効か、またはディレクティブの位置が正しくありません。

@debug ディレクティブに無効な引数が渡されました。有効な引数は **on** と **off** です。

または

コードに指定できないディレクティブがあります。

このエラーを解決するには

1. **@debug** ディレクティブに渡される引数が、**on** または **off** であることを確認します。
2. または
3. ディレクティブを別の場所に移動します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

ディレクティブ

JS1119: ディレクティブを入れ子にすることはできません。

@position ディレクティブが入れ子になっています。

このエラーを解決するには

- ディレクティブを終了してから、別の **@position** ディレクティブを開始します。

参照

関連項目

@position ディレクティブ

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1120: 循環定義

クラスまたはインターフェイスが、それ自体の表現で定義されています。このエラーは、2 つのクラスがあり、それぞれが他方を拡張している場合に発生します。

このエラーを解決するには

- クラスまたはインターフェイスを拡張するときに、基本クラスまたは基本インターフェイスが、定義しようとしているクラスまたはインターフェイスに依存しないことを確認します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

クラス ベースのオブジェクト

JScript リファレンス

JS1121: 使用しないでください。

コードに、現在使用が推奨されていない式が含まれています。別の式で同じタスクを実行できます。新しい方法を使用してください。この言語の今後のバージョンでは、現在使用が推奨されていない式は削除される可能性があります。

このエラーを解決するには

1. escape メソッドの代わりに encodeURI メソッドを使用します。
2. getYear メソッドの代わりに getFullYear メソッドを使用します。
3. setYear メソッドの代わりに setFullYear メソッドを使用します。
4. substr メソッドの代わりに substring メソッドを使用します。
5. toGMTString メソッドの代わりに toUTCString メソッドを使用します。
6. unescape メソッドの代わりに decodeURI メソッドを使用します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1122: 現在のコンテキストで 'this' を使用することはできません。

static クラスまたはメンバ関数で、**this** が使用されています。

このエラーを解決するには

1. クラスの特定のインスタンスを参照する完全限定名で、**this** を置き換えます。
2. **static** 修飾子を削除します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

クラス ベースのオブジェクト

JScript リファレンス

JS1123: このスコープからアクセスできません。

オブジェクトのメンバにアクセスしようとしていますが、メンバに可視性修飾子が指定されているため、現在のスコープからはアクセスできません。たとえば、プライベート フィールドにはクラスの外部からはアクセスできません。

このエラーを解決するには

1. パブリック メソッドを使用するなど、ほかの方法でメンバにアクセスします。
2. アクセスするメンバの修飾子を変更します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

クラス ベースのオブジェクト

JScript リファレンス

JS1124: コンストラクタ関数に対してのみ、含まれるクラスと同じ名前を指定することができます。

コンストラクタ以外のメソッドの名前が、クラスの名前と同じです。

このエラーを解決するには

1. 戻り値の型と **return** ステートメントを削除して、メソッドをコンストラクタにします。
2. メソッドの名前を変更します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

クラス ベースのオブジェクト

JScript リファレンス

JS1128: クラスで実装を指定しなければなりません。

final メソッドまたは **final** クラスのメソッドに、関連付けられた本体がありません。

このエラーを解決するには

1. メソッドの本体を指定します。
2. **final** 修飾子を削除します。

参照

概念

JScript の修飾子
JScript スクリプトのトラブルシューティング

その他の技術情報

クラス ベースのオブジェクト
JScript リファレンス

JS1129: インターフェイス名が必要です。

定義されたクラスが、存在していないインターフェイスを実装しています。

このエラーを解決するには

- クラス定義で、**implements** キーワードの後に、有効なインターフェイスの名前を指定します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

クラス ベースのオブジェクト

JScript リファレンス

JS1133: Catch 句に到達することはありません。

すべてのエラーをキャッチする **catch** ブロックの後に、別の **catch** ブロックがあります。ステートメントの引数に特定の型を指定しない場合、**catch** ステートメントはすべてのエラーをキャッチします。

このエラーを解決するには

- すべての **catch** ステートメントを調べ、各 **catch** ステートメントが各型を一度だけキャッチし、最後の **catch** ステートメントが型なしのエラーをキャッチすることを確認します。

参照

関連項目

try...catch...finally ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1134: 型を拡張できません。

final 修飾子を持つ型またはクラスを拡張しようとしています。

このエラーを解決するには

1. 型またはクラスを拡張しないようにします。
2. クラスから **final** 修飾子を削除します。

参照

概念

JScript の修飾子

JScript スクリプトのトラブルシューティング

その他の技術情報

クラス ベースのオブジェクト

JScript リファレンス

JS1135: 変数が宣言されていません。

var ステートメントを使用して定義されていない変数、または名前のスペルを間違っている変数が式に含まれていて、コードが高速モードでコンパイルされています。高速モードでコンパイルされるプログラムでは、すべての変数が明示的に宣言されている必要があります。

コマンド ライン コンパイラでコンパイルされるプログラムでは、高速モードをオフにできます。

このエラーを解決するには

1. すべての変数を定義します。
2. すべての識別子のスペルを正しく指定します。
3. **/fast-** オプションを指定し、高速モードをオフにしてプログラムをコンパイルします。この方法は、コマンド ライン コンパイルを行う場合だけ有効です。

参照

関連項目

/fast

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript の変数と定数

JScript リファレンス

JS1136: 変数を初期化しないでおくことは、安全ではありません。変数を使用する場合の速度が低下します。

変数を初期化しないでおくことは、安全ではありません。変数を使用する場合の速度が低下します。この変数を未初期化のままにしますか?

var ステートメントを使って定義された変数に、特定の型が指定されていません。または、この変数が使用される前に初期化されていません。

このエラーを解決するには

1. 変数に型の注釈を使用します。
2. すべての変数を使用する前に初期化します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript の変数と定数

JScript リファレンス

JS1137: これは新しい予約語です。識別子として使うことはできません。

識別子の名前として、新しい予約語が使用されています。

このエラーを解決するには

- 識別子の名前を予約語以外に変更します。

参照

概念

JScript の予約語

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1140: ベース クラス コンストラクタへの呼び出しでは許可されていません。

基本クラスのコンストラクタに、現在のクラスのプロパティを渡そうとしました。現在のクラスのプロパティは、基本クラスが生成される時点までは存在しないため、この操作は認められません。

このエラーを解決するには

- 生成されるクラスのプロパティが、基本クラスのコンストラクタに渡されないことを確認します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

クラス ベースのオブジェクト

JScript リファレンス

JS1141: このコンストラクタまたはプロパティの getter/setter メソッドを直接呼び出すことはできません。

クラスのコンストラクタ メソッドが、直接呼び出されました。

- または

プロパティの getter メソッドまたは setter メソッドが直接呼び出されました。

- どちらのメソッドも直接呼び出すことはできません。

このエラーを解決するには

1. コンストラクタ メソッドを呼び出さないようにします。

   または

2. "." 構文を使用してプロパティにアクセスします。

参照

関連項目

function get ステートメント
function set ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス
クラス ベースのオブジェクト

JS1142: このプロパティの get および set メソッドは一致していません。

プロパティの **get** および **set** アクセサが定義されています。**get** アクセサの戻り値の型と **set** アクセサの引数の型が同じでありません。

このエラーを解決するには

- **get** アクセサの戻り値の型が、**set** アクセサの引数の型と一致することを確認します。

参照

関連項目

function get ステートメント
function set ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1143: カスタム属性クラスは、System.Attribute から派生しなければなりません。

カスタム属性として使用されているクラスが、**System.Attribute** から派生していません。カスタム属性として使用できるのは、**System.Attribute** を基本クラスとするクラスだけです。

このエラーを解決するには

- カスタム属性クラスの基本クラスが **System.Attribute** であることを確認します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1144: カスタム属性では、primitive 型のみを使用できます。

カスタム属性のコンストラクタにプリミティブ型以外のデータを渡そうとしたか、属性を使用する場所で属性以外のデータが使用されています。

このエラーを解決するには

- 式で属性が使用されていること、およびカスタム属性のコンストラクタにはプリミティブ型だけが渡されることを確認します。

参照

概念

カスタム属性の記述

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1146: 不明なカスタム属性クラスまたはコンストラクタです。

属性を使用する場所に、属性またはカスタム属性のコンストラクタが使用されていません。

このエラーを解決するには

- 属性またはカスタム属性のコンストラクタが、そのコンテキストで使用されていることを確認します。

参照

概念

カスタム属性の記述

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1148: 引数が多すぎます。

引数が多すぎます。余分な引数は無視されます。

関数またはメソッドに、定義で指定された数よりも多くの引数が渡されています。

このエラーを解決するには

1. 関数またはメソッドに正しい数の引数を渡します。
2. 余分な引数がないことを確認します。

参照

関連項目

function ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1149: With ステートメントによって、この名前の使い方があいまいになりました。

現在のスコープで識別子と同じ名前のメンバを持つクラスに、**with** ステートメントを使用してアクセスしています。コンパイラは、アクセスする識別子を区別できません。

このエラーを解決するには

1. クラスのメンバまたは現在のスコープの識別子の名前を変更します。
2. **with** ステートメントの使用を避けます。

参照

関連項目

with ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1150: eval によって、この名前の使われ方があいまいになりました。

eval ステートメントを使用しているプログラムが、高速モードをオフにしてコンパイルされています。

高速モードがオフの場合、**eval** ステートメントによって、実行時にローカル スコープで新しい変数が宣言されることがあります。これらの新しい変数はグローバル変数を隠ぺいすることがあるため、ローカル スコープで明示的に定義されていない変数への参照があいまいになる可能性があります。

このエラーを解決するには

1. **fast** オプションを指定してコンパイルします。
2. **eval** ステートメントの使用を避けます。
3. 現在のローカル スコープで使用できる変数だけを使用します。

参照

関連項目

eval メソッド (JScript)

/fast

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1151: オブジェクトにそのようなメンバは含まれていません。

クラス ベースのオブジェクトのメンバを参照しましたが、指定された名前のメンバがオブジェクトにありません。

コードが ASP.NET ページに含まれる場合、コンストラクタ関数は、**this** ステートメントを使用する **<script runat="server">** ブロック内で定義されます。**expando** 修飾子は、**<script runat="server">** ブロック内のすべてのコンストラクタ関数定義に対して指定する必要があります。

たとえば、次の ASP.NET ページのコードでは、**expando** 修飾子を指定されたコンストラクタ関数が使用されています。

```
<script runat="server">
expando function Person(name) {
   // If the expando modifier was not applied to the definition of Person,
   // the this statment in the following line of code would generate error
   // JS1151

   this.name = name;
}
</script>

<%
var fred = new Person("Fred");
Response.Write(fred.name);
%>
```

このエラーを解決するには

1. 式がクラス ベースのオブジェクトの既存のメンバを参照していること、およびメンバ名のスペルが正しいことを確認します。
2. **<script runat="server">** ブロック内の各コンストラクタ関数宣言に **expando** 修飾子を指定します。

参照

関連項目

this ステートメント
expando 修飾子

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス
クラス ベースのオブジェクト

JS1152: Expando クラスでプロパティ項目を定義できません。

Expando クラスでプロパティ項目を定義できません。項目は、expando フィールドで予約されています。

expando クラスに **Item** という名前のメンバが存在します。expando クラスで暗黙的に定義される **Item** プロパティと競合するため、これは許可されません。

このエラーを解決するには

- クラス メンバ **Item** の名前を変更します。

参照

関連項目

expando 修飾子

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

クラス ベースのオブジェクト

JS1153: Expando クラスで get_Item または set_Item を定義できません。

Expando クラスで get_Item または set_Item を定義できません。メソッドは expando フィールドで予約されています。

expando クラスに、**get_Item** または **set_Item** という名前のメンバが存在します。expando クラスで暗黙的に定義される **get_Item** または **set_Item** プロパティと競合するため、これは許可されません。

このエラーを解決するには

- クラス メンバ **get_Item** または **set_Item** の名前を変更します。

参照

関連項目

expando 修飾子

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

クラス ベースのオブジェクト

JS1155: ベース クラスは get_Item または set_Item を定義しますが、expando クラスを作成することはできません。

ベース クラスは get_Item または set_Item を定義しますが、expando クラスを作成することはできません。メソッドは expando フィールドで予約されています。

expando クラスが、**get_Item** または **set_Item** という名前のメンバを持つ基本クラスを拡張しています。expando クラスで暗黙的に定義される **get_Item** または **set_Item** プロパティと競合するため、これは許可されません。

このエラーを解決するには

1. 基本クラス メンバの **get_Item** または **set_Item** の名前を変更します。
2. 基本クラスから継承しないようにします。
3. クラス定義から **expando** 修飾子を削除します。

参照

関連項目

expando 修飾子

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

クラス ベースのオブジェクト

JS1156: ベース クラスは既に expando に設定されています。現在の仕様は無視されます。

expando 基本クラスを拡張するクラスに、**expando** 修飾子を指定しています。expando 基本クラスから派生するクラスは、自動的に expando クラスになります。**expando** 修飾子を明示的に指定する必要はありません。

このエラーを解決するには

- 派生クラスから **expando** 修飾子を削除します。

参照

関連項目

expando 修飾子

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

クラス ベースのオブジェクト

JS1157: abstract メソッドをプライベートにすることはできません。

メソッドに **abstract** 修飾子と **private** 修飾子の両方が指定されています。プライベート メソッドはクラス内だけでアクセス可能ですが、抽象メソッドはクラス外部から継承されるため、これは許可されません。

このエラーを解決するには

- メソッドの宣言から **abstract** 修飾子と **private** 修飾子のいずれかを削除します。

参照

関連項目

abstract 修飾子
private 修飾子

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス
クラス ベースのオブジェクト

JS1158: この型のオブジェクトをインデックス化することはできません。

オブジェクトの要素をインデックスで指定しようとしましたが、このオブジェクトのデータ型はインデックスによる指定をサポートしていません。

配列や JScript オブジェクトなど、インデックス指定の可能なオブジェクト要素には、[] 表記を使用してアクセスします。

このエラーを解決するには

1. オブジェクトのデータ型を変更します。
2. インデックス アクセサを削除します。

参照

概念

JScript スクリプトのトラブルシューティング

型指定された配列

その他の技術情報

JScript リファレンス

JScript オブジェクト

JS1159: 構文エラーです。クラス初期化子を定義するには、'static classname {...}' を使用してください。

コード ブロックは、クラス内部の修飾子の直後に指定します。JScript の修飾子は、クラス メンバだけに適用できます。このエラーの原因として、次の 2 つが考えられます。

- クラス初期化子の定義で、クラス名が指定されていません。
- メソッドまたはプロパティ アクセサの定義で、**function**、**function get**、または **function set** ステートメントが指定されていません。

このエラーを解決するには

1. クラス初期化子を定義する場合は、**static** ステートメントの正しい構文を使用します。
2. メソッドまたはプロパティ アクセサを定義する場合は、**function**、**function get**、または **function set** ステートメントの正しい構文を使用します。

参照

関連項目

static ステートメント
function ステートメント
function get ステートメント
function set ステートメント

概念

JScript スクリプトのトラブルシューティング
JScript の修飾子

その他の技術情報

JScript リファレンス

JS1160: 属性の一覧は現在のコンテキストに適用しません。

現在のコンテキストに適用できない属性の一覧が指定されています。

このエラーを解決するには

- 現在のコンテキストに適用できる属性だけを使用します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1161: パッケージ内で使用できるのはクラスのみです。

関数、変数、または定数をパッケージの内部で宣言しました。パッケージで定義できるのはクラスとインターフェイスだけです。

このエラーを解決するには

- パッケージからクラスとインターフェイス以外の定義をすべて削除します。

参照

関連項目

package ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JScript オブジェクト

JS1162: Expando クラスで IEnumerable をインプリメントすることはできません。

Expando クラスで IEnumerable を実装することはできません。インターフェイスは expando クラスで暗黙的に定義されています。

expando クラスで **IEnumerable** を明示的に実装しています。**expando** 修飾子を指定したクラスは暗黙的に **IEnumerable** を実装するため、この実装は不要です。

このエラーを解決するには

- **IEnumerable** を明示的に実装しないようにします。

参照

関連項目

expando 修飾子

IEnumerable Interface

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

クラス ベースのオブジェクト

JS1163: 指定されたメンバは CLS 準拠ではありません。

プログラムに **CLSCompliantAttribute** 属性が含まれていますが、コンパイラが共通言語仕様 (CLS: Common Language Specification) 準拠でないクラス メンバを検出しました。このエラーの原因としては、次のようなことが考えられます。

- メンバ名が CLS 準拠でありません。CLS 準拠の名前では、名前をアンダースコア (_) で開始したり、名前にドル記号 ($) を含めることはできません。また、ほかのパブリック メンバと、大文字小文字の使用だけが異なる名前も使用できません。
- メンバがパブリック メソッドである場合は、パラメータまたは戻り値の型に CLS で使用できないデータ型が指定されています。
- メンバがフィールドまたはプロパティの場合は、フィールドまたはプロパティの型に CLS で使用できないデータ型が指定されています。

データ型を CLS で利用できない理由はいくつかあります。

- 型がクラス内で定義されていて、パブリックにアクセスできない。
- 型が定義されているが、CLS 準拠として指定されていない。
- 型が CLS 準拠ではないプリミティブ型である。たとえば、**uint** は CLS 準拠ではないプリミティブ型です。対応する CLS 準拠のシステム型は **System.UInt32** です。
- 型が組み込みの JScript オブジェクトである。組み込みの JScript オブジェクトは、いずれも CLS 準拠ではありません。一般的に使用される JScript オブジェクトの **Array**、**Date**、**Error**、**RegExp**、および **Function** は、それぞれ CLS 準拠のシステム型 **System.Array**、**System.DateTime**、**System.Exception**、**System.Text.RegularExpressions.RegEx**、および **System.EventHandler** に対応します。

このエラーを解決するには

1. メンバの名前がアンダースコアで開始されていたり、ドル記号 ($) を含んでいたりしないことを確認します。また、ほかのメンバ名との違いが、大文字小文字の違い以外にもあることを確認します。
2. パブリック メソッドのパラメータまたは戻り値の型、およびパブリック フィールドおよびパブリック プロパティの型が、共通言語ランタイムのデータ型であるか、CLS 準拠に指定されているパブリックにアクセス可能なクラスであることを確認します。

参照

関連項目

CLSCompliantAttribute Class

概念

JScript スクリプトのトラブルシューティング

CLS 準拠コードの記述

共通言語仕様

その他の技術情報

JScript リファレンス

JS1164: メンバを削除できません。

削除できないオブジェクトのメンバを削除しようとしました。削除できるのは expando プロパティ (オブジェクトに動的に追加されたプロパティ) だけです。

このエラーを解決するには

- オブジェクトのメンバを削除しないようにします。

参照

関連項目

delete 演算子
expando 修飾子

概念

JScript スクリプトのトラブルシューティング
JScript の Object オブジェクト

その他の技術情報

JScript リファレンス

JS1165: パッケージ名が必要です。

import ステートメントの後に、有効なパッケージ名が指定されていません。

このエラーを解決するには

- **import** ステートメントの後に、有効なパッケージ名を指定します。

参照

関連項目

import ステートメント
package ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1169: 式は実行されません。

使用されない値を返す式があります。

このエラーを解決するには

1. 式を削除します。
2. 式の値を関数または演算子の引数に使用します。

参照

概念

JScript スクリプトのトラブルシューティング

JScript の式

その他の技術情報

JScript リファレンス

JS1170: ベース クラスで宣言されたほかのメンバを非表示にします。

派生クラスのメンバが、基本クラスで定義されたフィールド、クラス、インターフェイス、または列挙を隠ぺいし、これらの意味を再定義しています。隠ぺいできるのは、メソッドとプロパティだけです。

このエラーを解決するには

- 派生クラスのメンバが、基本クラスのフィールド、クラス、インターフェイス、または列挙の定義を隠ぺいしないことを確認します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

クラス ベースのオブジェクト

JS1171: ベース メソッドの可視性仕様を変更できません。

派生クラスに、基本クラスのメソッドをオーバーライドするメソッドがあり、これらの 2 つのメソッドの可視性修飾子が異なっています。基本クラスのメンバの可視性は変更できません。

このエラーを解決するには

- 派生クラスの可視性修飾子を変更して、基本クラスのメソッドの参照可能範囲と一致させます。

参照

概念

JScript スクリプトのトラブルシューティング

JScript の修飾子

その他の技術情報

JScript リファレンス

JS1172: メソッドはベース クラスの抽象メソッドを非表示にします。

hide 修飾子を指定して定義された派生クラスのメソッドの名前と、基本クラスの抽象メソッドの名前が同じです。抽象メソッドは派生クラスによる実装を必要としますが、メソッドの隠ぺいにより実装が妨げられています。

このエラーを解決するには

1. 基本クラスのメソッドから **abstract** 修飾子を削除し、メソッドに実装を与えます。
2. 派生クラスのメソッドから **hide** 修飾子を削除します。

参照

概念

JScript スクリプトのトラブルシューティング

JScript の修飾子

その他の技術情報

JScript リファレンス

JS1173: メソッドがベース クラス内のメソッドと同じです。

メソッドがベース クラス内のメソッドと同じです。このメッセージを非表示にするには、'override' または 'hide' を指定してください。

バージョン セーフ修飾子を指定せずに定義した派生クラスのメソッドが、基本クラスのメソッドと一致しています。また、プログラムが **/versionsafe** オプションを指定してコンパイルされています。**/versionsafe** オプションを指定してコンパイルする場合、基本クラスのメソッドと一致するすべてのメソッドでバージョン セーフ修飾子 (**hide** または **override**) を使用する必要があります。

このエラーを解決するには

- メソッドの宣言で、適切なバージョン セーフ修飾子を指定します。

参照

関連項目

/versionsafe
hide 修飾子
override 修飾子

概念

JScript スクリプトのトラブルシューティング
JScript の修飾子

その他の技術情報

JScript リファレンス

JS1174: メソッドがオーバーライド不可能なメソッドとベース クラスで一致しています。

メソッドがオーバーライド不可能なメソッドとベース クラスで一致しています。このメッセージを非表示にするには 'hide' を指定してください。

final 修飾子を指定した基本クラスのメソッドが、派生クラスのメソッドと一致しています。また、派生クラスのメソッドに **override** 修飾子が指定されているか、コードが **/versionsafe** オプションを指定してコンパイルされています。final メソッドはオーバーライドできません。**/versionsafe** オプションを使用する場合は、派生クラスのメソッドに、明示的に **hide** 修飾子を指定する必要があります。

このエラーを解決するには

- 派生クラスのメソッドに **hide** 修飾子を使用します。

参照

関連項目

/versionsafe
hide 修飾子
override 修飾子
final 修飾子

概念

JScript スクリプトのトラブルシューティング
JScript の修飾子

その他の技術情報

JScript リファレンス

JS1175: ベース クラスには非表示にするメンバはありません。

派生クラスのメソッドに **hide** 修飾子が指定されていますが、基本クラスに対応するメソッドがありません。存在しないメソッドは隠ぺいできません。

このエラーを解決するには

1. メソッドの宣言から **hide** 修飾子を削除します。

   または

2. 基本クラスで、対応するメソッドを追加します。

参照

関連項目

hide 修飾子

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1176: ベースのメソッドに、異なる戻り値が指定されています。

派生クラスはインターフェイスを実装するか、基本クラスを拡張します。インターフェイスまたは基本クラスのメソッドと、同じ名前とパラメータ リストを持つメソッドが派生クラスにありますが、これらの戻り値の型が異なっています。2 つのメソッドの戻り値の型が異なる場合は、同じ名前とパラメータ リストを指定できません。

このエラーを解決するには

1. 派生クラスの関数の名前を変更します。
2. 戻り値の型またはメソッドを変更して、派生クラスのメソッドと、インターフェイスまたは基本クラスのメソッドが一致するようにします。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

クラス ベースのオブジェクト

JS1177: プロパティと競合しています。

派生クラスで定義されたフィールドまたはメソッドの名前が、基本クラスのプロパティと同じです。名前を使って派生クラスのメンバを参照するときにあいまいさが生じるため、これは認められていません。

このエラーを解決するには

- 基本クラスのプロパティと派生クラスのメンバのどちらかの名前を変更します。

参照

関連項目

function set ステートメント

function get ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1178: 'override' と 'hide' を同時に使うことはできません。

メソッドの定義で、**override** 修飾子と **hide** 修飾子の両方が指定されています。各クラス メンバには、バージョン セーフ修飾子は 1 つしか指定できません。

このエラーを解決するには

- メソッドの宣言から、**override** または **hide** のいずれかを削除します。

参照

関連項目

hide 修飾子

override 修飾子

概念

JScript スクリプトのトラブルシューティング

JScript の修飾子

その他の技術情報

JScript リファレンス

JS1179: 無効なオプション

@option ディレクティブを使用する式に、有効なオプションが指定されていません。**@option** ディレクティブは現在サポートされていません。

このエラーを解決するには

- **@option** ディレクティブを使用しないようにします。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1180: ベース クラスには、オーバーライドするメソッドはありません。

派生クラスのメソッドに **override** 修飾子が指定されていますが、対応するメソッドが基本クラスにありません。存在しないメソッドはオーバーライドできません。

このエラーを解決するには

1. メソッドの宣言から **override** 修飾子を削除します。

   または

2. 基本クラスで、対応するメソッドを追加します。

参照

関連項目

override 修飾子

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

クラス ベースのオブジェクト

JS1181: コンストラクタには有効ではありません。

クラス コンストラクタに無効な修飾子が指定されています。コンストラクタに適用できるのは、可視性修飾子 (**public**、**private**、**protected**、および **internal**)、バージョン セーフ修飾子 (**hide** および **override**)、または **final** 修飾子および **expando** 修飾子だけです。

このエラーを解決するには

- コンストラクタに有効な修飾子を使用します。

参照

概念

JScript スクリプトのトラブルシューティング

JScript の修飾子

その他の技術情報

JScript リファレンス

クラス ベースのオブジェクト

JS1182: コンストラクタまたは void 関数から値を返すことはできません。

コンストラクタまたは **void** 関数から値を返そうとしました。コンストラクタ関数は、作成された関数へのポインタを自動的に返し、値を返しません。戻り値の型が **void** で定義されている関数は、値を返しません。

このエラーを解決するには

1. return ステートメントを削除します。
2. 関数に **void** 以外の戻り値の型を指定します。

参照

関連項目

function ステートメント
return ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1183: 2 つ以上のメソッドまたはプロパティがこの引数リストに一致しています。

オーバーロードされたメソッドまたはプロパティを呼び出している式が、渡された引数の型と正確に一致する単一のメソッドまたはプロパティを見つけられません。コンパイラは、オーバーロードされた関数のうち、引数のデータ型の強制変換の回数が最も少なくて済む関数を決定しようとします。このエラーは、引数のデータ型の強制変換の数が同じ関数を、2 つ以上検出したことを示しています。

このエラーを解決するには

- オーバーロードされた関数が受け取るデータ型を確認し、引数のデータ型が、オーバーロードされた関数のいずれか 1 つだけに一致するようにします。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1184: 2 つ以上のコンストラクタがこの引数リストに一致しています。

オーバーロードされたコンストラクタを呼び出している式が、渡された引数の型と正確に一致する単一のコンストラクタを見つけられません。コンパイラは、オーバーロードされたコンストラクタのうち、引数のデータ型の強制変換の回数が最も少なくて済む関数を決定しようとします。このエラーは、引数の型の強制変換の回数が同じコンストラクタを、2 つ以上検出したことを表しています。

このエラーを解決するには

- オーバーロードされたコンストラクタが受け取るデータ型を確認し、引数のデータ型が、オーバーロードされたコンストラクタのいずれか 1 つだけに一致するようにします。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JScript リファレンス

JS1185: ベース クラス コンストラクタにこのスコープからアクセスすることはできません。

基本クラスを拡張したクラスがありますが、現在のスコープから基本クラスのコンストラクタにアクセスできません。このエラーは、基本クラスまたは基本クラスのコンストラクタに可視性修飾子が使用されている場合に発生します。

このエラーを解決するには

1. 基本クラスのコンストラクタに、別の可視性修飾子を使用します。
2. 基本クラスが **internal** 修飾子で定義されている場合、基本クラスと同じパッケージで派生クラスを定義します。

参照

概念

JScript スクリプトのトラブルシューティング

JScript の修飾子

その他の技術情報

JScript リファレンス

JS1186: 8 進数のリテラルは使用を避けてください。

数値を表す 8 進数のリテラルがコードにあります。8 進数のリテラルは、整数の前に 1 つ以上の 0 を指定して表します。8 進数リテラルの代わりに、10 進数リテラルまたは 16 進数リテラルを使用してください。

このエラーを解決するには

1. 数値が 10 進数の場合は、先頭の 0 を削除します。
2. 8 進数を 10 進数または 16 進数に変換します。

参照

概念

JScript スクリプトのトラブルシューティング

数値データ

その他の技術情報

JScript リファレンス

JS1187: 変数が初期化されていない可能性があります。

初期化も特定のデータ型としての定義もされていない変数の値にアクセスしています。

このエラーを解決するには

1. 変数を使用する前に初期化します。
2. 型の注釈を使用して変数を宣言します。

参照

関連項目

var ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1188: この場所からベース クラス コンストラクタを呼び出すことはできません。

コンストラクタ定義内の 1 行目以外の場所から、基本クラスのコンストラクタの **super** を呼び出しています。

このエラーを解決するには

- 基本クラスのコンストラクタが、コンストラクタ宣言内の 1 行目からだけ呼び出されていることを確認します。

参照

関連項目

super ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1189: この方法で super キーワードを使用することはできません。

静的クラス メンバに **super** ステートメントが使用されています。静的メンバはクラス自身に関連付けられますが、**super** ステートメントはクラスの現在のインスタンスの基本クラス メンバにアクセスするために使用されます。

このエラーを解決するには

1. 基本クラスの特定のインスタンスを参照する完全限定名で、**super** を置き換えます。
2. **static** 修飾子を削除します。

参照

関連項目

super ステートメント
static 修飾子

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1190: finally ブロックをこの状態で維持すると、パフォーマンスが低下し、エラーを発生させる可能性があります。

finally ブロックをこの状態で維持すると、パフォーマンスが低下し、エラーを発生させる可能性があります。よろしいですか?

プログラムの制御を **finally** ブロックから別の場所に移すステートメント (**return** または **break**) があります。**try** または **catch** ブロックに **return** または **break** ステートメントがあると、意図しない結果になる場合があります。

finally ブロックのコードは、常に **try** ブロックおよび (エラーがある場合は) **catch** ブロックのコードの後に実行されます。たとえば、**try** ブロックに **return** ステートメントがあると、**return** ステートメントが実行される前に **finally** ブロックが実行されます。finally ブロックに別の **return** ステートメントがあると、そのステートメントが実行され、try ブロックの return ステートメントは実行されません。このような状況を避けるために、**finally** ブロックには **return** ステートメントを使用しないでください。

```
function test() {
   try {
      return(5);   // Attempt to return 5.
   } catch(e) {
      print(e);
   } finally {
      return(10);  // This gets run first, returning 10 instead of 5.
   }
}
print(test());     // Prints 10, not 5.
```

このエラーを解決するには

1. **finally** ブロックに **return** ステートメントおよび **break** ステートメントが使用されていないことを確認します。
2. **return** または **break** ステートメントを **try** および **catch** ブロックの後に実行する場合は、これらを **finally** ブロックの直後に移動します。

参照

関連項目

try...catch...finally ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1191: ',' がありません。型指定されたパラメータを宣言するには、'Type identifier' よりも 'identifier : Type' を使用してください。

関数宣言に、カンマで区切られていないパラメータが含まれているか、`identifier : Type` ではなく `Type identifier` の形式で型の注釈を指定されたパラメータが含まれています。

このエラーを解決するには

1. すべてのパラメータがカンマで区切られていることを確認します。
2. パラメータの型の注釈を `identifier : Type` の形式で指定します。

参照

関連項目

function ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JScript の関数

JS1192: Abstract 関数に本体を指定することはできません。

関数本体がメソッドまたはプロパティに関連付けられていますが、メソッドまたはプロパティに **abstract** 修飾子が指定されているか、メソッドまたはプロパティがインターフェイスにあります。

このエラーを解決するには

1. 関数本体を削除します。
2. 修飾子を変更します。
3. インターフェイスの代わりにクラスを使用します。

参照

関連項目

class ステートメント
interface ステートメント

概念

JScript の修飾子
JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1193: ',' または ')' が必要です。

関数、メソッド、またはコンストラクタの呼び出しに、カンマまたは閉じかっこがありません。

このエラーを解決するには

- カンマまたは閉じかっこを追加します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JScript の関数

JScript オブジェクト

JS1194: ',' または ']' が必要です。

配列要素の参照で、カンマまたは閉じ角かっこがありません。

このエラーを解決するには

- カンマまたは閉じ角かっこを追加します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JScript の配列

JS1195: 式が必要です。

値または参照 (式の結果) が要求される場所に、式が指定されていません。値を返さないステートメントが使用されている可能性があります。

このエラーを解決するには

- 式が使用されていることを確認します。

参照

概念

JScript スクリプトのトラブルシューティング

JScript の式

その他の技術情報

JScript リファレンス

JS1196: ';' は無効です。

ここではセミコロンは使用できません。または、セミコロンで終了しているステートメントにエラーがあります。

このエラーを解決するには

1. セミコロンを削除します。
2. セミコロンで終了しているステートメントに、ほかのエラーがないことを確認します。

参照

概念

JScript スクリプトのトラブルシューティング

JScript のステートメント

その他の技術情報

JScript リファレンス

JS1197: エラーが多すぎます。

エラーが多すぎます。ファイルが JScript .NET ファイルでない可能性があります。

コードに多数のエラーが発生しています。このエラーの最も一般的な原因は、JScript .NET ファイルでないファイルをコンパイルしようとしていることです。または、コンパイラでは修復できないエラーが原因で、さらにほかのエラーが発生している場合もあります。

このエラーを解決するには

1. コンパイルしているファイルが JScript .NET ファイルであることを確認します。
2. 最初のいくつかのエラーを修正し、再コンパイルします。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1198: 構文エラーです。型指定された変数を宣言するには、'Type identifier' よりも 'var identifier : Type' を使用してください。

`Type identifier` という、C 形式のフィールド宣言があります。JScript でフィールドを宣言する場合は、`var identifier : Type` 構文を使用します。

このエラーを解決するには

- フィールドを `var identifier : Type` の構文で宣言します。

参照

関連項目

var ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

クラス ベースのオブジェクト

JS1199: 構文エラーです。型指定された関数を宣言するには、'Type identifier(...){' よりも 'function identifier(...) : Type{' を使用してください。

`Type identifier(...)` という、C 形式のメソッド宣言があります。JScript でメソッドを宣言する場合は、`function identifier(...) : Type` の構文を使用します。

このエラーを解決するには

- メソッドを `function identifier(...) : Type` の構文で宣言します。

参照

関連項目

function ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

クラス ベースのオブジェクト

JS1200: プロパティ宣言が無効です。

プロパティ宣言が無効です。getter に引数を指定することはできません。また、setter には 1 つの引数を指定する必要があります。

プロパティの getter 関数の定義で 1 つ以上のパラメータを指定しています。または、プロパティの setter 関数の定義でパラメータを指定していないか、複数のパラメータを指定しています。getter 関数の定義ではパラメータを指定できません。また、setter 関数ではパラメータを 1 つだけ指定する必要があります。

このエラーを解決するには

1. パラメータを指定せずにプロパティの getter 関数を定義します。
2. プロパティの setter 関数の定義で、パラメータを 1 つだけ指定します。

参照

関連項目

function get ステートメント
function set ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス
クラス ベースのオブジェクト

JS1203: 式にアドレスがありません。

コードで、アンパサンド (**&**) の後に、アドレスを持たない式が指定されています。アンパサンドは、参照によるパラメータを受け取る関数に、変数を参照で渡すためだけに使用します。アンパサンドを使用できるのは、(アドレスを持つ) 変数名の前だけです。

参照による変数の引き渡しにより、関数で変数の値を変更できます。

メモ :
JScript では、参照パラメータを使った関数を定義できません。JScript では、アンパサンドは、参照パラメータを受け取る外部オブジェクトを呼び出すために用意されています。

このエラーを解決するには

- 関数の呼び出しにおいてアンパサンド (**&**) を変数名の前に指定し、関数が参照によるパラメータを受け取るようにします。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1204: 必要な引数の一部が指定されていません。

関数またはメソッドが、定義されている数よりも少ない数の引数で呼び出されています。不足している引数の代わりに、関数またはメソッドの既定の値が使用されますが、定義されている引数すべてを指定することをお勧めします。

このエラーを解決するには

- 関数またはメソッドに渡す必要な引数をすべて指定します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1205: 割り当てによって、ただちに破棄される expando プロパティが作成されます。

コードに、オブジェクトに存在しないプロパティへの代入がありますが、そのオブジェクトで expando プロパティがサポートされていません。コンパイラは expando プロパティを作成しようとしますが、プロパティをオブジェクトに追加できないためプロパティは破棄されます。

このエラーを解決するには

1. expando プロパティをサポートするオブジェクトを使用します。
2. expando プロパティをサポートしないオブジェクトに、expando プロパティを追加しないようにします。

参照

概念

JScript スクリプトのトラブルシューティング
高度なクラス作成
JScript の Object オブジェクト

その他の技術情報

JScript リファレンス

JS1206: ここに代入式を書き込みますか?

条件付きステートメントの条件式として、代入演算子が使用されています。等価演算子または厳密等価演算子の間違いである可能性があります。

このエラーを解決するには

1. 代入演算子 (=) を等価演算子 (==) または厳密等価演算子 (===) に変更します。
2. 代入演算子を条件付きステートメントの直前に指定し、代入演算子の左辺のオペランドを条件式として使用します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JScript の条件構造

JS1207: If ステートメントのこの分岐に対して空のステートメントを指定しますか?

if ステートメントの後にセミコロンが指定されています。セミコロンは、**if** ステートメントの条件式が true の場合に実行される、空のステートメントの終端記号と解釈されます。

このエラーを解決するには

1. セミコロンを削除します。
2. **if** ステートメントの後に空のブロック ({}) を指定します。

参照

関連項目

if...else ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1208: 指定された conversion または coercion は、使用できません。

実行できない型変換または型の強制変換があります。変換元のデータ型と変換先のデータ型に、明らかな類似性がないことを示しています。

このエラーを解決するには

- 変換先のデータ型と互換性のあるデータ型が指定されていることを確認します。

参照

概念

JScript スクリプトのトラブルシューティング
型変換
JScript における型の強制変換

その他の技術情報

JScript リファレンス

JS1209: final と abstract を同時に使うことはできません。

クラスまたはクラス メンバに、**final** および **abstract** の両方の修飾子が指定されています。これらの修飾子は同時に指定できません。

このエラーを解決するには

- **final** または **abstract** のいずれかの修飾子を使用します。

参照

概念

JScript スクリプトのトラブルシューティング

JScript の修飾子

その他の技術情報

JScript リファレンス

JS1210: インスタンスでなければなりません。

クラス名を使用して、非静的なクラス メンバにアクセスしようとしています。クラス自身に関連付けることができるのは静的なクラス メンバだけです。非静的なメンバは、特定のクラス インスタンスに対して関連付けられ、そのインスタンスを通じてアクセスされます。

このエラーを解決するには

- 非静的なメンバが、クラス インスタンスを通じてアクセスされていることを確認します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

クラス ベースのオブジェクト

JS1212: 宣言するクラスが abstract に設定されている場合を除き、抽象にすることはできません。

abstract 修飾子が指定されたメンバがありますが、そのメンバが含まれているクラスに **abstract** が指定されていません。**abstract** メンバが 1 つでもあるクラスには、**abstract** として指定する必要があります。

このエラーを解決するには

- **abstract** メンバを持つすべてのクラスが、**abstract** として指定されていることを確認します。

参照

概念

JScript スクリプトのトラブルシューティング

JScript の修飾子

その他の技術情報

JScript リファレンス

JS1213: enum のベース型は、プリミティブの整数型でなければなりません。

列挙型の宣言で、基本型がプリミティブ整数型ではありません。列挙型の有効な基本型は、整数型の **int**、**short**、**long**、**byte**、**uint**、**ushort**、**ulong**、および **sbyte** です。

このエラーを解決するには

- 各列挙型の基本型が、有効な整数型であることを確認します。

参照

関連項目

enum ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1214: 抽象型クラスのインスタンスを構築することはできません。

new 演算子を使って、抽象クラスのインスタンスを生成しようとしています。**abstract** 修飾子が指定されているクラスは、インスタンス化できません。

このエラーを解決するには

1. クラスから **abstract** 修飾子を削除します。
2. 抽象クラスを拡張するクラスを定義し、すべての抽象メソッドおよび抽象プロパティに実装を指定します。
3. 抽象クラスをインスタンス化しないようにします。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1215: JScript Array を System.Array に変換すると、メモリが割り当てられて配列がコピーされます。

JScript の **Array** オブジェクトを、型指定された配列 (**System.Array**) に変換しています。

メモ：
この処理では、指定された配列のコピーを格納するための十分なメモリが割り当てられ、JScript 配列の要素が、型指定された配列にコピーされます。

したがって、型指定された配列への変更は、変更後に型指定された配列を JScript 配列にコピーし直さない限り、JScript 配列には反映されません。

このエラーを解決するには

- 明示的な型変換を使用して、JScript 配列を型指定された配列に変換します。

参照

概念

JScript スクリプトのトラブルシューティング

型変換

その他の技術情報

JScript リファレンス

JScript の配列

JS1216: スタティック メソッドを abstract にすることはできません。

メソッドに **static** 修飾子と **abstract** 修飾子の両方が指定されています。静的メソッドはクラス自身に関連付けられますが、抽象メソッドはクラス外部から継承されるため、この指定は認められません。

このエラーを解決するには

- メソッドの宣言から **static** 修飾子と **abstract** 修飾子のいずれかを削除します。

参照

概念

JScript スクリプトのトラブルシューティング

JScript の修飾子

その他の技術情報

JScript リファレンス

JS1217: スタティック メソッドを final にすることはできません。

メソッドに **static** 修飾子と **final** 修飾子の両方が指定されています。静的メソッドは最終メンバであるため、この指定は認められません。静的メソッドはクラス自身に関連付けられるので、オーバーライドできません。

このエラーを解決するには

- メソッドの宣言から **static** 修飾子と **final** 修飾子のいずれかを削除します。

参照

概念

JScript スクリプトのトラブルシューティング

JScript の修飾子

その他の技術情報

JScript リファレンス

JS1218: スタティック メソッドでベース クラス メソッドをオーバーライドすることはできません。

メソッドに **static** 修飾子と **override** 修飾子の両方が指定されています。静的メソッドは現在のクラスに関連付けられていて、オーバーライドはクラス インスタンスでだけ機能するため、これは許可されません。

このエラーを解決するには

- メソッドの宣言から **static** 修飾子と **override** 修飾子のいずれかを削除します。

参照

概念

JScript スクリプトのトラブルシューティング

JScript の修飾子

その他の技術情報

JScript リファレンス

JS1219: スタティック メソッドでベース クラス メソッドを非表示にすることはできません。

メソッドに **static** 修飾子と **hide** 修飾子の両方が指定されています。静的メソッドは現在のクラスに関連付けられていて、**hide** はクラス インスタンスに関連付けられたメンバにだけ適用されるため、この指定は認められません。

このエラーを解決するには

- メソッドの宣言から **static** 修飾子と **hide** 修飾子のいずれかを削除します。

参照

概念

JScript スクリプトのトラブルシューティング

JScript の修飾子

その他の技術情報

JScript リファレンス

JS1220: Expando メソッドでベース クラス メソッドをオーバーライドすることはできません。

メソッドに **expando** 修飾子と **override** 修飾子の両方が指定されています。expando メソッドは現在のクラスに関連付けられていて、**override** はクラス インスタンスに関連付けられたメンバにだけ適用されるため、この指定は認められません。

このエラーを解決するには

- メソッドの宣言から **expando** 修飾子と **override** 修飾子のいずれかを削除します。

参照

概念

JScript スクリプトのトラブルシューティング

JScript の修飾子

その他の技術情報

JScript リファレンス

JS1221: 変数の引数一覧は配列型でなければなりません。

関数の定義に、型指定された配列として、型の注釈が指定されていないパラメータ配列 (または可変個引数リスト) があります。パラメータ配列は、関数宣言の最後のパラメータとする必要があります。また、前に省略記号 (...) を指定して、型指定された配列として型の注釈を指定する必要があります。パラメータ配列は、JScript の **Array** オブジェクトになることはありません。

このエラーを解決するには

- 可変個引数リストを、型指定された配列として型の注釈を付けます。

参照

関連項目

function ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1222: Expando メソッドを抽象メソッドにすることはできません。

メソッドに **expando** 修飾子と **abstract** 修飾子の両方が指定されています。expando メソッドは抽象メソッドにできません。

このエラーを解決するには

- メソッドの宣言から **expando** 修飾子と **abstract** 修飾子のいずれかを削除します。

参照

概念

JScript スクリプトのトラブルシューティング

JScript の修飾子

その他の技術情報

JScript リファレンス

JS1223: ボディのない関数は抽象でなければなりません。

クラス内部の関数 (メソッドまたはプロパティ) に本体がなく、**abstract** 修飾子が指定されていません。**abstract** 修飾子が指定されている関数は、本体を持つことはできません。また、本体のある関数には **abstract** 修飾子を指定できません。

このエラーを解決するには

1. 関数に **abstract** 修飾子を指定します。
2. 関数に本体を追加します。

参照

概念

JScript スクリプトのトラブルシューティング

JScript の修飾子

その他の技術情報

JScript リファレンス

JS1224: この修飾子をインターフェイス メンバで使うことはできません。

インターフェイス メンバに、使用できない修飾子が指定されています。インターフェイス メンバには **public** 修飾子しか指定できません。

このエラーを解決するには

- インターフェイス メンバに指定されている修飾子が、**public** 修飾子だけであることを確認します。

参照

関連項目

interface ステートメント

概念

JScript スクリプトのトラブルシューティング

JScript の修飾子

その他の技術情報

JScript リファレンス

JS1226: 変数をインターフェイス内で宣言することはできません。

変数がインターフェイス内で宣言されていますが、これは許可されていません。変数は、フィールドを作成するためにクラス内で宣言します。

このエラーを解決するには

- インターフェイス宣言内に、変数宣言が指定されていないことを確認します。

参照

関連項目

interface ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1227: インターフェイスをインターフェイス内で宣言することはできません。

インターフェイスの宣言が、インターフェイス宣言内で入れ子になっていますが、これは許可されていません。インターフェイスの宣言は、グローバルスコープまたはパッケージ内でだけ指定できます。

このエラーを解決するには

- インターフェイス宣言が入れ子になっていないことを確認します。

参照

関連項目

interface ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1228: 列挙メンバ宣言で 'var' キーワードを使うことはできません。

列挙型の宣言に **var** キーワードが含まれていますが、列挙型の宣言内では変数を宣言できません。

このエラーを解決するには

- **var** キーワードまたは変数宣言を削除します。

参照

関連項目

enum ステートメント
var ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1229: import ステートメントはこのコンテキストでは有効ではありません。

コード中の **import** ステートメントが、グローバル スコープ以外の場所にあります。import ステートメントは、グローバル スコープでしか指定できません。

このエラーを解決するには

- **import** ステートメントを現在の場所からコードのメイン プログラム ブロック (グローバル スコープ) に移動します。

参照

関連項目

import ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1230: 列挙宣言はこのコンテキストでは許可されません。

列挙型の宣言が、不正な位置にあります。列挙型は、メイン プログラム ブロック、クラスの内部、パッケージ内部、または関数内でしか宣言できません。

このエラーを解決するには

- 列挙型をメイン プログラム ブロック、クラスの内部、パッケージの内部、または関数内で定義します。

参照

関連項目

enum ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1231: 属性がこの型の宣言で有効ではありません。

属性を指定できない宣言に、属性が適用されています。

このエラーを解決するには

- この種類の宣言に属性が適用されていないことを確認します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1232: パッケージ宣言はこのコンテキストでは許可されていません。

パッケージの宣言が、グローバル スコープ以外の場所にあります。パッケージはグローバル スコープでしか宣言できません。

このエラーを解決するには

- すべてのパッケージをグローバル スコープで宣言します。

参照

関連項目

package ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1233: コンストラクタ関数に戻り値を指定することはできません。

クラスのコンストラクタ関数に、戻り値の型が指定されています。コンストラクタ関数は、自動的に生成されたクラス インスタンスへの参照を返し、値を返しません。

このエラーを解決するには

1. 戻り値の型の指定をコンストラクタから削除します。
2. クラス名とは別の名前に変更して、コンストラクタをメソッドに変更します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

クラス ベースのオブジェクト

JS1234: 型およびパッケージ定義のみがライブラリ内で使用できます。

ライブラリを作成するためにコードがコンパイルされていますが、不正な宣言が含まれています。ライブラリを作成するためのコードには、クラス、インターフェイス、およびパッケージだけを含めることができます。

このエラーを解決するには

- コードに、クラス、インターフェイス、およびパッケージだけが含まれていることを確認します。

参照

関連項目

/target:library
class ステートメント
interface ステートメント
package ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1235: 無効なデバッグ ディレクティブです。

@debug ディレクティブに無効なオプションが使用されています。有効なオプションは `on` と `off` です。

このエラーを解決するには

- **@debug** ディレクティブのオプションに、`on` と `off` だけを使用します。

参照

関連項目

@debug ディレクティブ

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1236: この型の属性は一意でなければなりません。

識別子に属性が 2 回以上適用されていますが、その属性を適用できるのは 1 回だけです。

このエラーを解決するには

- 各識別子にその属性が 1 回だけ適用されていることを確認します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1237: 入れ子にされた非スタティック型は、同じクラス内で入れにされている非スタティック型によってのみ拡張できます。

クラス内に、入れ子になっているクラス定義がありますが、入れ子にされたクラスに **static** 修飾子が指定されていません。定義された別のクラスが、入れ子になっているクラスを拡張していますが、拡張クラスに適切な修飾子がないか、拡張クラスが、入れ子になったクラスと同じクラス内で定義されていません。入れ子になった非静的なクラスを拡張できるのは、同じクラス内で入れ子になっている別の非静的クラスだけです。

このエラーを解決するには

1. 入れ子になった非静的クラスだけが、同じクラス内で入れ子になっている非静的クラスを拡張することを確認します。
2. 拡張される入れ子になったクラスに **static** 修飾子を指定します。これにより、入れ子になっていないクラス、および入れ子になった静的なクラスは、入れ子になったクラスを拡張できます。

参照

関連項目

static 修飾子

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

クラス ベースのオブジェクト

JS1238: プロパティを指定する属性は、property getter が存在する場合、そこで指定しなければなりません。

setter (**function set** 宣言) および getter (**function get** 宣言) と共に定義されたプロパティに、setter に適用された属性がありますが、これは許可されていません。getter がある場合、属性はすべて明示的に getter に適用する必要があります。コンパイラは、属性を暗黙的に setter 関数に適用します。

このエラーを解決するには

- 属性をプロパティの getter に適用します。

参照

関連項目

function set ステートメント
function get ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1239: try ステートメントの catch ブロックの内側にないときに、スローを引数にすることはできません。

引数のない **throw** ステートメントが、**catch** ブロック以外の場所で使用されています。引数を指定せずに **throw** ステートメントを使用できるのは、**catch** ブロック内部だけです。この場合、キャッチされたエラーを再スローします。

このエラーを解決するには

1. **throw** ステートメントに引数を渡します。
2. **throw** ステートメントを catch ブロックに移動します。

参照

関連項目

try...catch...finally ステートメント
throw ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1240: 変数引数リストは最後の引数でなければなりません。

関数の定義で、パラメータ配列 (可変個引数リスト) の後に、別のパラメータが指定されています。パラメータ配列は、最後のパラメータとして指定する必要があります。

このエラーを解決するには

- パラメータ配列が、関数定義の最後のパラメータとして指定されていることを確認します。

参照

関連項目

function ステートメント

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1241: 型が見つかりませんでした。アセンブリ参照がない可能性があります。

型を参照する完全限定名に、パッケージ名と思われる修飾子が使用されています。型がパッケージに見つからないか、パッケージが見つかりません。この状況は、パッケージの型を提供するアセンブリが参照されていない場合に起こります。

このエラーを解決するには

1. 指定されたパッケージに型が存在することを確認します。
2. **/autoref** オプションをオンにするか、**/reference** オプションを使用してアセンブリを明示的に参照します。

参照

関連項目

import ステートメント
/reference
/autoref

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1242: 変形した octal リテラルが decimal リテラルとして扱われました。

リテラルの数字の先頭が 0 で始まっていて、小数点がありません。これは 8 進数の (基数 8) リテラルを表しています。このリテラルに、8 進数では使用しない数字の 8 または 9 が含まれています。コンパイラは、この数字を 10 進数 (基数 10) と解釈します。

このエラーを解決するには

1. リテラルが 10 進数リテラルである場合は、先頭の 0 を削除します。
2. リテラルが 8 進数リテラルである場合は、0 ～ 7 の数字だけを使用します。

参照

概念

JScript スクリプトのトラブルシューティング

数値データ

その他の技術情報

JScript リファレンス

JS1243: 非スタティックのメンバは、スタティック スコープからアクセスできません。

静的なメソッドまたはプロパティが、クラスの非静的なメンバにアクセスしています。静的なクラス メンバはクラス自身に関連付けられ、特定のインスタンスのメンバに関する情報を持ちます。非静的なメンバは、特定のインスタンスに関連付けられます。したがって、静的なメソッドとプロパティは、非静的なメンバにアクセスできません。

クラスのインスタンスを引数としてメソッドに渡す場合、静的メソッドは、非静的なメンバに間接的にアクセスできます。静的メソッドは、**private** 修飾子を指定したメンバを含む、クラス インスタンスのすべてのメンバにアクセスできます。

このエラーを解決するには

1. アクセスされるメンバおよびアクセスするメンバが、静的または非静的のいずれかになるように修飾子を変更します。
2. 静的メソッドにクラスのインスタンスを渡します。

参照

関連項目

static 修飾子

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

クラス ベースのオブジェクト

JS1244: スタティック メンバはクラス名でアクセスしなければなりません。

クラス インスタンスを通じて、静的なクラス メンバにアクセスしようとしています。静的なクラス メンバはクラス自身に関連付けられ、クラス インスタンスからはアクセスできません。これらのメンバには、クラス名を修飾子として使用して、直接アクセスする必要があります。

このエラーを解決するには

- 静的なクラス メンバには、クラス名を使ってアクセスするようにします。

参照

関連項目

static 修飾子

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

クラス ベースのオブジェクト

JS1245: 非スタティック メンバにクラス名でアクセスすることはできません。

クラス名を使用して、非静的なクラス メンバにアクセスしようとしています。クラス自身に関連付けることができるのは静的なクラス メンバだけです。非静的なメンバは、特定のクラス インスタンスに対して関連付けられ、そのインスタンスを通じてアクセスされます。

このエラーを解決するには

- 非静的なメンバが、クラス インスタンスを通じてアクセスされていることを確認します。

参照

関連項目

static 修飾子

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

クラス ベースのオブジェクト

JS1246: 型にこのようなスタティック メンバはありません。

クラス名を使用してメンバにアクセスしようとしていますが、対応する静的メンバが見つかりません。クラス名を使用する場合は、静的なメンバにしかアクセスできません。

このエラーを解決するには

- クラス名を使用してメンバにアクセスする場合、そのメンバが静的メンバであることを確認します。

参照

関連項目

static 修飾子

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

クラス ベースのオブジェクト

JS1247: ループ条件は関数参照です。

ループ条件は関数参照です。関数を呼び出そうとしましたか?

ループ ステートメントの条件式で、関数名の後に引数を囲むかっこが指定されていません。関数名自体は、関数の **Function** オブジェクトを参照し、関数が返す値を参照していません。

関数自身が変化する場合など、特定の状況では **Function** オブジェクトをループ条件に使用すると便利ですが、ほとんどの場合はエラーになります。

このエラーを解決するには

- 関数の引数をかっこで囲む形式の関数呼び出しの構文を使用して、関数の値を評価します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1248: 'assembly' を指定してください。

アセンブリのグローバル属性を定義しているようですが、アセンブリの識別子が使用されていません。アセンブリ属性を定義する正しい構文は次のとおりです。

```
[assembly: attribute]
```

`attribute` は、アセンブリの有効なグローバル属性である必要があり、**System.Reflection** 名前空間で提供されます。詳細については、「System.Reflection Namespace」を参照してください。

このエラーを解決するには

- 正しい構文を使用してグローバル属性を宣言します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1249: アセンブリ カスタム属性を別のコンストラクトの一部にすることはできません。

アセンブリのカスタム属性は、グローバル スコープでしか使用できません。

このエラーを解決するには

- アセンブリのカスタム属性がグローバル スコープで使用されていることを確認します。

参照

概念

JScript スクリプトのトラブルシューティング

変数と定数のスコープ

その他の技術情報

JScript リファレンス

JS1250: Expando メソッドをスタティックにすることはできません。

メソッドに **expando** 修飾子と **static** 修飾子の両方が指定されています。これは認められていません。

このエラーを解決するには

- メソッドの宣言から **expando** 修飾子と **static** 修飾子のいずれかを削除します。

参照

概念

JScript スクリプトのトラブルシューティング

JScript の修飾子

その他の技術情報

JScript リファレンス

JS1251: このメソッドには、このクラスの別のメソッドと同じ名前およびパラメータ型が指定されています。

クラスに同じ名前とパラメータ型を持つ複数のメソッドがあります。これらのメソッドを呼び出すときに区別できないため、これは許可されていません。

このエラーを解決するには

1. メソッドが重複している場合は、余分なメソッドを削除します。
2. 1 つ以上のメソッドの名前またはパラメータ型を変更します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

クラス ベースのオブジェクト

JS1252: コンストラクタとして使われるクラス メンバを expando 関数に設定することはできません。

new 演算子がクラス メンバに適用されています。**new** 演算子は、クラス メンバが **expando** 修飾子を指定したメソッドまたはプロパティである場合にだけ適用できます。これにより、クラス メンバをコンストラクタとして使用できます。

このエラーを解決するには

1. クラス メンバの定義に **expando** 修飾子を適用します。
2. expando クラスのメンバ以外には **new** 演算子を使用しないようにします。

参照

関連項目

expando 修飾子

new 演算子

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1253: 有効なバージョン文字列ではありません。

有効なバージョン文字列ではありません。major.minor[.build[.revision]] の形式を指定してください。

アセンブリのグローバル **AssemblyVersion** 属性を定義していますが、属性に渡されたバージョン文字列の形式が正しくありません。バージョン文字列は、"major.minor[.build[.revision]]" の形式で指定します。

このエラーを解決するには

- バージョン文字列を "major.minor[.build[.revision]]" の形式で指定します。

参照

関連項目

AssemblyVersionAttribute Class

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1254: 実行可能ファイルをローカライズできません。カルチャは常に空でなければなりません。

アセンブリのグローバル **AssemblyCulture** 属性を定義していますが、実行可能ファイルをローカライズできないため、この定義は不正です。

このエラーを解決するには

- 実行可能ファイルに **AssemblyCulture** 属性を指定しないようにします。

参照

関連項目

AssemblyCultureAttribute Class

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1255: 文字列の連結にプラス演算子を使うと速度が低下します。

文字列の連結にプラス演算子を使うと速度が低下します。System.Text.StringBuilder の使用を推奨します。

加算演算子 (+) は文字列を連結します。多数の短い文字列を別の文字列に追加する場合など、多くの状況では、**System.Text.StringBuilder** の方がより高速なコードを生成できます。

たとえば、文字列 "0123456789" を生成する次のコードを考えます。このコードは、コンパイル時にこの警告を生成します。

```
var a : String = "";
for(var i=0; i<10; i++)
   a += i;
print(a);
```

実行すると、文字列 "0123456789" が表示されます。

上の例に **System.Text.StringBuilder** を使用すると、プログラムはより高速になり、警告も生成されません。

```
var b : System.Text.StringBuilder;
b = new System.Text.StringBuilder();
for(var i=0; i<10; i++)
   b.Append(i);
print(b);
```

前のプログラムと同様に、"0123456789" が表示されます。

警告が表示されないようにする別の方法には、ほかの文字列が追加される文字列を、型指定されていない変数に格納する方法もあります。

このエラーを解決するには

1. ほかの文字列を追加する文字列の型に **System.Text.StringBuilder** を使用し、**+=** 演算子が使用されているステートメントを **Append** メソッドを使用して書き直します。
2. ほかの文字列を追加する文字列に、型指定されていない変数を使用します。この方法ではコードの実行は高速になりませんが、警告は表示されなくなります。

参照

関連項目

StringBuilder Class

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1256: 条件付コンパイルディレクティブおよび変数をデバッガで使うことはできません。

JScript プログラムのデバッグ中に、コマンド ウィンドウに条件付きコンパイル ディレクティブまたは変数が入力されました。条件付きコンパイル ディレクティブおよび変数は、プログラムのコンパイル時にだけ使用され、コンパイルの完了後には使用できません。

このエラーを解決するには

- コマンド ウィンドウに、条件付きコンパイル ディレクティブおよび変数を入力しないようにします。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JScript コードの作成、コンパイル、およびデバッグ

JS1257: Expando メソッドはパブリックでなければなりません。

expando 修飾子を指定したメソッドに、**public** 以外の可視性修飾子が指定されています。これは認められていません。

このエラーを解決するには

- メソッドの宣言から **expando** 修飾子と可視性修飾子のいずれかを削除します。

参照

概念

JScript スクリプトのトラブルシューティング

JScript の修飾子

その他の技術情報

JScript リファレンス

JS1258: Delegate を明示的に構築することはできません。メソッド名を使ってください。

関数からデリゲートを生成していますが、これは不要です。関数名自体がデリゲートを参照します。

このエラーを解決するには

- デリゲートを参照するには、かっこを除いた関数名を使用します。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JScript の関数

JS1259: 参照されたアセンブリは、参照されていない別のアセンブリに依存しているか、または見つかりませんでした。

ほかのアセンブリに依存するアセンブリを、**import** ステートメントを使用して暗黙的に、または **/reference** オプションを使用して明示的にインポートしています。参照されていないか、指定した場所に存在しないため、ほかのアセンブリが見つかりません。

このエラーの原因として、依存しているアセンブリを移動せずに、インポートしているアセンブリだけを別の場所に移動している可能性があります。別の原因としては、ほかのアセンブリが依存しているアセンブリを参照していない可能性があります。

このエラーを解決するには

1. プログラムで必要とされているアセンブリが存在していることを確認します。
2. 必要な各アセンブリに対して、場所と名前が正しく指定されていることを確認します。
3. ほかのアセンブリで必要で、プログラムでインポートされていないアセンブリを明示的に参照します。

参照

関連項目

import ステートメント

/reference

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1260: この変換は、実行時に失敗する可能性があります。

実行時に失敗する可能性のある、暗黙的な型変換があります。変換元のデータ値と変換先のデータ型に、明確な類似性がないことを表しています。

損失を伴う明示的な型変換を使用することで、コードの信頼性が向上し、実行時に失敗する可能性が低くなります。

このエラーを解決するには

1. 指定したデータと変換先のデータ型に互換性があることを確認します。
2. データを別の型に変換する場合は、明示的な型変換を使用します。

参照

概念

JScript スクリプトのトラブルシューティング

型変換

その他の技術情報

JScript リファレンス

JS1261: 文字列を数字または boolean 型に変換すると、実行時の速度が低下する可能性があります。

実行時に失敗する可能性のある、暗黙的な型変換があります。文字列値のいくつかに、数字または Boolean 値との明確な類似性がありません。

損失を伴う明示的な型変換を使用することで、コードの信頼性が向上し、実行時に失敗する可能性が低くなります。

このエラーを解決するには

1. 指定した文字列と変換先のデータ型に互換性があることを確認します。
2. データを別の型に変換する場合は、明示的な型変換を使用します。

参照

概念

JScript スクリプトのトラブルシューティング

型変換

その他の技術情報

JScript リファレンス

JS1262: 有効な .resources ファイルではありません。

プログラムは **/resource** オプションを指定してコンパイルされていますが、指定されたリソース ファイルの形式が不正です。ファイルが破損しているか、リソース ファイルでない可能性があります。

このエラーを解決するには

- **/resource** オプションで指定したファイルが、有効なリソース ファイルであることを確認します。

参照

関連項目

/resource

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1263: & 演算子は、引数の一覧内でのみ使用できます。

アンパサンド (**&**) が関数呼び出し以外の場所で使用されています。アンパサンドは、参照によるパラメータを受け取る関数に、変数を参照で渡すためだけに使用します。アンパサンドを使用できるのは、(アドレスを持つ) 変数名の前だけです。

参照による変数の引き渡しにより、関数で変数の値を変更できます。

メモ：
JScript では、参照パラメータを使った関数を定義できません。JScript では、アンパサンドは、参照パラメータを受け取る外部オブジェクトを呼び出すために用意されています。

このエラーを解決するには

- 関数の呼び出しにおいてアンパサンド (**&**) を変数名の前に指定し、関数が参照によるパラメータを受け取るようにします。

参照

概念

JScript スクリプトのトラブルシューティング

その他の技術情報

JScript リファレンス

JS1264: 指定された型は、CLS に準拠していません。

プログラムに **CLSCompliantAttribute** 属性が含まれていますが、コンパイラが共通言語仕様 (CLS: Common Language Specification) 準拠でないデータ型の定義を検出しました。このエラーの原因としては、次のようなことが考えられます。

- 型名が CLS 準拠でありません。CLS 準拠の名前では、名前をアンダースコア (_) で開始したり、名前にドル記号 ($) を含めることはできません。また、ほかのパブリック メンバと、大文字小文字の使用だけが異なる名前も使用できません。
- 列挙型が定義されていますが、基になるデータ型が CLS 準拠の型ではありません。たとえば、列挙型が、CLS 準拠でないプリミティブ型の **uint** を基にしている可能性があります。対応する CLS 準拠のシステム型は **System.UInt32** です。

このエラーを解決するには

1. データ型の名前をアンダースコアで開始したり、名前にドル記号 ($) を含まないようにします。また、ほかのメンバ名との違いが、大文字小文字の違いだけにならないようにします。
2. 列挙型の基になる型に、CLS 準拠のデータ型だけを使用します。

参照

関連項目

CLSCompliantAttribute Class

概念

JScript スクリプトのトラブルシューティング
CLS 準拠コードの記述
共通言語仕様

その他の技術情報

JScript リファレンス

JS1265: このクラスは、CLS 準拠として設定されていないため、クラス メンバを CLS 準拠として設定することはできません。

クラスに **CLSCompliantAttribute** 属性が指定されているメンバがありますが、クラス自身に **CLSCompliantAttribute** 属性が指定されていません。クラス メンバが CLS 準拠として指定される場合は、クラスも CLS 準拠として指定する必要があります。

このエラーを解決するには

- メンバに **CLSCompliantAttribute** 属性が指定されている各クラスに、**CLSCompliantAttribute** 属性を適用します。

参照

関連項目

CLSCompliantAttribute Class

概念

JScript スクリプトのトラブルシューティング

CLS 準拠コードの記述

共通言語仕様

その他の技術情報

JScript リファレンス

JS1266: アセンブリが CLS 準拠として設定されてないため、型を CLS 準拠として設定することはできません。

データ型に **CLSCompliantAttribute** 属性が指定されていますが、データ型を含むアセンブリに **CLSCompliantAttribute** 属性が指定されていません。アセンブリに CLS 準拠として指定されたデータ型がある場合は、アセンブリも CLS 準拠として指定する必要があります。

このエラーを解決するには

- データ型に **CLSCompliantAttribute** 属性が指定されている場合は、アセンブリにも **CLSCompliantAttribute** 属性を適用します。

参照

関連項目

CLSCompliantAttribute Class

概念

JScript スクリプトのトラブルシューティング

CLS 準拠コードの記述

共通言語仕様

その他の技術情報

JScript リファレンス

JS1267: 無効なプロセッサです。

参照アセンブリは、別のプロセッサをターゲットにしています。

このエラーを解決するには

- **/platform:anycpu** または作成するアセンブリと同じ **/platform** 値を使用して参照アセンブリが作成されていることを確認します。

参照

関連項目

CLSCompliantAttribute Class

概念

JScript スクリプトのトラブルシューティング

CLS 準拠コードの記述

共通言語仕様

その他の技術情報

JScript リファレンス

JS1268: 無効なアセンブリ キー ファイルです。

アセンブリ キー ファイルが見つかりません。または、アセンブリ キー ファイルに無効なデータが含まれています。

このエラーを解決するには

- アセンブリ属性 **System.Reflection.AssemblyKeyFile** に有効なキー データを含むファイルが設定されていることを確認します。

参照

関連項目

CLSCompliantAttribute Class

概念

JScript スクリプトのトラブルシューティング

CLS 準拠コードの記述

共通言語仕様

その他の技術情報

JScript リファレンス

JS1269: 型名が無効です。

完全修飾型名は 1024 文字未満である必要があります。

このエラーを解決するには

- 1024 文字未満の完全修飾型名を使用します。

参照

関連項目

CLSCompliantAttribute Class

概念

JScript スクリプトのトラブルシューティング

CLS 準拠コードの記述

共通言語仕様

その他の技術情報

JScript リファレンス

JS2013: 無効なターゲット

ターゲットの種類が無効です。

このエラーを解決するには

- **/target** コマンド ライン オプションに 'exe'、'library'、または 'winexe' を指定してください。

参照

関連項目

CLSCompliantAttribute Class

概念

JScript スクリプトのトラブルシューティング

CLS 準拠コードの記述

共通言語仕様

その他の技術情報

JScript リファレンス

JS2039: 無効なプラットフォームです。

プラットフォームの種類が無効です。

このエラーを解決するには

- **/platform** コマンド ライン オプションに 'x86'、'Itanium'、'x64'、または 'anycpu' を指定します。

参照

関連項目

CLSCompliantAttribute Class

概念

JScript スクリプトのトラブルシューティング

CLS 準拠コードの記述

共通言語仕様

その他の技術情報

JScript リファレンス

関数 (JScript)

次に示す関数は JScript の戻り値に組み込まれ、その他の関数で後の演算に使用できます。

このセクションの内容

GetObject 関数

ファイルのオートメーション オブジェクトへの参照を返します。

ScriptEngine 関数

使用中のスクリプト言語を表す文字列を返します。

ScriptEngineBuildVersion 関数

使用中のスクリプト エンジンのビルド バージョン番号を返します。

ScriptEngineMajorVersion 関数

使用中のスクリプト エンジンのメジャー バージョン番号を返します。

ScriptEngineMinorVersion 関数

使用中のスクリプト エンジンのマイナ バージョン番号を返します。

関連するセクション

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

.NET Framework リファレンス

.NET Framework クラス ライブラリおよびその他の基本要素の構文と構造を説明するトピックへのリンクを示します。

GetObject 関数 (JScript 8.0)

ファイルのオートメーション オブジェクトへの参照を返します。関数には、次の 2 つの形式があります。

```
function GetObject(class : String)
function GetObject(pathname : String [, class : String])
```

引数
class

必ず指定します。"*appName.objectType*" 形式の文字列を指定します。*appName* は、オブジェクトを提供するアプリケーションの名前、*objectType* は作成するオブジェクトの型またはクラスです。

pathname

必ず指定します。オブジェクトが含まれているファイル名までの完全パスを指定します。省略した場合は、引数 *class* を指定する必要があります。

解説

ファイルのオートメーション オブジェクトにアクセスし、オブジェクトをオブジェクト変数に代入するには、**GetObject** 関数を使用します。**GetObject** 関数で返されるオブジェクトへの参照をオブジェクト変数に代入します。次に例を示します。

```
var CADObject;
CADObject = GetObject("C:\\CAD\\SCHEMA.CAD");
```

このコード例が実行されると、指定された引数 *pathname* に対応するアプリケーションが起動し、指定されたファイルのオブジェクトがアクティブになります。**GetObject** 関数では、引数 *pathname* に長さ 0 の文字列 ("") が指定されると、指定の種類の新規オブジェクトのインスタンスを返します。引数 *pathname* を省略すると、**GetObject** 関数は、指定した種類の現在アクティブなオブジェクトを返します。指定した種類のオブジェクトが存在しない場合はエラーが発生します。

アプリケーションによっては、ファイルの一部をアクティブにできる場合もあります。ファイル名の最後に感嘆符 (!) を付け、続けてアクティブにするファイルの部分を表す文字列を指定します。このような文字列を作成する方法については、オブジェクトを作成したアプリケーションのドキュメントを参照してください。

たとえば、描画のアプリケーションでは、1 つのファイルの複数のレイヤに描画を保存する場合があります。次のコードを使用すると、SCHEMA.CAD という描画ファイルの 1 つのレイヤをアクティブにできます。

```
var LayerObject = GetObject("C:\\CAD\\SCHEMA.CAD!Layer3");
```

オブジェクトのクラスが指定されていない場合は、指定されたファイル名を基に、オートメーションは起動するアプリケーションとアクティブにするオブジェクトを決定します。ただし、ファイルによっては、複数のオブジェクトのクラスをサポートしている場合もあります。たとえば、1 つの描画で **Application** オブジェクト、**Drawing** オブジェクト、および **Toolbar** オブジェクトという 3 つの異なる種類のオブジェクトを同一ファイルの構成要素としてサポートすることもあります。ファイルのどのオブジェクトをアクティブにするかは、引数 *class* で指定します。次に例を示します。

```
var MyObject;
MyObject = GetObject("C:\\DRAWINGS\\SAMPLE.DRW", "FIGMENT.DRAWING");
```

このコード例では、FIGMENT が描画のアプリケーションの名前で、DRAWING がそのアプリケーションによってサポートされるオブジェクトの種類です。アクティブになったオブジェクトをコードで参照するには、定義したオブジェクト変数を使います。上の例では、オブジェクト変数 MyObject を使って新規オブジェクトのプロパティとメソッドにアクセスします。次に例を示します。

```
MyObject.Line(9, 90);
MyObject.InsertText(9, 100, "Hello, world.");
MyObject.SaveAs("C:\\DRAWINGS\\SAMPLE.DRW");
```

メモ :

GetObject 関数は、オブジェクトの現在のインスタンスがある場合や、既に読み込んだファイルからオブジェクトを作成する場合に使います。現在のインスタンスがない場合、または読み込み済みのファイルからオブジェクトを起動しない場合は、**ActiveXObject** オブジェクトを使います。

複数のインスタンスを作成できないオブジェクトの場合は、何度 **ActiveXObject** 関数を実行しても、そのオブジェクトのインスタンスは 1 つしか作成されません。単一インスタンス オブジェクトの場合、引数 *pathname* に長さ 0 の文字列 ("") を指定して **GetObject** 関数を呼び出すと、常に同じインスタンスを返します。また、引数 *pathname* を省略すると、エラーになります。

必要条件

Version 5

参照

関連項目

ActiveXObject オブジェクト

ScriptEngine 関数 (JScript 8.0)

使用中のスクリプト言語を表す文字列を返します。

```
function ScriptEngine() : String
```

解説

ScriptEngine 関数の戻り値は次のとおりです。

文字列	説明
JScript	現在のスクリプト エンジンは、Microsoft JScript です。
VBA	現在のスクリプト エンジンは、Microsoft Visual Basic for Applications です。
VBScript	現在のスクリプト エンジンは、Microsoft Visual Basic Scripting Edition です。

使用例

ScriptEngine 関数の使用例を次に示します。

```
function GetScriptEngineInfo(){
   var s;
   s = ""; // Build string with necessary info.
   s += ScriptEngine() + " Version ";
   s += ScriptEngineMajorVersion() + ".";
   s += ScriptEngineMinorVersion() + ".";
   s += ScriptEngineBuildVersion();
   return(s);
}
```

必要条件

Version 2

参照

関連項目

ScriptEngineBuildVersion 関数 (JScript 8.0)
ScriptEngineMajorVersion 関数 (JScript 8.0)
ScriptEngineMinorVersion 関数 (JScript 8.0)

ScriptEngineBuildVersion 関数 (JScript 8.0)

使用中のスクリプト エンジンのビルド バージョン番号を返します。

```
function ScriptEngineBuildVersion() : int
```

解説

この関数の戻り値は、使用中のスクリプト言語のダイナミック リンク ライブラリ (DLL) 内に格納されているバージョン情報から直接取得されます。

使用例

ScriptEngineBuildVersion 関数の使用例を次に示します。

```
function GetScriptEngineInfo(){
   var s;
   s = ""; // Build string with necessary info.
   s += ScriptEngine() + " Version ";
   s += ScriptEngineMajorVersion() + ".";
   s += ScriptEngineMinorVersion() + ".";
   s += ScriptEngineBuildVersion();
   return(s);
}
```

必要条件

Version 2

参照

関連項目

ScriptEngine 関数 (JScript 8.0)
ScriptEngineMajorVersion 関数 (JScript 8.0)
ScriptEngineMinorVersion 関数 (JScript 8.0)

ScriptEngineMajorVersion 関数 (JScript 8.0)

使用中のスクリプト エンジンのメジャー バージョン番号を返します。

```
function ScriptEngineMajorVersion() : int
```

解説

この関数の戻り値は、使用中のスクリプト言語のダイナミック リンク ライブラリ (DLL) 内に格納されているバージョン情報から直接取得されます。

使用例

ScriptEngineMajorVersion 関数の使用例を次に示します。

```
function GetScriptEngineInfo(){
   var s;
   s = ""; // Build string with necessary info.
   s += ScriptEngine() + " Version ";
   s += ScriptEngineMajorVersion() + ".";
   s += ScriptEngineMinorVersion() + ".";
   s += ScriptEngineBuildVersion();
   return(s);
}
```

必要条件

Version 2

参照

関連項目

ScriptEngine 関数 (JScript 8.0)
ScriptEngineBuildVersion 関数 (JScript 8.0)
ScriptEngineMinorVersion 関数 (JScript 8.0)

ScriptEngineMinorVersion 関数 (JScript 8.0)

使用中のスクリプト エンジンのマイナ バージョン番号を返します。

```
function ScriptEngineMinorVersion() : int
```

解説

この関数の戻り値は、使用中のスクリプト言語のダイナミック リンク ライブラリ (DLL) 内に格納されているバージョン情報から直接取得されます。

使用例

ScriptEngineMinorVersion 関数の使用例を次に示します。

```
function GetScriptEngineInfo(){
   var s;
   s = ""; // Build string with necessary info.
   s += ScriptEngine() + " Version ";
   s += ScriptEngineMajorVersion() + ".";
   s += ScriptEngineMinorVersion() + ".";
   s += ScriptEngineBuildVersion();
   return(s);
}
```

必要条件

Version 2

参照

関連項目

ScriptEngine 関数 (JScript 8.0)
ScriptEngineBuildVersion 関数 (JScript 8.0)
ScriptEngineMajorVersion 関数 (JScript 8.0)

リテラル

リテラルは、JScript コードのコンテキスト内で特別な意味を持つ不変のプログラム要素です。

このセクションの内容

関連するセクション

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

.NET Framework リファレンス

.NET Framework クラス ライブラリおよびその他の基本要素の構文と構造を説明するトピックへのリンクを示します。

false リテラル

false を表すブール値です。

解説

ブール値は **true** または **false** のいずれかです。**false** の逆、つまり **false** でないことを表すのは **true** です。

必要条件

Version 1

参照

関連項目

boolean 型 (JScript)
Boolean オブジェクト
true リテラル

その他の技術情報

JScript のデータ型

null リテラル

"オブジェクトがない" ことを表すオブジェクトです。

解説

null 値を代入することによって、変数自体は削除せずに、変数の内容を消去できます。

メモ：
JScript では、C および C++ 同様、**null** と 0 が区別されます。また、JScript の **typeof** 演算子では、null 値は **null** 型ではなく **Object** 型として扱われます。この処理は混乱を招く可能性がありますが、下位互換性のために残されています。

必要条件

Version 1

参照

関連項目

Object オブジェクト

その他の技術情報

JScript のデータ型

true リテラル

true を表すブール値です。

解説

ブール値は **true** または **false** のいずれかです。**true** の逆、つまり **true** でないことを表すのは **false** です。

必要条件

Version 1

参照

関連項目

boolean 型 (JScript)
Boolean オブジェクト
false リテラル

その他の技術情報

JScript のデータ型

メソッド

メソッドは、オブジェクトのメンバとなっている関数です。JScript では、多くのメソッドをメソッド名に従ってアルファベット順に分類しています。

このセクションの内容

メソッド (A ～ E)

メソッド (F ～ I)

メソッド (J ～ R)

メソッド (S)

メソッド (T ～ Z)

関連するセクション

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

オブジェクト

JScript のオブジェクトの概念と、オブジェクトがプロパティとメソッドにどのように関連するかについて説明します。JScript がサポートするオブジェクトの詳細を説明するトピックへのリンクも示します。

.NET Framework リファレンス

.NET Framework クラス ライブラリおよびその他の基本要素の構文と構造を説明するトピックへのリンクを示します。

メソッド (A ～ E)

メソッドは、オブジェクトのメンバとなっている関数です。ここでは、名前が A ～ E で始まるメソッドを示します。

このセクションの内容

指定された 2 つの文字列を連結し、連結した文字列を格納した **String** オブジェクトを返します。

cos メソッド

指定された数値のコサインを返します。

decodeURI メソッド

エンコード前の URI (Uniform Resource Identifier) を返します。

decodeURIComponent メソッド

エンコード前の URI (Uniform Resource Identifier) コンポーネントを返します。

dimensions メソッド

VBArray の次元数を返します。

encodeURI メソッド

文字列を有効な URI にエンコードします。

encodeURIComponent メソッド

文字列を有効な URI にエンコードします。

escape メソッド

String オブジェクトに含まれる非 ASCII 文字などをエスケープコード付きの文字に変換します。

eval メソッド

JScript のコードを評価し、実行します。

exec メソッド

指定された文字列内で、パターンに一致する文字列の検索を実行します。

exp メソッド

e (自然対数の底) の累乗を返します。

関連するセクション

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

メソッド

JScript で使用できるメソッドのアルファベット順の一覧と、メソッドの各カテゴリへのリンクを示します。

オブジェクト

JScript のオブジェクトの概念と、オブジェクトがプロパティとメソッドにどのように関連するかについて説明します。JScript がサポートするオブジェクトの詳細を説明するトピックへのリンクも示します。

abs メソッド

指定された数値の絶対値を返します。

```
function abs( number : Number ) : Number
```

引数
number

必ず指定します。任意の数式を指定します。

解説

戻り値は、引数 *number* の絶対値です。

使用例

abs メソッドの使用例を次に示します。

```
function ComparePosNegVal(n) {
   var s = "";
   var v1 = Math.abs(n);
   var v2 = Math.abs(-n);
   if (v1 == v2) {
      s = "The absolute values of " + n + " and "
      s += -n + " are identical.";
   }
   return(s);
}
```

必要条件

Version 1

対象

Math オブジェクト

参照
その他の技術情報
メソッド

acos メソッド

指定された数値のアークコサインを返します。

```
function acos( number : Number ) : Number
```

引数
number

必ず指定します。任意の数式を指定します。

解説

戻り値は、引数 *number* のアークコサインを表す 0 ～πの範囲の値です。*number* が -1 より小さいか、+1 より大きい場合、戻り値は **NaN** になります。

必要条件

Version 1

対象：

Math オブジェクト

参照
関連項目
asin メソッド
atan メソッド
cos メソッド
sin メソッド
tan メソッド
PI プロパティ

anchor メソッド

オブジェクトの指定されたテキストを HTML の NAME 属性付きの <A> タグと </A> タグで囲み、文字列として返します。

```
function anchor(anchorString : String ) : String
```

引数
anchorString

必ず指定します。アンカー タグの NAME 属性に設定する文字列を指定します。

解説

anchor メソッドを呼び出すと、**String** オブジェクトから名前付きアンカーが作成されます。

このメソッドの実行時には、タグが既に元の文字列内にあるかどうかの確認は行われません。

使用例

anchor メソッドの使用例を次に示します。

```
var strVariable = "This is an anchor";
strVariable = strVariable.anchor("Anchor1");
```

2 つ目のステートメントの実行後、変数 `strVariable` は次の値になります。

```
<A NAME="Anchor1">This is an anchor</A>
```

必要条件

Version 1

対象：

String オブジェクト

参照
関連項目
link メソッド

apply メソッド

現在のオブジェクトの代わりに他のオブジェクトを使用して、オブジェクトのメソッドを返します。

```
function apply([thisObj : Object [,argArray : { Array | arguments }]])
```

引数

thisObj

省略可能です。現在のオブジェクトとして使用するオブジェクトを指定します。

argArray

省略可能です。関数に渡す引数の配列か **arguments** オブジェクトを指定します。

解説

argArray が有効な配列でない場合、または **arguments** オブジェクトでない場合は、**TypeError** になります。

argArray と *thisObj* のどちらも指定しない場合は、**global** オブジェクトが *thisObj* として使用され、引数は渡されません。

必要条件

Version 5.5

対象

Function オブジェクト

参照

その他の技術情報

メソッド

asin メソッド

指定された数値のアークサインを返します。

```
function asin(number : Number) : Number
```

引数
number

必ず指定します。任意の数式を指定します。

解説

戻り値は、引数 *number* のアークサインを表す -π/2 ～π/2 の範囲の値です。*number* が -1 より小さいか、+1 より大きい場合、戻り値は **NaN** になります。

必要条件

Version 1

対象

Math オブジェクト

参照
関連項目
acos メソッド
atan メソッド
cos メソッド
sin メソッド
tan メソッド
PI プロパティ

JScript 8.0

atan メソッド

指定された数値のアークタンジェントを返します。

```
function atan(number : Number ) Number
```

引数
number

必ず指定します。任意の数式を指定します。

解説

戻り値は、引数 *number* のアークタンジェントを表す -π/2 ～π/2 の範囲の値です。

必要条件

Version 1

対象：

Math オブジェクト

参照
関連項目
acos メソッド
asin メソッド
atan2 メソッド
cos メソッド
sin メソッド
tan メソッド
PI プロパティ

atan2 メソッド

X 軸から座標 (*x*,*y*) までのラジアン単位の角度を返します。

```
function atan2(y : Number , x : Number ) : Number
```

引数

x

必ず指定します。デカルト座標系での x 座標を表す数式を指定します。

y

必ず指定します。デカルト座標系での y 座標を表す数式を指定します。

解説

戻り値は、座標 (*x*,*y*) から求められた角度を表す -π～π の範囲の値です。

必要条件

Version 1

対象：

Math オブジェクト

参照

関連項目

atan メソッド
tan メソッド
PI プロパティ

atEnd メソッド

列挙子がコレクションの最後に置かれているかどうかを示すブール値を返します。

```
function atEnd() : Boolean
```

解説

atEnd メソッドは、現在の項目がコレクションの最後に置かれている場合、コレクションが空の場合、または現在の項目が未定義の場合は、真 (**true**) を返します。それ以外の場合は **false** を返します。

使用例

atEnd メソッドを使用し、ドライブの一覧が最後まで列挙されているかどうかを調べるコードの例を次に示します。

```
function ShowDriveList(){
   var fso, s, n, e, x;
   fso = new ActiveXObject("Scripting.FileSystemObject");
   e = new Enumerator(fso.Drives);
   s = "";
   for (; !e.atEnd(); e.moveNext())
   {
      x = e.item();
      s = s + x.DriveLetter;
      s += " - ";
      if (x.DriveType == 3)
         n = x.ShareName;
      else if (x.IsReady)
         n = x.VolumeName;
      else
         n = "[Drive not ready]";
      s +=   n + "<br>";
   }
   return(s);
}
```

必要条件

Version 2

対象：

Enumerator オブジェクト

参照

関連項目

item メソッド (JScript)
moveFirst メソッド
moveNext メソッド

big メソッド

String オブジェクトに格納されているテキストを HTML の <BIG> タグと </BIG> タグで囲み、文字列として返します。

```
function big() : String
```

解説

このメソッドの実行時には、タグが既に元の文字列内にあるかどうかの確認は行われません。

使用例

big メソッドの使用例を次に示します。

```
var strVariable = "This is a string object";
strVariable = strVariable.big();
```

2 つ目のステートメントの実行後、変数 `strVariable` は次の値になります。

```
<BIG>This is a string object</BIG>
```

必要条件

Version 1

対象：

String オブジェクト

参照

関連項目

small メソッド

blink メソッド

String オブジェクトに格納されているテキストを HTML の <BLINK> タグと </BLINK> タグで囲み、文字列として返します。

```
function blink() : String
```

解説

このメソッドの実行時には、タグが既に元の文字列内にあるかどうかの確認は行われません。

<BLINK> タグは、Microsoft Internet Explorer ではサポートされていません。

使用例

blink メソッドの使用例を次に示します。

```
var strVariable = "This is a string object";
strVariable = strVariable.blink();
```

2 つ目のステートメントの実行後、変数 `strVariable` は次の値になります。

```
<BLINK>This is a string object</BLINK>
```

必要条件

Version 1

対象：

String オブジェクト

参照

その他の技術情報

メソッド

bold メソッド

String オブジェクトに格納されているテキストを HTML の <B> タグと </B> タグで囲み、文字列として返します。

```
function bold() : String
```

解説

このメソッドの実行時には、タグが既に元の文字列内にあるかどうかの確認は行われません。

使用例

bold メソッドの使用例を次に示します。

```
var strVariable = "This is a string object";
strVariable = strVariable.bold();
```

2 つ目のステートメントの実行後、変数 `strVariable` は次の値になります。

```
<B>This is a string object</B>
```

必要条件

Version 1

対象：

String オブジェクト

参照

関連項目

italics メソッド

call メソッド

現在のオブジェクトの代わりに、他のオブジェクトを使用してメソッドを呼び出します。

```
function call([thisObj : Object [, arg1[, arg2[, ... [, argN]]]]])
```

引数
thisObj

省略可能です。現在のオブジェクトとして使用するオブジェクトを指定します。

arg1, arg2, ..., argN

省略可能です。メソッドに渡す引数のリストを指定します。

解説

別のオブジェクトに代わってメソッドを呼び出すときに **call** メソッドを使用します。**call** メソッドを使用すると、関数のオブジェクト コンテキストを元のコンテキストから *thisObj* で指定した新しいオブジェクトに変更できます。

thisObj を指定しない場合は、**global** オブジェクトが *thisObj* として使用されます。

必要条件

Version 5.5

対象

Function オブジェクト

参照
その他の技術情報
メソッド

ceil メソッド

指定された数値以上の最小の整数を返します。

```
function ceil(number : Number ) : Number
```

引数
number

必ず指定します。任意の数式を指定します。

解説

戻り値は、引数 *number* と等しいかそれより大きい値となる最小の整数です。

必要条件

Version 1

対象：

Math オブジェクト

参照
関連項目
floor メソッド

charAt メソッド

String オブジェクトの指定されたインデックス番号の位置にある文字を返します。

```
function charAt(index : Number) : String
```

引数
index

必ず指定します。文字の位置を 0 から始まるインデックスで指定します。指定できる値の範囲は、0 から文字列の長さより 1 小さい値までの範囲です。

解説

charAt メソッドの戻り値は、指定された位置にある文字です。文字列内の 1 文字目のインデックス番号は 0、2 文字目のインデックス番号は 1 となります。有効範囲外の値を指定した場合は空の文字列を返します。

使用例

charAt メソッドの使用例を次に示します。

```
function charAtTest(n){
   var str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"; // Initialize variable.
   var s;                                  // Declare variable.
   s = str.charAt(n - 1);                  // Get correct character
                                           // from position n – 1.
   return(s);                              // Return character.
}
```

必要条件

Version 1

対象

String オブジェクト

参照
その他の技術情報
メソッド

charCodeAt メソッド

String オブジェクトの指定した位置にある文字の Unicode コードを整数値で返します。

```
function charCodeAt(index : Number) : String
```

引数
index

必ず指定します。文字の位置を 0 から始まるインデックスで指定します。指定できる値の範囲は、0 から文字列の長さより 1 小さい値までの範囲です。

解説

文字列内の 1 文字目のインデックス番号は 0、2 文字目のインデックス番号は 1 となります。

index で指定した位置に文字がない場合は、**NaN** を返します。

使用例

charCodeAt メソッドの使用例を次に示します。

```
function charCodeAtTest(n){
  var str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"; //Initialize variable.
  var n;                                  //Declare variable.
  n = str.charCodeAt(n - 1);              //Get the Unicode value of the
                                          // character at position n.
  return(n);                              //Return the value.
}
```

必要条件

Version 5.5

対象

String オブジェクト

参照
関連項目
fromCharCode メソッド

compile メソッド (JScript)

実行を高速化するために、正規表現を内部形式にコンパイルします。

```
function compile(pattern : String [, flags : String] )
```

引数
pattern

必ず指定します。コンパイルする正規表現パターンを格納した文字列式を指定します。

flags

省略可能です。指定できるフラグは、次のとおりです。

- g (引数 *pattern* に指定したパターンと一致する文字列をすべて検索するグローバル検索)
- i (大文字小文字を区別しない)
- m (複数行検索)

解説

compile メソッドは、検索の実行を高速化するために、引数 *pattern* に指定したパターンを内部形式に変換します。これにより、たとえばループ内などで、正規表現をより効率よく使用できるようになります。正規表現をコンパイルすると、繰り返し同じ表現を使用する場合に処理が速くなります。ただし、正規表現を変更すると使用できなくなります。

使用例

compile メソッドの使用例を次に示します。

```
function CompileDemo(){
   var rs;
   var s = "AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPp"
   // Create regular expression for uppercase only.
   var r = new RegExp("[A-Z]", "g");
   var a1 = s.match(r)              // Find matches.
   // Compile the regular expression for lowercase only.
   r.compile("[a-z]", "g");
   var a2 = s.match(r)              // Find matches.
   return(a1 + "\n" + a2);
}
```

必要条件

Version 3

対象

Regular Expression オブジェクト

参照
概念
正規表現の構文

concat メソッド (Array)

現在の配列と追加項目を連結した新しい配列を返します。

```
function concat([item1 : { Object | Array } [, ... [, itemN : { Object | Array }]]]) : Arr
ay
```

引数
item1, item2, ..., itemN

省略可能です。現在の配列の後に追加する項目を指定します。

解説

concat メソッドは、現在の配列と他の項目とを連結した配列が格納されている **Array** オブジェクトを返します。

配列に追加する項目 (*item1 ... itemN*) は、左から順に追加されます。追加する項目が配列である場合は、その配列の内容が現在の配列の末尾に追加されます。項目が配列以外である場合は、配列の末尾に 1 つの配列要素として追加されます。

元の配列の要素は、次のように結果の配列にコピーされます。

- 連結前の配列から新しい配列にオブジェクト参照をコピーした場合、オブジェクト参照によって指定されるオブジェクトは変わりません。新しい配列と元の配列のいずれかが変更されると、その変更は他方にも反映されます。
- 新しい配列に数値型または文字列型の値が連結された場合は、値だけがコピーされます。一方の配列の値が変更された場合でも、その変更は他方の配列の値には反映されません。

使用例

concat メソッドを配列に対して使用した例を次に示します。

```
function ConcatArrayDemo(){
   var a, b, c, d;
   a = new Array(1,2,3);
   b = "JScript";
   c = new Array(42, "VBScript");
   d = a.concat(b, c);
   //Returns the array [1, 2, 3, "JScript", 42, "VBScript"]
   return(d);
}
```

必要条件

Version 3

対象

Array オブジェクト

参照
関連項目
concat メソッド (String)
join メソッド
String オブジェクト

concat メソッド (String)

現在の文字列と指定した文字列を連結して、連結した文字列の値を返します。

```
function concat([string1 : String [, ... [, stringN : String]]]]) : String
```

引数
string1, ..., stringN

省略可能です。現在の文字列の末尾に連結する **String** オブジェクトの名前またはリテラルを指定します。

解説

concat メソッドの処理内容は、*result = curstring + string1 + ... + stringN* と記述した場合と同じです。*curstring* は、**concat** メソッドを提供するオブジェクトに格納された文字列です。元の文字列と結果の文字列のどちらか一方が変更された場合でも、その変更はもう一方の文字列には反映されません。引数が文字列でない場合は、文字列に変換された後 *curstring* に連結されます。

使用例

concat メソッドを文字列に対して使用した例を次に示します。

```
function concatDemo(){
   var str1 = "ABCDEFGHIJKLM"
   var str2 = "NOPQRSTUVWXYZ";
   var s = str1.concat(str2);
   // Return concatenated string.
   return(s);
}
```

必要条件

Version 3

対象

String オブジェクト

参照
関連項目
加算演算子 (+)
Array オブジェクト
concat メソッド (Array)

cos メソッド

指定された数値のコサインを返します。

```
function cos(number : Number) : Number
```

引数
number

必ず指定します。任意の数式を指定します。

解説

戻り値は、引数 *number* のコサインの値です。

必要条件

Version 1

対象：

Math オブジェクト

参照
関連項目
acos メソッド
asin メソッド
atan メソッド
sin メソッド
tan メソッド

decodeURI メソッド

エンコード前の URI (Uniform Resource Identifier) を返します。

```
function decodeURI(URIstring : String) : String
```

引数
URIstring

必ず指定します。エンコードされた URI を表す文字列を指定します。

解説

旧版の **unescape** メソッドの代わりに **decodeURI** メソッドを使用します。

decodeURI メソッドは文字列値を返します。

URIString が有効でない場合は **URIError** が発生します。

必要条件

Version 5.5

対象：

Global オブジェクト

参照
関連項目
decodeURIComponent メソッド
encodeURI メソッド

decodeURIComponent メソッド

エンコード前の URI (Uniform Resource Identifier) コンポーネントを返します。

```
function decodeURIComponent(encodedURIString : String) : String
```

解説

引数 *encodedURIString* は必ず指定し、エンコードされた URI コンポーネントを表す値を指定します。

解説

URIComponent は、URI の構成要素です。

encodedURIString が有効でない場合は **URIError** が発生します。

必要条件

Version 5.5

対象：

Global オブジェクト

参照

関連項目

decodeURI メソッド
encodeURI メソッド

dimensions メソッド

VBArray の次元数を返します。

```
function dimensions() : Number
```

解説

dimensions メソッドは、指定した VBArray が持つ次元数を取得するためのメソッドです。

次のコードは、3 つの部分から構成されます。最初の部分は、Visual Basic のセーフ配列を作成する VBScript のコードです。2 番目の部分は、このセーフ配列の次元数と各次元の上限値を取得する JScript のコードです。どちらのコードも、HTML ページの <HEAD> セクションに記述します。3 番目の部分は、他の 2 つのコードを実行するために <BODY> セクションに記述する JScript のコードです。

```
<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Function CreateVBArray()
   Dim i, j, k
   Dim a(2, 2)
   k = 1
   For i = 0 To 2
      For j = 0 To 2
         a(j, i) = k
         k = k + 1
      Next
   Next
   CreateVBArray = a
End Function
-->
</SCRIPT>

<SCRIPT LANGUAGE="JScript">
<!--
function VBArrayTest(vba)
{
   var i, s;
   var a = new VBArray(vba);
   for (i = 1; i <= a.dimensions(); i++)
   {
      s = "The upper bound of dimension ";
      s += i + " is ";
      s += a.ubound(i)+ ".<BR>";
   }
   return(s);
}
-->
</SCRIPT>
</HEAD>

<BODY>
<SCRIPT language="jscript">
   document.write(VBArrayTest(CreateVBArray()));
</SCRIPT>
</BODY>
```

必要条件

Version 3

対象：

VBArray オブジェクト

参照

関連項目

getItem メソッド
lbound メソッド
toArray メソッド
ubound メソッド

encodeURI メソッド

有効な URI (Uniform Resource Identifier) にエンコードされた文字列を返します。

```
function encodeURI(URIString : String) : String
```

引数
URIString

必ず指定します。エンコードされた URI を表す文字列を指定します。

解説

encodeURI メソッドはエンコードした URI を返します。結果を **decodeURI** に渡すと、元の文字列が返されます。**encodeURI** メソッドは、「:」、「/」、「;」、「?」の各文字はエンコードしません。これらの文字をエンコードする場合、**encodeURIComponent** を使用します。

必要条件

Version 5.5

対象

Global オブジェクト

参照
関連項目
decodeURI メソッド
decodeURIComponent メソッド

encodeURIComponent メソッド

有効な URI (Uniform Resource Identifier) のコンポーネントにエンコードされた文字列を返します。

```
function encodeURIComponent(encodedURIString : String) : String
```

引数
encodedURIString

必ず指定します。エンコードされた URI コンポーネントを表す文字列を指定します。

解説

encodeURIComponent メソッドはエンコードした URI を返します。結果を **decodeURIComponent** に渡すと、元の文字列が返されます。**encodeURIComponent** メソッドでは、すべての文字がエンコードされるため、*/folder1/folder2/default.html* などのパスを表す文字列には注意が必要です。スラッシュ (/) もエンコードされるため、Web サーバーへの要求として送信する場合は無効になります。文字列に URI コンポーネントが複数含まれる場合は、**encodeURI** メソッドを使用します。

必要条件

Version 5.5

対象

Global オブジェクト

参照
関連項目
decodeURI メソッド
decodeURIComponent メソッド

escape メソッド

すべてのコンピュータで正しく読み取ることができる、エンコードされた **String** オブジェクトを返します。

```
function escape(charString : String) : String
```

引数
charString

必ず指定します。エンコード対象の **String** オブジェクトまたはリテラルを指定します。

解説

escape メソッドは、*charstring* の内容を格納する文字列値 (Unicode 形式) を返します。All スペース、区切り記号、アクセント記号付き文字などの非 ASCII 文字は、**%***xx* の形式にエンコードします。*xx* は、エンコード対象の文字を表す 16 進数です。たとえば、スペースを指定すると、"%20" という文字列が返されます。

256 以上の値を持つ文字は **%u***xxxx* という形式で保存されます。

メモ :

escape メソッドは、URI (Uniform Resource Identifier) のエンコードには使用しないでください。代わりに、**encodeURI** メソッドまたは **encodeURIComponent** メソッドの使用をお勧めします。

必要条件

Version 1

対象

Global オブジェクト

参照
関連項目
encodeURI メソッド
encodeURIComponent メソッド
String オブジェクト
unescape メソッド

eval メソッド (JScript)

JScript のコードを評価し、実行します。

```
function eval(codeString : String [, override : String])
```

引数
codeString

必ず指定します。有効な JScript コードを含む文字列を指定します。

override

省略可能です。*codeString* のコードに適用するセキュリティ アクセス許可を決定する文字列を指定します。

解説

eval メソッドを使用すると、JScript ソース コードを動的に実行できます。

eval メソッドに渡されたコードは、**eval** メソッドを呼び出した場合と同じように実行されます。**eval** ステートメントで定義されている新しい変数またはデータ型は、外側のプログラムでは参照できません。

eval メソッドに渡されるコードは、文字列 "unsafe" が 2 番目のパラメータとして渡されていない限り、制限付きのセキュリティ コンテキストで実行されます。制限付きのセキュリティ コンテキストでは、ファイル システム、ネットワーク、ユーザー インターフェイスなどのシステム リソースへのアクセスが禁止されます。これらのリソースにアクセスしようとすると、セキュリティ例外が生成されます。

eval の 2 番目のパラメータが文字列 "unsafe" である場合、**eval** メソッドに渡されるコードは、呼び出し元のコードと同じセキュリティ コンテキストで実行されます。2 番目のパラメータの大文字と小文字は区別されるため、"Unsafe" または "UnSAfE" という文字列を指定しても、制限付きのセキュリティ コンテキストはオーバーライドされません。

セキュリティに関するメモ :
eval は、信頼できるソースから取得したコード文字列を実行する場合だけ、unsafe モードで使用してください。

使用例

たとえば、次のコードでは、`doTest` 変数の値に応じて、変数 `mydate` をテスト用の日付または現在の日付で初期化します。

```
var doTest : boolean = true;
var dateFn : String;
if(doTest)
   dateFn = "Date(1971,3,8)";
else
   dateFn = "Date()";

var mydate : Date;
eval("mydate = new "+dateFn+";");
print(mydate);
```

このプログラムの出力は次のようになります。

```
Thu Apr 8 00:00:00 PDT 1971
```

必要条件

Version 1

対象

Global オブジェクト

参照
関連項目
String オブジェクト

exec メソッド

正規表現パターンを使って文字列に対して検索を実行し、検索結果を配列に格納して返します。

```
function exec(str : String) : Array
```

引数
str

必ず指定します。検索対象とする **String** オブジェクトの名前またはリテラル文字列を指定します。

解説

パターンに一致する文字列が見つからなかった場合、**exec** メソッドは **null** を返します。一致する文字列が見つかった場合は、配列を返し、さらにグローバルな **RegExp** オブジェクトのプロパティが検索結果を反映して更新されます。配列の要素 0 には一致結果全体が、要素が 1 から *n* には、一致結果の中に副次的に含まれる一致の内容が格納されます。この処理は、グローバル フラグ (**g**) が設定されていない場合の **match** メソッドの処理と同じです。

正規表現でグローバル フラグが設定されている場合は、**lastIndex** の値で指定された位置から文字列の検索が開始されます。グローバル フラグが設定されていない場合、**lastIndex** の値に関係なく検索は文字列の先頭から開始されます。

exec メソッドが返す配列には、**input**、**index**、および **lastIndex** の 3 つのプロパティがあります。**input** プロパティには、検索された文字列全体が格納されます。**index** プロパティには、検索された文字列内の一致した部分文字列の位置が格納されます。**lastIndex** プロパティには、一致文字列の末尾の文字に続く位置が格納されます。

使用例

exec メソッドの使用例を次に示します。

```
function RegExpTest() {
   var s = "";
   var src = "The rain in Spain falls mainly in the plain.";
   // Create regular expression pattern for matching a word.
   var re = /\w+/g;
   var arr;
   // Loop over all the regular expression matches in the string.
   while ((arr = re.exec(src)) != null)
      s += arr.index + "-" + arr.lastIndex + "\t" + arr + "\n";
   return s;
}
```

必要条件

Version 3

対象

Regular Expression オブジェクト

参照
関連項目
match メソッド
RegExp オブジェクト
search メソッド
test メソッド
概念
正規表現の構文

exp メソッド

指定された数値を指数とする e (自然対数の底) の累乗を返します。

```
function exp(number : Number) : Number
```

引数
number

必ず指定します。任意の数式を指定します。

解説

戻り値は、enumber です。定数 *e* は自然対数の底 (約 2.178) です。*number* は引数として渡された値です。

必要条件

Version 1

対象

Math オブジェクト

参照
関連項目
E プロパティ

メソッド (F ～ I)

メソッドは、オブジェクトのメンバとなっている関数です。ここでは、名前が F ～ I で始まるメソッドを示します。

このセクションの内容

Date オブジェクトに格納されている曜日の値を世界協定時刻 (UTC) で返します。

getUTCFullYear メソッド

Date オブジェクトに格納されている年の値を世界協定時刻 (UTC) で返します。

getUTCHours メソッド

Date オブジェクトに格納されている時の値を世界協定時刻 (UTC) で返します。

getUTCMilliseconds メソッド

Date オブジェクトに格納されているミリ秒の値を世界協定時刻 (UTC) で返します。

getUTCMinutes メソッド

Date オブジェクトに格納されている分の値を世界協定時刻 (UTC) で返します。

getUTCMonth メソッド

Date オブジェクトに格納されている月の値を世界協定時刻 (UTC) で返します。

getUTCSeconds メソッド

Date オブジェクトに格納されている秒の値を世界協定時刻 (UTC) で返します。

getVarDate メソッド

Date オブジェクトの VT_DATE 値を返します。

getYear メソッド

Date オブジェクトに格納されている年の値を返します。このメソッドは互換性のために残されています。代わりに、**getFullYear** メソッドの使用をお勧めします。

hasOwnProperty メソッド

指定した名前のプロパティがオブジェクトにあるかどうかを表すブール値を返します。

indexOf メソッド

String オブジェクト (文字列) 内を先頭から、指定された文字列で検索します。

isFinite メソッド

指定した数値が有限かどうかを表すブール値を返します。

isNaN メソッド

指定した値が予約済みの値 **NaN** (非数) かどうかを表すブール値を返します。

isPrototypeOf メソッド

オブジェクトが、別のオブジェクトのプロトタイプ チェインに存在するかどうかを表すブール値を返します。

italics メソッド

String オブジェクトに格納されている文字列を HTML の <I> タグと </I> タグで囲みます。

item メソッド

コレクション内の現在の項目を返します。

関連するセクション

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

メソッド

JScript で使用できるメソッドのアルファベット順の一覧と、メソッドの各カテゴリへのリンクを示します。

オブジェクト

JScript のオブジェクトの概念と、オブジェクトがプロパティとメソッドにどのように関連するかについて説明します。JScript がサポートするオブジェクトの詳細を説明するトピックへのリンクも示します。

fixed メソッド

String オブジェクトに格納されているテキストを HTML の <TT> タグと </TT> タグで囲み、文字列として返します。

```
function fixed() : String
```

解説

このメソッドの実行時には、タグが既に元の文字列内にあるかどうかの確認は行われません。

使用例

fixed メソッドの使用例を次に示します。

```
var strVariable = "This is a string object";
strVariable = strVariable.fixed();
```

2 つ目のステートメントの実行後、変数 `strVariable` は次の値になります。

```
<TT>This is a string object</TT>
```

必要条件

Version 1

対象：

String オブジェクト

参照

その他の技術情報

メソッド

JScript 8.0

floor メソッド

指定された数値以下の最大の整数を返します。

```
function floor(number : Number) : Number
```

引数
number

必ず指定します。任意の数式を指定します。

解説

戻り値は、引数 *number* と等しいかそれより小さい値となる最大の整数です。

必要条件

Version 1

対象：

Math オブジェクト

参照
関連項目
ceil メソッド

fontcolor メソッド

String オブジェクトに格納されているテキストを HTML の COLOR 属性付きの <FONT> タグと </FONT> タグで囲み、文字列として返します。

```
function fontcolor(colorVal : String) : String
```

引数
colorVal

必ず指定します。文字色を指定する文字列値を指定します。文字色を表す 16 進数または既定の文字色名を指定できます。

解説

使用できる色の文字列定数は、使用する Microsoft JScript のホスト アプリケーションごとに (つまり、ブラウザ、サーバーなどの種類により) 異なります。また、ホスト アプリケーションのバージョンにより使用できる色名が異なる場合もあります。詳細については、ホスト アプリケーションのドキュメントを参照してください。

このメソッドの実行時には、タグが既に元の文字列内にあるかどうかの確認は行われません。

使用例

fontcolor メソッドの使用例を次に示します。

```
var strVariable = "This is a string";
strVariable = strVariable.fontcolor("red");
```

2 つ目のステートメントの実行後、変数 `strVariable` は次の値になります。

```
<FONT COLOR="RED">This is a string</FONT>
```

必要条件

Version 1

対象：

String オブジェクト

参照
関連項目
fontsize メソッド

fontsize メソッド

String オブジェクトに格納されているテキストを HTML の SIZE 属性付きの <FONT> タグと </FONT> タグで囲み、文字列として返します。

```
function fontsize(intSize : Number) : String
```

引数
intSize

必ず指定します。テキストのサイズを指定する整数を指定します。

解説

この引数に指定できる値は、使用する Microsoft JScript のホスト アプリケーションごとに異なります。詳細については、ホスト アプリケーションのドキュメントを参照してください。

このメソッドの実行時には、タグが既に元の文字列内にあるかどうかの確認は行われません。

使用例

fontsize メソッドの使用例を次に示します。

```
var strVariable = "This is a string";
strVariable = strVariable.fontsize(-1);
```

2 つ目のステートメントの実行後、変数 `strVariable` は次の値になります。

```
<FONT SIZE="-1">This is a string</FONT>
```

必要条件

Version 1

対象：

String オブジェクト

参照
関連項目
fontcolor メソッド

fromCharCode メソッド

1 つ以上の Unicode のコード値から 1 つの文字列を作成します。

```
function fromCharCode([code1 : Number [, ... [, codeN : Number]]]]) : String
```

引数
code1, ... , *codeN*

省略可能です。文字列に変換する文字の Unicode でのコード値を指定します。省略した場合は空の文字列になります。

解説

fromCharCode メソッドは、グローバルな String オブジェクトから呼び出されます。

使用例

`test` に "plain" という文字列を代入する例を次に示します。

```
var test = String.fromCharCode(112, 108, 97, 105, 110);
```

必要条件

Version 3

対象

String オブジェクト

参照
関連項目
charCodeAt メソッド

getDate メソッド

Date オブジェクトに格納されている日付の値を現地時間で返します。

```
function getDate() : Number
```

解説

世界協定時刻 (UTC) を使用して日付の値を取得するには、**getUTCDate** メソッドを使用します。

戻り値は、**Date** オブジェクトに格納されている日付の日の部分を表す 1 ～ 31 の範囲内の整数です。

使用例

getDate メソッドの使用例を次に示します。

```
function DateDemo(){
   var d, s = "Today's date is: ";
   d = new Date();
   s += (d.getMonth() + 1) + "/";
   s += d.getDate() + "/";
   s += d.getYear();
   return(s);
}
```

必要条件

Version 1

対象

Date オブジェクト

参照

関連項目

getUTCDate メソッド
setDate メソッド
setUTCDate メソッド

getDay メソッド

Date オブジェクトに格納されている曜日の値を現地時間で返します。

```
function getDay() : Number
```

解説

世界協定時刻 (UTC) を使用して曜日を取得するには、**getUTCDay** メソッドを使用します。

getDay メソッドの戻り値は、曜日を表す 0 ～ 6 の範囲内の数値です。各数値が表す曜日は次のとおりです。

値	曜日
0	日曜日
1	月曜日
2	火曜日
3	水曜日
4	木曜日
5	金曜日
6	土曜日

getDay メソッドの使用例を次に示します。

```
function DateDemo(){
   var d, day, x, s = "Today is: ";
   var x = new Array("Sunday", "Monday", "Tuesday");
   var x = x.concat("Wednesday","Thursday", "Friday");
   var x = x.concat("Saturday");
   d = new Date();
   day = d.getDay();
   return(s += x[day]);
}
```

必要条件

Version 1

対象：

Date オブジェクト

参照

関連項目

getUTCDay メソッド

getFullYear メソッド

Date オブジェクトに格納されている年の値を現地時間で返します。

```
function .getFullYear() : Number
```

解説

世界協定時刻 (UTC) を使用して年を取得するには、**getUTCFullYear** メソッドを使用します。

getFullYear メソッドの戻り値は 4 桁の年です。たとえば、1976 年の場合は、"1976" を返します。これにより、2000 年 1 月 1 日を 1900 年 1 月 1 日と誤って認識するという 2000 年問題を回避できます。

getFullYear メソッドの使用例を次に示します。

```
function DateDemo(){
   var d, s = "Today's date is: ";
   d = new Date();
   s += (d.getMonth() + 1) + "/";
   s += d.getDate() + "/";
   s += d.getFullYear();
   return(s);
}
```

必要条件

Version 3

対象

Date オブジェクト

参照

関連項目

getUTCFullYear メソッド
setFullYear メソッド
setUTCFullYear メソッド

getHours メソッド

Date オブジェクトに格納されている時 (時間) の値を現地時間で返します。

```
function getHours() : Number
```

解説

世界協定時刻 (UTC) を使用して時の値を取得するには、**getUTCHours** メソッドを使用します。

getHours メソッドの戻り値は、Date オブジェクトに格納されている時刻の 0 時からの経過時間を表す 0 ～ 23 の範囲内の整数です。ただし、戻り値が 0 のときは、午前 1:00:00 より前の場合の他に、**Date** オブジェクトが作成されたときにオブジェクトに時刻データが格納されなかった場合を考慮する必要があります。どちらの場合かを判断するには、分および秒の値も調べます。**Date** オブジェクトに時刻データが格納されていない場合は、時、分、秒のすべてが 0 になります。

getHours メソッドの使用例を次に示します。

```
function TimeDemo(){
   var d, s = "The current local time is: ";
   var c = ":";
   d = new Date();
   s += d.getHours() + c;
   s += d.getMinutes() + c;
   s += d.getSeconds() + c;
   s += d.getMilliseconds();
   return(s);
}
```

必要条件

Version 1

対象

Date オブジェクト

参照

関連項目

getUTCHours メソッド
setHours メソッド
setUTCHours メソッド

getItem メソッド

VBArray オブジェクトの指定した位置にある項目を返します。

```
function getItem(dimension1 : Number [, ...], dimensionN : Number) : Object
```

引数
dimension1, ..., dimensionN

必要な VBArray の要素の正確な位置。引数の数は、VBArray の次元数と一致する必要があります。

使用例

次のコードは、3 つの部分から構成されます。最初の部分は、Visual Basic のセーフ配列を作成する VBScript のコードです。2 番目の部分は、Visual Basic のセーフ配列に対して反復処理を行い、各要素の内容を出力する JScript のコードです。どちらのコードも、HTML ページの <HEAD> セクションに記述します。3 番目の部分は、他の 2 つのコードを実行するために <BODY> セクションに記述する JScript のコードです。

```
<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Function CreateVBArray()
   Dim i, j, k
   Dim a(2, 2)
   k = 1
   For i = 0 To 2
      For j = 0 To 2
         a(i, j) = k
         document.writeln(k)
         k = k + 1
      Next
      document.writeln("<BR>")
   Next
   CreateVBArray = a
End Function
-->
</SCRIPT>
<SCRIPT LANGUAGE="JScript">
<!--
function GetItemTest(vbarray)
{
   var i, j;
   var a = new VBArray(vbarray);
   for (i = 0; i <= 2; i++)
   {
      for (j =0; j <= 2; j++)
      {
         document.writeln(a.getItem(i, j));
      }
   }
}-->
</SCRIPT>
</HEAD>
&ltBODY>
<SCRIPT LANGUAGE="JScript">
<!--
   GetItemTest(CreateVBArray());
-->
</SCRIPT>
</BODY>
```

必要条件

Version 1

対象：

VBArray オブジェクト

参照
関連項目
dimensions メソッド
lbound メソッド
toArray メソッド
ubound メソッド

getMilliseconds メソッド

Date オブジェクトに格納されているミリ秒の値を現地時間で返します。

```
function getMilliseconds() : Number
```

解説

世界協定時刻 (UTC) を使用してミリ秒の値を取得するには、**getUTCMilliseconds** メソッドを使用します。

戻り値のミリ秒の範囲は、0 ～ 999 です。

使用例

getMilliseconds メソッドの使用例を次に示します。

```
function TimeDemo(){
   var d, s = "The current local time is: ";
   var c = ":";
   d = new Date();
   s += d.getHours() + c;
   s += d.getMinutes() + c;
   s += d.getSeconds() + c;
   s += d.getMilliseconds();
   return(s);
}
```

必要条件

Version 3

対象：

Date オブジェクト

参照

関連項目

getUTCMilliseconds メソッド
setMilliseconds メソッド
setUTCMilliseconds メソッド

getMinutes メソッド

ローカル時間を使用して Date オブジェクトの分の値を返します。

```
function getMinutes() : Number
```

解説

世界協定時刻 (UTC) を使用して分の値を取得するには、**getUTCMinutes** メソッドを使用します。

getMinutes メソッドの戻り値は、**Date** オブジェクトに格納されている時刻の分の部分を表す 0 ～ 59 の範囲内の整数です。ただし、戻り値が 0 のときは、午前 1:00:00 より前の場合の他に、**Date** オブジェクトが作成されたときにオブジェクトに時刻データが格納されなかった場合を考慮する必要があります。どちらの場合かを判断するには、時および秒の値も調べます。**Date** オブジェクトに時刻データが格納されていない場合は、時、分、秒のすべてが 0 になります。

使用例

getMinutes メソッドの使用例を次に示します。

```
function TimeDemo(){
   var d, s = "The current local time is: ";
   var c = ":";
   d = new Date();
   s += d.getHours() + c;
   s += d.getMinutes() + c;
   s += d.getSeconds() + c;
   s += d.getMilliseconds();
   return(s);
}
```

必要条件

Version 3

対象：

Date オブジェクト

参照

関連項目

getUTCMinutes メソッド
setMinutes メソッド
setUTCMinutes メソッド

getMonth メソッド

Date オブジェクトに格納されている月の値を現地時間で返します。

```
function getMonth() : Number
```

解説

世界協定時刻 (UTC) を使用して月の値を取得するには、**getUTCMonth** メソッドを使用します。

getMonth メソッドの戻り値は、**Date** オブジェクトに格納されている日付の月の部分を表す 0 ～ 11 の範囲内の整数です。ただし、返される整数の月は一般に使用される数字ではなく、一般に表す数字より 1 小さい値です。たとえば、**Date** オブジェクトに "Jan 5, 1996 08:47:00" というデータが格納されている場合は、0 が返されます。

使用例

getMonth メソッドの使用例を次に示します。

```
function DateDemo(){
   var d, s = "Today's date is: ";
   d = new Date();
   s += (d.getMonth() + 1) + "/";
   s += d.getDate() + "/";
   s += d.getYear();
   return(s);
}
```

必要条件

Version 1

対象：

Date オブジェクト

参照

関連項目

getUTCMonth メソッド
setMonth メソッド
setUTCMonth メソッド

getSeconds メソッド

ローカル時間を使用して Date オブジェクトの秒の値を返します。

```
function getSeconds() : Number
```

解説

世界協定時刻 (UTC) を使用して秒の値を取得するには、**getUTCSeconds** メソッドを使用します。

getSeconds メソッドの戻り値は、**Date** オブジェクトに格納されている時刻の秒の部分を表す 0 ～ 59 の範囲内の整数です。戻り値は、2 つの状況で 0 になります。現在の分の最初の 1 秒を経過していない場合と、**Date** オブジェクトが作成されたときにオブジェクトに時刻データが格納されなかった場合です。どちらの場合かを判断するには、時および分の値も調べます。**Date** オブジェクトに時刻データが格納されていない場合は、時、分、秒のすべてが 0 になります。

使用例

getSeconds メソッドの使用例を次に示します。

```
function TimeDemo(){
   var d, s = "The current local time is: ";
   var c = ":";
   d = new Date();
   s += d.getHours() + c;
   s += d.getMinutes() + c;
   s += d.getSeconds() + c;
   s += d.getMilliseconds();
   return(s);
}
```

必要条件

Version 1

対象

Date オブジェクト

参照

関連項目

getUTCSeconds メソッド
setSeconds メソッド
setUTCSeconds メソッド

getTime メソッド

Date オブジェクトに格納されている時刻の値を返します。

```
function getTime() : Number
```

解説

getTime メソッドの戻り値は、1970 年 1 月 1 日午前 00:00:00 と **Date** オブジェクトに格納されている時刻との差をミリ秒単位で表す整数値です。使用できる日付の範囲は、概算で 1970 年 1 月 1 日午前 00:00:00 から 285,616 年 12 月 31 日午後 11:59:59 までです。負の数値は、1970 年以前の日付を示します。

複数の日付データや時刻データを使って計算を行う場合は、1 日、1 時間、1 分間などの時間をミリ秒単位で表す変数を定義しておくと便利です。次に例を示します。

```
var MinMilli = 1000 * 60
var HrMilli = MinMilli * 60
var DyMilli = HrMilli * 24
```

使用例

getTime メソッドの使用例を次に示します。

```
function GetTimeTest(){
   var d, s, t;
   var MinMilli = 1000 * 60;
   var HrMilli = MinMilli * 60;
   var DyMilli = HrMilli * 24;
   d = new Date();
   t = d.getTime();
   s = "It's been "
   s += Math.round(t / DyMilli) + " days since 1/1/70";
   return(s);
}
```

必要条件

Version 1

対象

Date オブジェクト

参照

関連項目

setTime メソッド

getTimezoneOffset メソッド

ホスト コンピュータの時刻と世界協定時刻 (UTC) との差を分単位の値で返します。

```
function getTimezoneOffset() : Number
```

解説

getTimezoneOffset メソッドの戻り値は、現在のコンピュータの時刻と UTC の時刻との差を分単位で表す整数です。戻り値は、スクリプトが実行されているコンピュータの時刻を基にした値になります。サーバー側のスクリプトから呼び出された場合は、戻り値はサーバーの時刻を基にした値になります。クライアント側のスクリプトから呼び出された場合は、戻り値はクライアントの時刻を基にした値になります。

コンピュータの時刻が UTC より遅い場合 (太平洋夏時間など) は正の値に、UTC より早い場合 (日本時間など) は負の値になります。

たとえば、ニューヨークのサーバーが、12 月 1 日にロサンゼルスのクライアントからのアクセスを受けたとします。この場合、**getTimezoneOffset** メソッドは、クライアント側のスクリプトで実行されると 480 を返し、サーバー側のスクリプトで実行されると 300 を返します。

使用例

getTimezoneOffset メソッドの使用例を次に示します。

```
function TZDemo(){
   var d, tz, s = "The current local time is ";
   d = new Date();
   tz = d.getTimezoneOffset();
   if (tz < 0)
      s += tz / 60 + " hours before UTC";
   else if (tz == 0)
      s += "UTC";
   else
      s += tz / 60 + " hours after UTC";
   return(s);
}
```

必要条件

Version 1

対象

Date オブジェクト

参照

その他の技術情報

メソッド

getUTCDate メソッド

世界協定時刻 (UTC) を使用して **Date** オブジェクトの日付の値を返します。

```
function getUTCDate() : Number
```

解説

ローカル時間を使用して日付の値を取得するには、**getDate** メソッドを使用します。

戻り値は、**Date** オブジェクトに格納されている日付の日の部分を表す 1 ～ 31 の範囲内の整数です。

使用例

getUTCDate メソッドの使用例を次に示します。

```
function UTCDateDemo(){
   var d, s = "Today's UTC date is: ";
   d = new Date();
   s += (d.getUTCMonth() + 1) + "/";
   s += d.getUTCDate() + "/";
   s += d.getUTCFullYear();
   return(s);
}
```

必要条件

Version 3

対象

Date オブジェクト

参照

関連項目

getDate メソッド
setDate メソッド
setUTCDate メソッド

getUTCDay メソッド

Date オブジェクトに格納されている曜日の値を世界協定時刻 (UTC) で返します。

```
function getUTCDay() : Number
```

解説

ローカル時間を使用して曜日を取得するには、getDate メソッドを使用します。

getUTCDay メソッドの戻り値は、Date オブジェクトの曜日の部分を表す 0 ～ 6 の範囲内の整数です。各整数が表す曜日は次のとおりです。

値	曜日
0	日曜日
1	月曜日
2	火曜日
3	水曜日
4	木曜日
5	金曜日
6	土曜日

使用例

getUTCDay メソッドの使用例を次に示します。

```
function DateDemo(){
   var d, day, x, s = "Today is ";
   var x = new Array("Sunday", "Monday", "Tuesday");
   x = x.concat("Wednesday","Thursday", "Friday");
   x = x.concat("Saturday");
   d = new Date();
   day = d.getUTCDay();
   return(s += x[day] + " in UTC");
}
```

必要条件

Version 3

対象

Date オブジェクト

参照

関連項目

getDay メソッド

getUTCFullYear メソッド

Date オブジェクトに格納されている年の値を世界協定時刻 (UTC) で返します。

```
function getUTCFullYear() : Number
```

解説

ローカル時間を使用して年を取得するには、**getFullYear** メソッドを使用します。

getUTCFullYear メソッドの戻り値は 4 桁の年です。これにより、2000 年 1 月 1 日を 1900 年 1 月 1 日と誤って認識するという 2000 年問題を回避できます。

使用例

getUTCFullYear メソッドの使用例を次に示します。

```
function UTCDateDemo(){
   var d, s = "Today's UTC date is: ";
   d = new Date();
   s += (d.getUTCMonth() + 1) + "/";
   s += d.getUTCDate() + "/";
   s += d.getUTCFullYear();
   return(s);
}
```

必要条件

Version 3

対象

Date オブジェクト

参照

関連項目

getFullYear メソッド
setFullYear メソッド
setUTCFullYear メソッド

getUTCHours メソッド

Date オブジェクトに格納されている時の値を世界協定時刻 (UTC) で返します。

```
function getUTCHours() : Number
```

解説

ローカル時間での 0 時からの時単位の経過時間を取得するには、**getHours** メソッドを使用します。

getUTCHours メソッドの戻り値は、Date オブジェクトの 0 時からの経過時間を時単位で表す 0 ～ 23 の範囲内の整数です。ただし、戻り値が 0 のときは、午前 1:00:00 より前の場合の他に、**Date** オブジェクトが作成されたときにオブジェクトに時刻データが格納されなかった場合を考慮する必要があります。どちらの場合かを判断するには、分および秒の値も調べます。**Date** オブジェクトに時刻データが格納されていない場合は、時、分、秒のすべてが 0 になります。

使用例

getUTCHours メソッドの使用例を次に示します。

```
function UTCTimeDemo(){
   var d, s = "Current Coordinated Universal Time (UTC) is: ";
   var c = ":";
   d = new Date();
   s += d.getUTCHours() + c;
   s += d.getUTCMinutes() + c;
   s += d.getUTCSeconds() + c;
   s += d.getUTCMilliseconds();
   return(s);
}
```

必要条件

Version 3

対象

Date オブジェクト

参照

関連項目

getHours メソッド
setHours メソッド
setUTCHours メソッド

getUTCMilliseconds メソッド

Date オブジェクトに格納されているミリ秒の値を世界協定時刻 (UTC) で返します。

```
function getUTCMilliseconds() : Number
```

解説

ローカル時間でのミリ秒の値を取得するには、**getMilliseconds** メソッドを使用します。

戻り値のミリ秒の範囲は、0 ～ 999 です。

使用例

getUTCMilliseconds メソッドの使用例を次に示します。

```
function UTCTimeDemo(){
   var d, s = "Current Coordinated Universal Time (UTC) is: ";
   var c = ":";
   d = new Date();
   s += d.getUTCHours() + c;
   s += d.getUTCMinutes() + c;
   s += d.getUTCSeconds() + c;
   s += d.getUTCMilliseconds();
   return(s);
}
```

必要条件

Version 3

対象

Date オブジェクト

参照

関連項目

getMilliseconds メソッド
setMilliseconds メソッド
setUTCMilliseconds メソッド

getUTCMinutes メソッド

Date オブジェクトに格納されている分の値を世界協定時刻 (UTC) で返します。

```
function getUTCMinutes() : Number
```

解説

ローカル時間を使用して分の値を取得するには、**getMinutes** メソッドを使用します。

getUTCMinutes メソッドの戻り値は、**Date** オブジェクトの分の値を表す 0 ～ 59 の範囲内の整数です。ただし、戻り値が 0 のときは、オブジェクトに格納されていた時刻が 1 時間のうちの最初の 1 分を経過していない時刻の場合の他に、**Date** オブジェクトが作成されたときにオブジェクトに時刻データが格納されなかった場合を考慮する必要があります。どちらの場合かを判断するには、時および秒の値も調べます。**Date** オブジェクトに時刻データが格納されていない場合は、時、分、秒のすべてが 0 になります。

使用例

getUTCMinutes メソッドの使用例を次に示します。

```
function UTCTimeDemo()
{
   var d, s = "Current Coordinated Universal Time (UTC) is: ";
   var c = ":";
   d = new Date();
   s += d.getUTCHours() + c;
   s += d.getUTCMinutes() + c;
   s += d.getUTCSeconds() + c;
   s += d.getUTCMilliseconds();
   return(s);
}
```

必要条件

Version 3

対象

Date オブジェクト

参照

関連項目

getMinutes メソッド
setMinutes メソッド
setUTCMinutes メソッド

getUTCMonth メソッド

Date オブジェクトに格納されている月の値を世界協定時刻 (UTC) で返します。

```
function getUTCMonth(): Number
```

解説

ローカル時間を使用して月の値を取得するには、**getMonth** メソッドを使用します。

getUTCMonth メソッドの戻り値は、Date オブジェクトの月の値を表す 0 ～ 11 の範囲内の整数です。ただし、返される整数の月は一般に使用される数字ではなく、一般に表す数字より 1 小さい値です。たとえば、**Date** オブジェクトに "Jan 5, 1996 08:47:00.0" というデータが格納されている場合、**getUTCMonth** メソッドは 0 を返します。

使用例

getUTCMonth メソッドの使用例を次に示します。

```
function UTCDateDemo(){
   var d, s = "Today's UTC date is: ";
   d = new Date();
   s += (d.getUTCMonth() + 1) + "/";
   s += d.getUTCDate() + "/";
   s += d.getUTCFullYear();
   return(s);
}
```

必要条件

Version 3

対象

Date オブジェクト

参照

関連項目

getMonth メソッド
setMonth メソッド
setUTCMonth メソッド

getUTCSeconds メソッド

Date オブジェクトに格納されている秒の値を世界協定時刻 (UTC) で返します。

```
function getUTCSeconds(): Number
```

解説

ローカル時間を使用して秒の値を取得するには、**getSeconds** メソッドを使用します。

getUTCSeconds メソッドの戻り値は、**Date** オブジェクトに格納されている時刻の秒の部分を表す 0 ～ 59 の範囲内の整数です。ただし、戻り値が 0 のときは、オブジェクトに格納されていた時刻が 1 分のうちの最初の 1 秒を経過していない時刻の場合の他に、**Date** オブジェクトが作成されたときにオブジェクトに時刻データが格納されなかった場合を考慮する必要があります。どちらの場合かを判断するために、時および分の値も調べます。**Date** オブジェクトに時刻データが格納されていない場合は、時、分、秒のすべてが 0 になります。

使用例

getUTCSeconds メソッドの使用例を次に示します。

```
function UTCTimeDemo(){
   var d, s = "Current Coordinated Universal Time (UTC) is: ";
   var c = ":";
   d = new Date();
   s += d.getUTCHours() + c;
   s += d.getUTCMinutes() + c;
   s += d.getUTCSeconds() + c;
   s += d.getUTCMilliseconds();
   return(s);
}
```

必要条件

Version 3

対象

Date オブジェクト

参照

関連項目

getSeconds メソッド
setSeconds メソッド
setUTCSeconds メソッド

getVarDate メソッド

Date オブジェクトの VT_DATE 値を返します。

```
function getVarDate(): System.DateTime
```

解説

getVarDate メソッドは、VT_DATE 形式で日付の値をやり取りする COM オブジェクトや ActiveX® オブジェクトなどのオブジェクトを扱う場合に使用します。たとえば、Visual Basic や VBScript などがあります。実際の形式は地域設定によって変わるため、JScript の中では実際の形式に依存しないようにしてください。

必要条件

Version 3

対象

Date オブジェクト

参照
関連項目
getDate メソッド
parse メソッド

getYear メソッド

Date オブジェクトに格納されている年の値を返します。

```
function getYear(): Number
```

解説

getYear メソッドは廃止されました。現在は以前のバージョンとの互換性のためだけに残されています。代わりに、**getFullYear** メソッドの使用をお勧めします。

getYear メソッドでは、Date オブジェクトに格納されている年が 1900 年から 1999 年の間の場合は、1900 年との差を表す "年" を 2 桁の整数値として返します。それ以外の場合は、4 桁の整数値が返されます。たとえば、Date オブジェクトに格納されている年が 1996 年の場合は、年数を 96 で返し、1825 年や 2025 年の場合は、その 4 桁の値をそのまま返します。

メモ :

JScript バージョン 1.0 では、**getYear** メソッドは年を表す値にかかわらず、**Date** オブジェクトに指定された年の値から 1900 を引いた結果を値として返します。たとえば、Date オブジェクトに格納されている年が 1899 年の場合は年数を -1 で返し、2000 年の場合は、100 を返します。

使用例

getYear メソッドの使用例を次に示します。

```
function DateDemo(){
   var d, s = "Today's date is: ";
   d = new Date();
   s += (d.getMonth() + 1) + "/";
   s += d.getDate() + "/";
   s += d.getYear();
   return(s);
}
```

必要条件

Version 1

対象

Date オブジェクト

参照

関連項目

getFullYear メソッド
getUTCFullYear メソッド
setFullYear メソッド
setUTCFullYear メソッド
setYear メソッド

hasOwnProperty メソッド

指定した名前のプロパティがオブジェクトにあるかどうかを表すブール値を返します。

```
function hasOwnProperty(proName : String) : Boolean
```

引数
proName

必ず指定します。プロパティ名の文字列値を指定します。

解説

指定した名前のプロパティがオブジェクトにある場合、**hasOwnProperty** メソッドは **true** を返し、ない場合は **false** を返します。このメソッドでは、プロパティがオブジェクトのプロトタイプ チェインに存在するかどうかはチェックされません。プロパティはオブジェクトのメンバである必要があります。

使用例

すべての **String** オブジェクトが共通メソッド split を共有している場合の例を次に示します。

```
var s = new String("JScript");
print(s.hasOwnProperty("split"));
print(String.prototype.hasOwnProperty("split"));
```

このプログラムの出力は次のようになります。

```
false
true
```

必要条件

Version 5.5

対象：

Object オブジェクト

参照
関連項目
in 演算子

indexOf メソッド

String オブジェクト (文字列) 内を先頭から、指定された文字列で検索します。

```
function indexOf(subString : String [, startIndex : Number]) : Number
```

引数
subString

必ず指定します。**String** オブジェクト内の検索する文字列を指定します。

startIndex

省略可能です。**String** オブジェクト内での検索開始位置番号を整数値で指定します。省略した場合は、文字列の先頭から検索が開始されます。

解説

indexOf メソッドの戻り値は、**String** オブジェクト内で見つかった検索文字列の先頭位置を示す整数値です。検索文字列が見つからなかった場合は、-1 が返されます。

引数 *startIndex* に負の値を指定した場合は、0 として処理されます。また、最大位置番号より大きい値を指定した場合は、最大位置番号として処理されます。

検索は左から右へと行われます。この点を除けば、このメソッドは **lastIndexOf** と同じ処理を行います。

使用例

indexOf メソッドの使用例を次に示します。

```
function IndexDemo(str2){
   var str1 = "BABEBIBOBUBABEBIBOBU"
   var s = str1.indexOf(str2);
   return(s);
}
```

必要条件

Version 1

対象

String オブジェクト

参照
関連項目
lastIndexOf メソッド

isFinite メソッド

指定した数値が有限かどうかを表すブール値を返します。

```
function isFinite(number : Number) : Boolean
```

引数
number

必ず指定します。数値を指定します。

解説

引数 *number* に指定した値が **NaN**、負の無限大、正の無限大のいずれでもない場合、真 (**true**) が返されます。いずれかに該当する場合は、偽 (**false**) が返されます。

必要条件

Version 3

対象：

Global オブジェクト

参照
関連項目
isNaN メソッド

isNaN メソッド

指定した値が予約済みの値 **NaN** (非数) かどうかを表すブール値を返します。

```
function isNaN(number : Number) : Boolean
```

引数
number

必ず指定します。数値を指定します。

解説

isNaN メソッドは、指定した値が **NaN** である場合は真 (**true**) を返し、それ以外の場合は偽 (**false**) を返します。通常、**parseInt** メソッドや **parseFloat** メソッドからの戻り値を調べるために使用します。

また、変数をその変数自体と比較して、NaN かどうかを調べることもできます。値が等しいと評価されない場合、その変数は **NaN** となります。**NaN** は、自身と比較して等しいと評価されない唯一の値です。

必要条件

Version 1

対象：

Global オブジェクト

参照
関連項目
isFinite メソッド
NaN プロパティ (Global)
parseFloat メソッド
parseInt メソッド

isPrototypeOf メソッド

オブジェクトが、別のオブジェクトのプロトタイプ チェインに存在するかどうかを表すブール値を返します。

```
function isPrototypeOf(obj : Object) : Boolean
```

引数
obj

必ず指定します。プロトタイプ チェインをチェックするオブジェクトを指定します。

解説

obj のプロトタイプ チェインに現在のオブジェクトがある場合、**isPrototypeOf** メソッドは **true** を返します。プロトタイプ チェインは、同じオブジェクト型のインスタンス間で機能を共有するときに使用します。*obj* がオブジェクトでない場合、または現在のオブジェクトが *obj* のプロトタイプ チェインにない場合、**isPrototypeOf** メソッドは **false** を返します。

使用例

isPrototypeof メソッドの使用例を次に示します。

```
function test(){
   var re = new RegExp();                        //Initialize variable.
   return (RegExp.prototype.isPrototypeOf(re));  //Return true.
}
```

必要条件

Version 5.5

対象：

Object オブジェクト

参照
その他の技術情報
メソッド

italics メソッド

文字列オブジェクトのテキストを HTML の <I> タグと </I> タグで囲み、文字列として返します。

```
function italics() : String
```

解説

このメソッドの実行時には、タグが既に元の文字列内にあるかどうかの確認は行われません。

使用例

italics メソッドの使用例を次に示します。

```
var strVariable = "This is a string";
strVariable = strVariable.italics();
```

2 つ目のステートメントの実行後、変数 `strVariable` は次の値になります。

```
<I>This is a string</I>
```

必要条件

Version 1

対象

String オブジェクト

参照

関連項目

bold メソッド

item メソッド (JScript)

コレクション内の現在の項目を返します。

```
function item() : Number
```

解説

item メソッドは、**Enumerator** オブジェクトの現在の項目を返します。コレクションが空の場合、または現在の項目が未定義の場合は、**undefined** を返します。

使用例

item メソッドを使って **Drives** コレクションのメンバを返すコード例を次に示します。

```
function ShowDriveList(){
   var fso, s, n, e, x;
   fso = new ActiveXObject("Scripting.FileSystemObject");
   e = new Enumerator(fso.Drives);
   s = "";
   for (; !e.atEnd(); e.moveNext())
   {
      x = e.item();
      s = s + x.DriveLetter;
      s += " - ";
      if (x.DriveType == 3)
         n = x.ShareName;
      else if (x.IsReady)
         n = x.VolumeName;
      else
         n = "[Drive not ready]";
      s +=  n + "<br>";
   }
   return(s);
}
```

必要条件

Version 3

対象

Enumerator オブジェクト

参照

関連項目

atEnd メソッド
moveFirst メソッド
moveNext メソッド

メソッド (J ～ R)

メソッドは、オブジェクトのメンバとなっている関数です。ここでは、名前が J ～ R で始まるメソッドを示します。

このセクションの内容

0 ～ 1 の範囲の擬似乱数を返します。

replace メソッド

正規表現を使って置換された文字列のコピーを返します。

reverse メソッド

Array オブジェクトの要素を反転させます。

round メソッド

指定された数式の値に最も近い整数を返します。

関連するセクション

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

メソッド

JScript で使用できるメソッドのアルファベット順の一覧と、メソッドの各カテゴリへのリンクを示します。

オブジェクト

JScript のオブジェクトの概念と、オブジェクトがプロパティとメソッドにどのように関連するかについて説明します。JScript がサポートするオブジェクトの詳細を説明するトピックへのリンクも示します。

join メソッド

指定した配列の各要素を指定した区切り文字で連結して 1 つの文字列値として返します。

```
function join(separator : String) : String
```

引数
separator

必ず指定します。結果の **String** オブジェクトで配列要素の区切り文字として使用する文字列を指定します。省略した場合、各配列要素はコンマで区切られます。

解説

undefined または **NULL** の要素は、空の文字列として処理されます。

使用例

join メソッドの使用例を次に示します。

```
function JoinDemo(){
   var a, b;
   a = new Array(0,1,2,3,4);
   b = a.join("-");
   return(b);
}
```

必要条件

Version 2

対象：

Array オブジェクト

参照
関連項目
String オブジェクト

lastIndexOf メソッド

String オブジェクト (文字列) 内を指定された検索文字列で後方から検索し、一致した部分文字列のインデックスを返します。

```
function lastIndexOf(substring : String [, startindex : Number ]) : Number
```

引数
substring

必ず指定します。**String** オブジェクト内の検索する文字列を指定します。

startindex

省略可能です。**String** オブジェクト内での検索開始位置番号を整数値で指定します。省略した場合は、文字列の末尾から検索が開始されます。

解説

lastIndexOf メソッドの戻り値は、String オブジェクト内で見つかった検索文字列の先頭位置を示す整数値です。検索文字列が見つからなかった場合は、-1 が返されます。

引数 *startindex* で負の値を指定した場合は、0 として処理されます。また、最大位置番号より大きい値を指定した場合は、最大位置番号として処理されます。

検索は右から左へと行われます。この点を除けば、このメソッドは **indexOf** と同じ処理を行います。

使用例

lastIndexOf メソッドの使用例を次に示します。

```
function lastIndexDemo(str2) {
   var str1 = "BABEBIBOBUBABEBIBOBU"
   var s = str1.lastIndexOf(str2);
   return(s);
}
```

必要条件

Version 1

対象

String オブジェクト

参照
関連項目
indexOf メソッド

lbound メソッド

VBArray オブジェクトの配列で、指定した次元で使用されている最小のインデックス値を返します。

```
function lbound([dimension : Number]) : Object
```

引数
dimension

省略可能です。VBArray の、最も小さいインデックスを取得する次元を指定します。省略した場合は、1 を仮定します。

解説

VBArray が空の場合、**lbound** メソッドは undefined を返します。引数 *dimension* に VBArray の次元数より大きい値を指定したり、負の値を指定したりすると、"有効範囲外のインデックス" エラーが発生します。

使用例

次のコードは、3 つの部分から構成されます。最初の部分は、Visual Basic のセーフ配列を作成する VBScript のコードです。2 番目の部分は、このセーフ配列の次元数と各次元の下限値を取得する JScript のコードです。セーフ配列は、Visual Basic というよりもむしろ VBScript で作成されるため、下限値は常に 0 になります。どちらのコードも、HTML ページの <HEAD> セクションに記述します。3 番目の部分は、他の 2 つのコードを実行するために <BODY> セクションに記述する JScript のコードです。

```
<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Function CreateVBArray()
   Dim i, j, k
   Dim a(2, 2)
   k = 1
   For i = 0 To 2
      For j = 0 To 2
         a(j, i) = k
         k = k + 1
      Next
   Next
   CreateVBArray = a
End Function
-->
</SCRIPT>

<SCRIPT LANGUAGE="JScript">
<!--
function VBArrayTest(vba){
   var i, s;
   var a = new VBArray(vba);
   for (i = 1; i <= a.dimensions(); i++)
   {
      s = "The lower bound of dimension ";
      s += i + " is ";
      s += a.lbound(i)+ ".<BR>";
      return(s);
   }
}
-->
</SCRIPT>
</HEAD>

<BODY>
<SCRIPT language="jscript">
   document.write(VBArrayTest(CreateVBArray()));
</SCRIPT>
</BODY>
```

必要条件

Version 3

対象

VBArray オブジェクト

参照

関連項目

dimensions メソッド
getItem メソッド
toArray メソッド
ubound メソッド

link メソッド

String オブジェクトのテキストを HTML の HREF 属性付きの <A> タグと </A> タグで囲み、文字列として返します。

```
function link(linkstring : String) : String
```

解説

この **link** メソッドを呼び出すと、**String** オブジェクトからのハイパーリンクが作成されます。

このメソッドの実行時には、タグが既に元の文字列内にあるかどうかの確認は行われません。

使用例

このメソッドの使用例を次に示します。

```
var strVariable = "This is a hyperlink";
strVariable = strVariable.link("http://www.microsoft.com");
```

2 つ目のステートメントの実行後、変数 `strVariable` は次の値になります。

```
<A HREF="http://www.microsoft.com">This is a hyperlink</A>
```

必要条件

Version 1

対象：

String オブジェクト

参照

関連項目

anchor メソッド

localeCompare メソッド

現在のロケールにおいて、2 つの文字列が等しいかどうかを表す値を返します。

```
function localeCompare(stringExp : String) : Number
```

引数
stringExp

必ず指定します。現在の文字列オブジェクトと比較する文字列を指定します。

解説

localeCompare では、ロケールに基づいて現在の文字列オブジェクトと *stringExp* の文字列を比較し、システムの既定のロケールの並べ替え順序に従って -1、0、または +1 を返します。

現在の文字列オブジェクトが *stringExp* の前になる場合、**localeCompare** は –1 を返します。現在の文字列オブジェクトが *stringExp* の後ろになる場合は +1 を返します。戻り値がゼロ (0) の場合は、2 つの文字列は等しいことを表します。

必要条件

Version 5.5

対象：

String オブジェクト

参照
関連項目
toLocaleString メソッド

log メソッド

指定された数値の自然対数を返します。

```
function log(number : Number) : Number
```

引数
number

必ず指定します。数値を指定します。

解説

戻り値は、引数 *number* の自然対数です。*e* が底となります。

必要条件

Version 1

対象：

Math オブジェクト

参照
その他の技術情報
メソッド

match メソッド

正規表現パターンを使って文字列に対して検索を実行し、検索結果を配列に格納して返します。

```
function match(rgExp : RegExp) : Array
```

引数
rgExp

必ず指定します。正規表現パターンおよび適用できるフラグを含む **Regular Expression** オブジェクトのインスタンスを指定します。正規表現パターンおよびフラグを含む変数名または文字列リテラルを指定することもできます。

解説

パターンに一致する文字列が見つからなかった場合、**match** メソッドは **null** を返します。一致する文字列が見つかった場合は、配列を返し、さらにグローバルな **RegExp** オブジェクトのプロパティが検索結果を反映して更新されます。

match メソッドが返す配列には、**input**、**index**、および **lastIndex.** の 3 つのプロパティがあります。**input** プロパティには、検索された文字列全体が格納されます。**index** プロパティには、検索された文字列内の一致した部分文字列の位置が格納されます。**lastIndex** プロパティには、最後に一致したサブストリングの最後の文字の直後の位置が格納されます。

グローバル フラグ (**g**) が設定されていない場合、配列の要素 0 には一致した文字列全体が、要素 1 から n には、一致した文字列の中に副次的に含まれている内容が格納されます。この処理は、グローバルなフラグが設定されていない場合の **exec** メソッドの処理と同じです。グローバル フラグが設定されている場合、要素 0 から *n* には一致したすべての文字列がそれぞれ格納されます。

使用例

match メソッドの使用例を次に示します。

```
function MatchDemo(){
   var r, re;         //Declare variables.
   var s = "The rain in Spain falls mainly in the plain";
   re = /ain/i;    //Create regular expression pattern.
   r = s.match(re);   //Attempt match on search string.
   return(r);         //Return first occurrence of "ain".
}
```

次のコードは、**g** フラグが設定されている場合の match メソッドの使用例を次に示します。

```
function MatchDemo(){
   var r, re;         //Declare variables.
   var s = "The rain in Spain falls mainly in the plain";
   re = /ain/ig;      //Create regular expression pattern.
   r = s.match(re);   //Attempt match on search string.
   return(r);         //Return array containing all four
                      // occurrences of "ain".
}
```

match メソッドの文字列リテラルの使用例を次に示します。

```
var r, re = "Spain";
r = "The rain in Spain".replace(re, "Canada");
```

必要条件

Version 3

対象

String オブジェクト

参照
関連項目

max メソッド

指定された 0 個以上の数式のうち、最も値の大きい数式を返します。

```
function max([number1 : Number [, ... [, numberN : Number]]]) : Number
```

引数
number1, ..., numberN

必ず指定します。数式を指定します。

解説

引数を指定しない場合、戻り値は **NEGATIVE_INFINITY** に等しくなります。引数 **NaN** を指定すると、戻り値も **NaN** になります。

必要条件

Version 1

対象

Math オブジェクト

参照
関連項目
min メソッド
NEGATIVE_INFINITY プロパティ

min メソッド

指定された 0 個以上の数式のうち、最も小さい値を持つ数式を返します。

```
function min([number1 : Number [, ... [, numberN : Number]]]) : Number
```

引数
number1, ..., numberN

必ず指定します。数式を指定します。

解説

引数を指定しない場合、戻り値は **POSITIVE_INFINITY** に等しくなります。引数 **NaN** を指定すると、戻り値も **NaN** になります。

必要条件

Version 1

対象

Math オブジェクト

参照
関連項目
max メソッド
POSITIVE_INFINITY プロパティ

moveFirst メソッド

Enumerator オブジェクト内の現在の項目を最初の項目に設定し直します。

```
function moveFirst()
```

解説

コレクション内に項目がない場合、現在の項目は undefined に設定されます。

使用例

moveFirst メソッドを使って **Drives** コレクションのメンバを先頭から評価する例を次に示します。

```
function ShowFirstAvailableDrive(){
   var fso, s, e, x;                //Declare variables.
   fso = new ActiveXObject("Scripting.FileSystemObject");
   e = new Enumerator(fso.Drives);  //Create Enumerator object.
   e.moveFirst();                   //Move to first drive.
   s = "";                          //Initialize s.
   do
   {
      x = e.item();                 //Test for existence of drive.
      if (x.IsReady)                //See if it's ready.
      {
         s = x.DriveLetter + ":";   //Assign 1<SUP>st</SUP> drive letter to s.
         break;
      }
      else
         if (e.atEnd())             //See if at the end of the collection.
         {
            s = "No drives are available";
            break;
         }
      e.moveNext();                 //Move to the next drive.
   }
   while (!e.atEnd());              //Do while not at collection end.
   return(s);                       //Return list of available drives.
}
```

必要条件

Version 3

対象：

Enumerator オブジェクト

参照

関連項目

atEnd メソッド
item メソッド (JScript)
moveNext メソッド

moveNext メソッド

Enumerator オブジェクト内の現在の項目を次の項目へ移動します。

```
function moveNext()
```

解説

列挙子がコレクション内の末尾位置である場合、またはコレクションが空の場合、現在の項目は undefined に設定されます。

moveNext メソッドを使って **Drives** コレクションの次の項目 (ドライブ) に移動する例を次に示します。

```
function ShowDriveList(){
   var fso, s, n, e, x;                 //Declare variables.
   fso = new ActiveXObject("Scripting.FileSystemObject");
   e = new Enumerator(fso.Drives);      //Create Enumerator object.
   s = "";                              //Initialize s.
   for (; !e.atEnd(); e.moveNext())
   {
      x = e.item();
      s = s + x.DriveLetter;            //Add drive letter
      s += " - ";                       //Add "-" character.
      if (x.DriveType == 3)
         n = x.ShareName;               //Add share name.
      else if (x.IsReady)
         n = x.VolumeName;              //Add volume name.
      else
         n = "[Drive not ready]";       //Indicate drive not ready.
      s +=   n + "\n";
   }
   return(s);                           //Return drive status.
}
```

必要条件

Version 3

対象：

Enumerator オブジェクト

参照

関連項目

atEnd メソッド
item メソッド (JScript)
moveFirst メソッド

parse メソッド

日付を表す文字列を解析し、その日付と 1970 年 1 月 1 日午前 00:00:00 との差を表すミリ秒単位の値を返します。

```
function parse(dateVal : {String | System.DateTime} ) : Number
```

引数
dateVal

必ず指定します。日付を特定の形式 ("Jan 5, 1996 08:47:00") で表した文字列、または ActiveX® などのオブジェクトから取得した VT_DATE 値を指定します。

解説

parse メソッドの戻り値は、指定された日付と 1970 年 1 月 1 日午前 00:00:00 との差をミリ秒単位で表す整数値です。

parse メソッドは **Date** オブジェクトの静的な (static) メソッドです。静的なメソッドなので、作成済の **Date** オブジェクトのメソッドとして呼び出されるよりも、むしろ次の使用例のような方法で呼び出されます。

```
var datestring = "November 1, 1997 10:15 AM";
Date.parse(datestring)
```

parse メソッドで正しく日付を解析させるには、次の条件を満たす文字列を指定する必要があります。

- 日付の指定には "/" または "-" を使用できます。ただし、指定する順序は米国で一般的に使用されている Short Date 形式 ("月/日/年") で記述します。たとえば "7/20/96" と記述します。
- "July 10 1995" という Long Date 形式で日付を指定する場合、年、月、日はどのような順序でも指定できます。年は 2 桁でも 4 桁でも指定できます。ただし、年を 2 桁で指定できるのは日付が 1970 年以降の場合だけです。
- かっこで囲まれた文字列はすべてコメントとして扱われます。かっこは入れ子でも記述できます。
- カンマおよびスペースは、両方とも区切り記号として扱われます。複数の区切り記号を使用できます。
- 月と曜日の名前は 2 文字以上で指定します。2 文字で指定した名前に複数の月または曜日が該当する場合は、その中の最後の名前が使用されます。たとえば、月に "Ju" と指定すると、June (6 月) ではなく July (7 月) となります。
- 指定した日付に合わない曜日を指定した場合、曜日は無視されます。たとえば、1996 年 11 月 9 日は実際には金曜日に当たるのに、"Tuesday November 9 1996" という文字列を指定した場合でも、メソッドは曜日だけを無視して正常に実行されます。その結果、**Date** オブジェクトの日付は、"Friday November 9 1996" となります。
- すべての標準時間帯、および UTC と GMT に対応しています。
- コロンは、時、分、秒の区切り記号として使用します。ただし、時、分、秒をすべて指定する必要はありません。"10:"、"10:11"、"10:11:12" という指定はすべて有効です。
- 24 時間形式で時刻を指定する場合は、12 時以降の時刻に "PM" を指定するとエラーが発生します。たとえば、"23:15 PM" と指定するとエラーが発生します。
- 無効な日付の入った文字列を指定するとエラーが発生します。たとえば、年が 2 つ指定されている、月が 2 つ指定されているなどの文字列を指定するとエラーが発生します。

使用例

parse メソッドの使用例を次に示します。関数に日付を渡すと、関数はその日付と 1970 年 1 月 1 日との差を返します。

```
function GetTimeTest(testdate){
   var s, t;                    //Declare variables.
   var MinMilli = 1000 * 60;       //Initialize variables.
   var HrMilli = MinMilli * 60;
   var DyMilli = HrMilli * 24;
   t = Date.parse(testdate);       //Parse testdate.
   s = "There are "                //Create return string.
   s += Math.round(Math.abs(t / DyMilli)) + " days "
```

```
    s += "between " + testdate + " and 1/1/70";
    return(s);                      //Return results.
}
```

必要条件

Version 1

対象

Date オブジェクト

参照

その他の技術情報

メソッド

parseFloat メソッド

文字列から変換された浮動小数点数を返します。

```
function parseFloat(numString : String) : Number
```

引数
numString

必ず指定します。浮動小数点数を表す文字列を指定します。

解説

parseFloat メソッドは、引数 *numString* に含まれる数値と等しい数値を返します。引数 *numString* の先頭に浮動小数点数がない場合は、**NaN** (not a number) を返します。

値が **NaN** かどうかを調べるには、**isNaN** メソッドを使用します。

使用例

次の例では、**parseFloat** メソッドを使用して 2 つの文字列を数値に変換しています。

```
      parseFloat("abc")      // Returns NaN.
parseFloat("1.2abc")   // Returns 1.2.
```

必要条件

Version 1

対象

Global オブジェクト

参照
関連項目
isNaN メソッド
parseInt メソッド
String オブジェクト

parseInt メソッド

文字列から変換された整数を返します。

```
function parseInt(numString : String [, radix : Number]) : Number
```

引数
numString

必ず指定します。数値に変換する文字列を指定します。

radix

省略可能です。引数 *numString* に含まれる数値の基数を表す 2 ～ 36 の範囲の値を指定します。省略した場合、先頭に "0x" が付いている文字列は 16 進数、"0" が付いている文字列は 8 進数と見なされます。これ以外の文字列は、10 進数と見なされます。

解説

parseInt メソッドは、引数 *numString* に含まれる数字と等しい整数を返します。引数 *numString* の先頭に整数がない場合は、**NaN** (not a number) を返します。

値が **NaN** かどうかを調べるには、**isNaN** メソッドを使用します。

使用例

次の例では、**parseInt** メソッドを使用して 2 つの文字列を数値に変換しています。

```
      parseInt("abc")     // Returns NaN.
parseInt("12abc")   // Returns 12.
```

必要条件

Version 1

対象

Global オブジェクト

参照
関連項目
isNaN メソッド
parseFloat メソッド
String オブジェクト
valueOf メソッド

pop メソッド

配列の最後の要素を削除し、削除した要素を返します。

```
function pop() : Object
```

解説

配列が空の場合は、**undefined** を返します。

必要条件

Version 5.5

対象

Array オブジェクト

参照

関連項目

push メソッド

pow メソッド

引数 *exponent* を指数とする引数 *base* の累乗を返します。

```
function pow(base : Number, exponent : Number) : Number
```

引数
base

必ず指定します。累乗の基数とする値を数式で指定します。

exponent

必ず指定します。累乗の指数とする値を数式で指定します。

解説

pow メソッドは、baseexponent に等しい数式を返します。

使用例

pow メソッドの使用例を次に示します。

```
var x = Math.pow(10,3); // x is assigned the value 1000.
```

必要条件

Version 1

対象

Math オブジェクト

参照
その他の技術情報
メソッド

push メソッド

配列に新しい要素を追加し、その要素を追加した後の配列の長さを返します。

```
function push([item1 : Object [, ... [, itemN : Object]]]) : Number
```

引数
item1, ..., itemN

省略可能です。**Array** の新しい要素を指定します。

解説

push メソッドでは、引数に指定された順に要素を追加します。引数の 1 つが配列である場合は、1 つの要素として追加します。複数の配列の要素を連結する場合は **concat** メソッドを使用します。

必要条件

Version 5.5

対象

Array オブジェクト

参照
関連項目
concat メソッド (Array)
pop メソッド

random メソッド

0 ～ 1 の範囲の擬似乱数を返します。

```
function random() : Number
```

解説

生成される擬似乱数は、0 ～ 1 の範囲内の値をとります (0 は含まれ、1 は含まれません)。つまり、戻り値が 0 になることはあっても 1 になることはありません。乱数ジェネレータのシードは、JScript が読み込まれた時点で自動的に生成されます。

必要条件

Version 1

対象：

Math オブジェクト

参照

その他の技術情報

メソッド

replace メソッド

正規表現または検索文字列を使って置換された文字列のコピーを返します。

```
function replace(rgExp : RegExp, replaceText : String) : String
```

引数

rgExp

必ず指定します。正規表現パターンおよび適用できるフラグを含む **Regular Expression** オブジェクトのインスタンスを指定します。**String** オブジェクトの名前またはリテラルを指定することもできます。*rgExp* が **Regular Expression** オブジェクトのインスタンスでない場合は文字列に変換され、完全に一致する文字列の検索が実行されます。文字列から正規表現への変換は行われません。

replaceText

必ず指定します。現在の文字列オブジェクトで、*rgExp* と一致した部分と置き換えるテキストを格納する **String** オブジェクトまたは文字列リテラルを指定します。JScript 5.5 以降では、置換するテキストを返す関数を指定することもできます。

解説

replace メソッドは、指定された置換を終えた後の現在の文字列オブジェクトのコピーを返します。

次の表にある変数を使用して、最後に検出した一致やその文字列を指定できます。一致変数は、テキストの置換処理において、対象を動的に指定する必要がある場合に使用します。

文字	説明
$$	**$** (JScript 5.5 以降)
$&	現在の文字列オブジェクト内で、パターンが完全に一致した部分を表します。(JScript 5.5 以降)
$`	現在の文字列オブジェクトの先頭から **$&** の直前までの部分を表します。(JScript 5.5 以降)
$'	現在の文字列オブジェクトの、**$&** 以降の部分を表します。(JScript 5.5 以降)
$ *n*	サブ文字列の中で *n* 番目に検出されたものを取得します。*n* には 1 桁の 10 進数値 (1 ～ 9) を指定します (JScript 5.5 以降)。
$ *nn*	サブ文字列の中で *nn* 番目に検出されたものを取得します。*nn* には 2 桁の 10 進数値 (01 ～ 99) を指定します (JScript 5.5 以降)。

replaceText が関数の場合、一致する各文字列に対してこの関数が *m* + 3 個の引数を伴って呼び出されます。*m* は、*rgExp* の中で使用されている取得を示す開きかっこの数です。1 個目の引数は一致したサブ文字列です。次の引数は、検索結果そのものです。*m* + 2 個目の引数は現在の文字列オブジェクトで一致したオフセットを表し、*m* + 3 個目の引数は現在の文字列オブジェクトを表します。結果は、該当する関数呼び出しの戻り値をそれぞれ一致する文字列と置換した文字列値で返されます。

replace メソッドを実行すると、グローバルなオブジェクト **RegExp** のプロパティが更新されます。

使用例

replace メソッドを使用して、単語 "The" という単語の最初のインスタンスを単語 "A" に置換する例を次に示します。使用している正規表現のパターンでは大文字と小文字が区別されるため、"The" の最初のインスタンスだけが置換されます。

```
function ReplaceDemo(){
   var r, re;                    //Declare variables.
   var ss = "The man hit the ball with the bat.\n";
   ss += "while the fielder caught the ball with the glove.";
   re = /The/g;                 //Create regular expression pattern.
   r = ss.replace(re, "A");     //Replace "The" with "A".
   return(r);                   //Return string with replacement made.
}
```

さらに、**replace** メソッドでは、パターンに一致する文字列どうしを置換できます。文字列の中の単語の各ペアを交換する例を次に示します。

```
function ReplaceDemo(){
   var r, re;                      //Declare variables.
   var ss = "The rain in Spain falls mainly in the plain.";
   re = /(\S+)(\s+)(\S+)/g;        //Create regular expression pattern.
   r = ss.replace(re, "$3$2$1");   //Swap each pair of words.
   return(r);                      //Return resulting string.
}
```

*replaceText*に関数を使用して華氏を摂氏に変換する例を次に示します。JScript 5.5 以降で使用できます。この関数は、数値の直後に "F" が続く文字列 (次の例では、"Water freezes at 32F and boils at 212F.") で機能します。

```
function f2c(s) {
  var test = /(\d+(\.\d*)?)F\b/g;    //Initialize pattern.
  return(s.replace
    (test,
      function($0,$1,$2) {
        return((($1-32) * 5/9) + "C");
      }
    )
  );
}
document.write(f2c("Water freezes at 32F and boils at 212F."));
```

必要条件

Version 1

対象

String オブジェクト

参照

関連項目

exec メソッド
match メソッド
RegExp オブジェクト
search メソッド
test メソッド

reverse メソッド

Array オブジェクトの要素を反転させます。

```
function reverse() : Array
```

解説

指定された **Array** オブジェクト内で要素の位置を反転させます。このメソッドを実行しても、新しい **Array** オブジェクトは作成されません。

配列内の要素が連続していない場合、このメソッドを実行すると新しい要素が作成され、配列内の連続していない要素の間に挿入されます。作成された要素の値は、undefined になります。

使用例

reverse メソッドの使用例を次に示します。

```
function ReverseDemo(){
   var a, l;                  //Declare variables.
   a = new Array(0,1,2,3,4);  //Create an array and populate it.
   l = a.reverse();           //Reverse the contents of the array.
   return(l);                 //Return the resulting array.
}
```

必要条件

Version 2

対象：

Array オブジェクト

参照

その他の技術情報

メソッド

round メソッド

指定された数式の値を整数に丸めて返します。

```
function round(number : Number) : Number
```

引数
number

必ず指定します。任意の数式を指定します。

解説

引数 *number* の小数部分が 0.5 以上の場合、この引数より大きい値となる最小の整数が戻り値として返されます。小数部分が 0.5 未満の場合は、この引数より小さいか等しい値となる最大の整数が戻り値として返されます。

必要条件

Version 1

対象：

Math オブジェクト

参照
その他の技術情報
メソッド

メソッド (S)

メソッドは、オブジェクトのメンバとなっている関数です。ここでは、名前が S で始まるメソッドを示します。

このセクションの内容

search メソッド

正規表現検索に一致する、最初の部分文字列の位置を返します。

setDate メソッド

Date オブジェクトに格納されている日の値を現地時間で設定します。

setFullYear メソッド

Date オブジェクトに格納されている年の値を現地時間で設定します。

setHours メソッド

Date オブジェクトの時刻の時の部分を現地時間で設定します。

setMilliseconds メソッド

Date オブジェクトの時刻のミリ秒の部分を現地時間で設定します。

setMinutes メソッド

Date オブジェクトに格納されている分の値を現地時間で設定します。

setMonth メソッド

Date オブジェクトの日付の月の部分を現地時間で設定します。

setSeconds メソッド

Date オブジェクトの時刻の秒の部分を現地時間で設定します。

setTime メソッド

Date オブジェクトの日付と時刻の値を設定します。

setUTCDate メソッド

Date オブジェクトの日付の日の部分を世界協定時刻 (UTC) で設定します。

setUTCFullYear メソッド

Date オブジェクトの日付の年の部分を世界協定時刻 (UTC) で設定します。

setUTCHours メソッド

Date オブジェクトの時刻の時の部分を世界協定時刻 (UTC) で設定します。

setUTCMilliseconds メソッド

Date オブジェクトの時刻のミリ秒の部分を世界協定時刻 (UTC) で設定します。

setUTCMinutes メソッド

Date オブジェクトの時刻の分の部分を世界協定時刻 (UTC) で設定します。

setUTCMonth メソッド

Date オブジェクトの日付の月の部分を世界協定時間 (UTC) で設定します。

setUTCSeconds メソッド

Date オブジェクトの時刻の秒の部分を世界協定時刻 (UTC) で設定します。

setYear メソッド

Date オブジェクトの日付の年の部分を設定します。

shift メソッド

配列の先頭の要素を削除し、削除した要素を返します。

sin メソッド

指定された数値のサインの値を返します。

slice メソッド (Array)

配列の一部を返します。

slice メソッド (String)

文字列の一部分を返します。

small メソッド

String オブジェクトに格納されている文字列を HTML の <SMALL> タグと </SMALL> タグで囲みます。

sort メソッド

要素の順序を並べ替えた **Array** オブジェクトを返します。

splice メソッド

配列から要素を削除し、必要に応じて新しい要素を削除位置に挿入し、削除した要素を返します。

split メソッド

文字列が複数の部分文字列に分割されたときの文字列の配列を返します。

sqrt メソッド

指定された数値の平方根を返します。

strike メソッド

String オブジェクトに格納されている文字列を HTML の <STRIKE> タグと </STRIKE> タグで囲みます。

sub メソッド

String オブジェクトに格納されている文字列を HTML の _{タグと} タグで囲みます。

substr メソッド

文字列内の、指定位置からの指定された長さを持つ部分を返します。

substring メソッド

String オブジェクト内の指定された位置にある部分文字列を返します。

sup メソッド

String オブジェクトに格納されている文字列を HTML の ^{タグと} タグで囲みます。

関連するセクション

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

メソッド

JScript で使用できるメソッドのアルファベット順の一覧と、メソッドの各カテゴリへのリンクを示します。

オブジェクト

JScript のオブジェクトの概念と、オブジェクトがプロパティとメソッドにどのように関連するかについて説明します。JScript がサポートするオブジェクトの詳細を説明するトピックへのリンクも示します。

search メソッド

正規表現検索に一致する、最初の部分文字列の位置を返します。

```
function search(rgExp : RegExp) : Number
```

引数
rgExp

必ず指定します。検索に使用する正規表現のパターンと適用できるフラグを格納した **Regular Expression** オブジェクトのインスタンスです。

解説

search メソッドは、パターンが見つかったかどうかを示します。一致する文字列が見つかった場合、**search** メソッドは、その文字列が先頭からどれだけ離れているかを示す整数値を返します。一致する文字列が見つからなかった場合は、-1 を返します。

使用例

search メソッドの使用例を次に示します。

```
function SearchDemo(){
   var r, re;                    //Declare variables.
   var s = "The rain in Spain falls mainly in the plain.";
   re = /falls/i;                //Create regular expression pattern.
   r = s.search(re);             //Search the string.
   return(r);                    //Return the index to the first match
                                 //or -1 if no match is found.
}
```

必要条件

Version 3

対象

String オブジェクト

参照
関連項目
exec メソッド
match メソッド
Regular Expression オブジェクト
replace メソッド
test メソッド
概念
正規表現の構文

setDate メソッド

Date オブジェクトの日付の日の部分を現地時間帯で設定します。

```
function setDate(numDate : Number)
```

引数
numDate

必ず指定します。設定する日を表す数値を指定します。

解説

日の値を世界協定時刻 (UTC) で設定するには、**setUTCDate** メソッドを使用します。

Date オブジェクトに格納されている月の日数より大きな値や負の値を引数 *numDate* に指定すると、引数 *numDate* に指定した値から格納されている月の日数を引いた値が日の値に設定されます。たとえば、1996 年 1 月 5 日の日付が格納されている Date オブジェクトに対して **setDate(32)** というメソッドを実行すると、日付は 1996 年 2 月 6 日に変更されます。負の値を指定した場合も同様の動作になります。

使用例

setDate メソッドの使用例を次に示します。

```
function SetDateDemo(newdate){
   var d, s;                  //Declare variables.
   d = new Date();            //Create date object.
   d.setDate(newdate);        //Set date to newdate.
   s = "Current setting is ";
   s += d.toLocaleString();
   return(s);                 //Return newly set date.
}
```

必要条件

Version 3

対象

Date オブジェクト

参照
関連項目
getDate メソッド
getUTCDate メソッド
setUTCDate メソッド

setFullYear メソッド

Date オブジェクトの日付の年の部分を現地時間で設定します。

```
function setFullYear(numYear : Number [, numMonth Number [, numDate Number]])
```

引数
numYear

必ず指定します。設定する年を表す数値を指定します。

numMonth

省略可能です。設定する月を表す数値を指定します。引数 *numDate* を指定する場合は、この引数を指定する必要があります。

numDate

省略可能です。設定する日を表す数値を指定します。

解説

省略可能な引数を指定せずに **set** で始まる名前の各メソッドを使用した場合、省略した設定の部分には対応する **get** で始まる名前のメソッドで返される値が設定されます。たとえば、省略可能な引数 *numMonth* を指定しなかった場合は、**getMonth** メソッドにより返される値が設定されます。

引数に有効範囲を超える値や負の値を指定すると、その値に応じて格納されている他の値が変更されます。

年の部分を世界協定時刻 (UTC) で設定するには、**setUTCFullYear** メソッドを使用してください。

Date オブジェクトでサポートされている年の範囲は、1970 年の前後の約 285,616 年です。

使用例

setFullYear メソッドの使用例を次に示します。

```
function SetFullYearDemo(newyear){
   var d, s;                  //Declare variables.
   d = new Date();            //Create Date object.
   d.setFullYear(newyear);    //Set year.
   s = "Current setting is ";
   s += d.toLocaleString();
   return(s);                 //Return new date setting.
}
```

必要条件

Version 3

対象

Date オブジェクト

参照
関連項目
getFullYear メソッド
getUTCFullYear メソッド
setUTCFullYear メソッド

setHours メソッド

Date オブジェクトの時刻の時の部分を現地時間で設定します。

```
function setHours(numHours : Number [, numMin : Number [, numSec : Number [, numMilli : Num
ber ]]])
```

引数
numHours

必ず指定します。設定する時を表す数値を指定します。

numMin

省略可能です。設定する分を表す数値を指定します。

numSec

省略可能です。設定する秒を表す数値を指定します。

numMilli

省略可能です。設定するミリ秒を表す数値を指定します。

解説

省略可能な引数を指定せずに、**set** で始まる名前の各メソッドを使用した場合、省略した設定の部分には対応する **get** で始まる名前のメソッドで返される値が設定されます。たとえば、省略可能な引数 *numMinutes* を指定しなかった場合は、**getMinutes** メソッドにより返される値が設定されます。

時の値を世界協定時刻 (UTC) で設定するには、**setUTCHours** メソッドを使用します。

引数に有効範囲を超える値や負の値を指定すると、値に応じて格納される他の値が変更されます。たとえば、日付が "Jan 5, 1996 00:00:00" と格納されている場合に **setHours(30)** メソッドを使用すると、日付は "Jan 6, 1996 06:00:00" に変更されます。負の値を指定した場合も、同様に処理されます。

使用例

setHours メソッドの使用例を次に示します。

```
function SetHoursDemo(nhr, nmin, nsec){
   var d, s;                     //Declare variables.
   d = new Date();               //Create Date object.
   d.setHours(nhr, nmin, nsec);  //Set hours, minutes, & seconds.
   s = "Current setting is " + d.toLocaleString()
   return(s);                    //Return new date setting.
}
```

必要条件

Version 3

対象

Date オブジェクト

参照
関連項目
getHours メソッド
getUTCHours メソッド
setUTCHours メソッド

setMilliseconds メソッド

Date オブジェクトの時刻のミリ秒の部分を現地時間で設定します。

```
function setMilliseconds(numMilli : Number)
```

引数
numMilli

必ず指定します。設定するミリ秒を表す数値を指定します。

解説

ミリ秒の値を世界協定時刻 (UTC) で設定するには、**setUTCMilliseconds** メソッドを使用してください。

引数 *numMilli* に 999 を超える値か負の値を指定すると、値に応じて格納されている秒 (場合によっては分、時) の値が変更されます。

使用例

setMilliseconds メソッドの使用例を次に示します。

```
function SetMSecDemo(nmsec){
   var d, s;                    //Declare variables.
   var sep = ":";               //Initialize separator.
   d = new Date();              //Create Date object.
   d.setMilliseconds(nmsec);    //Set milliseconds.
   s = "Current setting is ";
   s += d.toLocaleString() + sep + d.getMilliseconds();
   return(s);                   //Return new date setting.
}
```

必要条件

Version 3

対象：

Date オブジェクト

参照
関連項目
getMilliseconds メソッド
getUTCMilliseconds メソッド
setUTCMilliseconds メソッド

setMinutes メソッド

Date オブジェクトの時刻の分の部分を現地時間で設定します。

```
function setMinutes(numMinutes : Number [, numSeconds : Number [, numMilli : Number]])
```

引数
numMinutes

必ず指定します。設定する分を表す数値を指定します。

numSeconds

省略可能です。設定する秒を表す数値を指定します。引数 numMilli を指定する場合は、この引数を指定する必要があります。

numMilli

省略可能です。設定するミリ秒を表す数値を指定します。

解説

省略可能な引数を指定せずに、**set** で始まる名前の各メソッドを使用した場合、省略した設定の部分には対応する **get** で始まる名前のメソッドで返される値が設定されます。たとえば、省略可能な引数 *numSeconds* を指定しなかった場合は、**getSeconds** メソッドにより返される値が設定されます。

分の値を世界協定時刻 (UTC) で設定するには、**setUTCMinutes** メソッドを使用します。

引数に有効範囲を超える値や負の値を指定すると、値に応じて格納される他の値が変更されます。たとえば、日付が "Jan 5, 1996 00:00:00" と格納されている場合に **setMinutes(90)** メソッドを使用すると、日付は "Jan 5, 1996 01:30:00" に変更されます。負の値を指定した場合も、同様に処理されます。

使用例

setMinutes メソッドの使用例を次に示します。

```
function SetMinutesDemo(nmin, nsec){
   var d, s;                     //Declare variables.
   d = new Date();               //Create Date object.
   d.setMinutes(nmin, nsec);     //Set minutes.
   s = "Current setting is " + d.toLocaleString()
   return(s);                    //Return new setting.
}
```

必要条件

Version 1

対象

Date オブジェクト

参照
関連項目
getMinutes メソッド
getUTCMinutes メソッド
setUTCMinutes メソッド

setMonth メソッド

Date オブジェクトの日付の月の部分を現地時間で設定します。

```
function setMonth(numMonth : Number [, dateVal : Number])
```

引数
numMonth

必ず指定します。設定する月を表す数値を指定します。

dateVal

省略可能です。設定する日を表す数値を指定します。省略した場合は、**getDate** メソッドにより返される値が使用されます。

解説

月の値を世界協定時刻 (UTC) で設定するには、**setUTCMonth** メソッドを使用してください。

引数 *numMonth* に 11 を超える値 (1 月は 0) または負の値を指定すると、値に応じて格納されている年の値が変更されます。たとえば、日付が "Jan 5, 1996" と格納されている場合に **setMonth(14)** メソッドを使用すると、日付は "Mar 5, 1997" に変更されます。

使用例

setMonth メソッドの使用例を次に示します。

```
function SetMonthDemo(newmonth){
   var d, s;               //Declare variables.
   d = new Date();         //Create Date object.
   d.setMonth(newmonth);   //Set month.
   s = "Current setting is ";
   s += d.toLocaleString();
   return(s);              //Return new setting.
}
```

必要条件

Version 1

対象

Date オブジェクト

参照
関連項目
getMonth メソッド
getUTCMonth メソッド
setUTCMonth メソッド

setSeconds メソッド

Date オブジェクトの時刻の秒の部分を現地時間で設定します。

```
function setSeconds(numSeconds : Number [, numMilli : Number])
```

引数
numSeconds

必ず指定します。設定する秒を表す数値を指定します。

numMilli

省略可能です。設定するミリ秒を表す数値を指定します。

解説

省略可能な引数を指定せずに、**set** で始まる名前の各メソッドを使用した場合、省略した設定の部分には対応する **get** で始まる名前のメソッドで返される値が設定されます。たとえば、省略可能な引数 *numMilli* を指定しなかった場合は、**getMilliseconds** メソッドにより返される値が設定されます。

秒の値を世界協定時刻 (UTC) で設定するには、**setUTCSeconds** メソッドを使用してください。

引数に有効範囲を超える値や負の値を指定すると、値に応じて格納される他の値が変更されます。たとえば、日付が "Jan 5, 1996 00:00:00" と格納されている場合に **setSeconds(150)** メソッドを使用すると、日付は "Jan 5, 1996 00:02:30" に変更されます。

使用例

setSeconds メソッドの使用例を次に示します。

```
function SetSecondsDemo(nsec, nmsec){
   var d, s;                    //Declare variables.
   var sep = ":";
   d = new Date();              //Create Date object.
   d.setSeconds(nsec, nmsec);  //Set seconds and milliseconds.
   s = "Current setting is ";
   s += d.toLocaleString() + sep + d.getMilliseconds();
   return(s);                   //Return new setting.
}
```

必要条件

Version 1

対象

Date オブジェクト

参照
関連項目
getSeconds メソッド
getUTCSeconds メソッド
setUTCSeconds メソッド

setTime メソッド

Date オブジェクトの日付と時刻の値を設定します。

```
function setTime(milliseconds : Number)
```

引数
milliseconds

必ず指定します。グリニッジ標準時の 1970 年 1 月 1 日 0 時 0 分 0 秒から経過したミリ秒数単位の数値を指定します。

解説

引数 *milliseconds* に負の値を指定すると、1970 年以前の日付となります。有効な日付の範囲は、1970 年の前後の約 285,616 年です。

setTime メソッドを使用して日付と時刻を設定する場合は、指定する値はタイムゾーンに依存しません。

使用例

setTime メソッドの使用例を次に示します。

```
function SetTimeTest(newtime){
   var d, s;                  //Declare variables.
   d = new Date();            //Create Date object.
   d.setTime(newtime);        //Set time.
   s = "Current setting is ";
   s += d.toUTCString();
   return(s);                 //Return new setting.
}
```

必要条件

Version 1

対象

Date オブジェクト

参照
関連項目
getTime メソッド

setUTCDate メソッド

Date オブジェクトの日付の日の部分を世界協定時刻 (UTC) で設定します。

```
function setUTCDate(numDate : Number)
```

引数
numDate

必ず指定します。設定する日を表す数値を指定します。

解説

日の設定をローカル時間で行うには、**setDate** メソッドを使用してください。

Date オブジェクトに格納されている月の日数より大きな値や負の値を引数 *numDate* に指定すると、引数 *numDate* に指定した値から格納されている月の日数を引いた値が日の値に設定されます。たとえば、1996 年 1 月 5 日の日付が格納されていて **setUTCDate(32)** という呼び出しを行うと、日付は 1996 年 2 月 1 日になります。負の値を指定した場合も、同様に処理されます。

使用例

setUTCDate メソッドの使用例を次に示します。

```
function SetUTCDateDemo(newdate){
   var d, s;                    //Declare variables.
   d = new Date();              //Create Date object.
   d.setUTCDate(newdate);       //Set UTC date.
   s = "Current setting is ";
   s += d.toUTCString();
   return(s);                   //Return new setting.
}
```

必要条件

Version 3

対象

Date オブジェクト

参照
関連項目
getDate メソッド
getUTCDate メソッド
setDate メソッド

setUTCFullYear メソッド

Date オブジェクトの日付の年の部分を世界協定時刻 (UTC) で設定します。

```
function setUTCFullYear(numYear : Number [, numMonth : Number [, numDate : Number]])
```

引数
numYear

必ず指定します。設定する年を表す数値を指定します。

numMonth

省略可能です。設定する月を表す数値を指定します。

numDate

省略可能です。設定する日を表す数値を指定します。

解説

省略可能な引数を指定せずに、**set** で始まる名前の各メソッドを使用した場合、省略した設定の部分には対応する **get** で始まる名前のメソッドで返される値が設定されます。たとえば、省略可能な引数 *numMonth* を指定しなかった場合は、**getUTCMonth** メソッドにより返される値が設定されます。

引数に有効範囲を超える値や負の値を指定すると、値に応じて格納されている他の値が変更されます。

年の設定をローカル時間で行うには、**setFullYear** メソッドを使用してください。

Date オブジェクトでサポートされている年の範囲は、1970 年の前後の約 285,616 年です。

使用例

setUTCFullYear メソッドの使用例を次に示します。

```
function SetUTCFullYearDemo(newyear){
   var d, s;                    //Declare variables.
   d = new Date();              //Create Date object.
   d.setUTCFullYear(newyear);   //Set UTC full year.
   s = "Current setting is ";
   s += d.toUTCString();
   return(s);                   //Return new setting.
}
```

必要条件

Version 3

対象

Date オブジェクト

参照
関連項目
getFullYear メソッド
getUTCFullYear メソッド
setFullYear メソッド

setUTCHours メソッド

Date オブジェクトの時刻の時の部分を世界協定時刻 (UTC) で設定します。

```
function setUTCHours(numHours : Number [, numMin : Number [, numSec : Number [, numMilli :
Number]]])
```

引数
numHours

必ず指定します。設定する時を表す数値を指定します。

numMin

省略可能です。設定する分を表す数値を指定します。

numSec

省略可能です。設定する秒を表す数値を指定します。

numMilli

省略可能です。設定するミリ秒を表す数値を指定します。

解説

省略可能な引数を指定せずに、**set** で始まる名前の各メソッドを使用した場合、省略した設定の部分には対応する **get** で始まる名前のメソッドで返される値が設定されます。たとえば、省略可能な引数 *numMin* を指定しなかった場合は、**getUTCMinutes** メソッドにより返される値が設定されます。

時の設定をローカル時間で行うには、**setHours** メソッドを使用してください。

引数に有効範囲を超える値や負の値を指定すると、値に応じて格納される他の値が変更されます。たとえば、格納されている日付が "Jan 5, 1996 00:00:00.00" の場合に **setUTCHours(30)** メソッドを使用すると、日付は "Jan 6, 1996 06:00:00.00" に変更されます。

使用例

setUTCHours メソッドの使用例を次に示します。

```
function SetUTCHoursDemo(nhr, nmin, nsec){
   var d, s;                        //Declare variables.
   d = new Date();                  //Create Date object.
   d.setUTCHours(nhr, nmin, nsec);  //Set UTC hours, minutes, seconds.
   s = "Current setting is " + d.toUTCString()
   return(s);                       //Return new setting.
}
```

必要条件

Version 3

対象

Date オブジェクト

参照
関連項目
getHours メソッド
getUTCHours メソッド
setHours メソッド

setUTCMilliseconds メソッド

Date オブジェクトの時刻のミリ秒の部分を世界協定時刻 (UTC) で設定します。

```
function setUTCMilliseconds(numMilli : Number)
```

引数
numMilli

必ず指定します。設定するミリ秒を表す数値を指定します。

解説

ミリ秒の設定をローカル時間で行うには、**setMilliseconds** メソッドを使用してください。

引数 *numMilli* に 999 を超える値か負の値を指定すると、値に応じて格納されている秒 (場合によっては分、時) の値が変更されます。

使用例

setUTCMilliseconds メソッドの使用例を次に示します。

```
function SetUTCMSecDemo(nmsec){
   var d, s;                          //Declare variables.
   var sep = ":";                     //Initialize separator.
   d = new Date();                    //Create Date object.
   d.setUTCMilliseconds(nmsec);       //Set UTC milliseconds.
   s = "Current setting is ";
   s += d.toUTCString() + sep + d.getUTCMilliseconds();
   return(s);                         //Return new setting.
}
```

必要条件

Version 3

対象

Date オブジェクト

参照
関連項目
getMilliseconds メソッド
getUTCMilliseconds メソッド
setMilliseconds メソッド

setUTCMinutes メソッド

Date オブジェクトの時刻の分の部分を世界協定時刻 (UTC) で設定します。

```
function setUTCMinutes(numMinutes : Number [, numSeconds : Number [, numMilli : Number ]])
```

引数
numMinutes

必ず指定します。設定する分を表す数値を指定します。

numSeconds

省略可能です。設定する秒を表す数値を指定します。引数 *numMilli* を指定する場合は、この引数を指定する必要があります。

numMilli

省略可能です。設定するミリ秒を表す数値を指定します。

解説

省略可能な引数を指定せずに、**set** で始まる名前の各メソッドを使用した場合、省略した設定の部分には対応する **get** で始まる名前のメソッドで返される値が設定されます。たとえば、省略可能な引数 *numSeconds* を指定しなかった場合は、**getUTCSeconds** メソッドにより返される値が設定されます。

分の設定をローカル時間で行うには、**setMinutes** メソッドを使用してください。

引数に有効範囲を超える値や負の値を指定すると、値に応じて格納される他の値が変更されます。たとえば、格納されている日付が "Jan 5, 1996 00:00:00.00" の場合に、**setUTCMinutes(70)** メソッドを使用すると、日付は "Jan 5, 1996 01:10:00.00" に変更されます。

使用例

setUTCMinutes メソッドの使用例を次に示します。

```
function SetUTCMinutesDemo(nmin, nsec){
   var d, s;                    //Declare variables.
   d = new Date();              //Create Date object.
   d.setUTCMinutes(nmin,nsec);  //Set UTC minutes.
   s = "Current setting is " + d.toUTCString()
   return(s);                   //Return new setting.
}
```

必要条件

Version 3

対象

Date オブジェクト

参照
関連項目
getMinutes メソッド
getUTCMinutes メソッド
setMinutes メソッド

setUTCMonth メソッド

Date オブジェクトの日付の月の部分を世界協定時間 (UTC) で設定します。

```
function setUTCMonth(numMonth : Number [, dateVal : Number ])
```

引数
numMonth

必ず指定します。設定する月を表す数値を指定します。

dateVal

省略可能です。設定する日を表す数値を指定します。省略した場合は、**getUTCDate** メソッドにより返される値が使用されます。

解説

月の設定をローカル時間で行うには、**setMonth** メソッドを使用してください。

引数 *numMonth* に 11 を超える値 (1 月は 0) や負の値を指定すると、値に応じて格納されている年の値が変更されます。たとえば、格納されている日付が "Jan 5, 1996 00:00:00.00" の場合に **setUTCMonth(14)** メソッドを使用すると、日付は "Mar 5, 1997 00:00:00.00" に変更されます。

使用例

setUTCMonth メソッドの使用例を次に示します。

```
function SetUTCMonthDemo(newmonth){
   var d, s;                        //Declare variables.
   d = new Date();                  //Create Date object.
   d.setUTCMonth(newmonth);         //Set UTC month.
   s = "Current setting is ";
   s += d.toUTCString();
   return(s);                       //Return new setting.
}
```

必要条件

Version 3

対象

Date オブジェクト

参照
関連項目
getMonth メソッド
getUTCMonth メソッド
setMonth メソッド

setUTCSeconds メソッド

Date オブジェクトの時刻の秒の部分を世界協定時刻 (UTC) で設定します。

```
function setUTCSeconds(numSeconds : Number [, numMilli : Number ])
```

引数
numSeconds

必ず指定します。設定する秒を表す数値を指定します。

numMilli

省略可能です。設定するミリ秒を表す数値を指定します。

解説

省略可能な引数を指定せずに、**set** で始まる名前の各メソッドを使用した場合、省略した設定の部分には対応する **get** で始まる名前のメソッドで返される値が設定されます。たとえば、省略可能な引数 *numMilli* を指定しなかった場合は、**getUTCMilliseconds** メソッドにより返される値が設定されます。

秒の設定をローカル時間で行うには、**setSeconds** メソッドを使用してください。

引数に有効範囲を超える値や負の値を指定すると、値に応じて格納される他の値が変更されます。たとえば、格納されている日付が "Jan 5, 1996 00:00:00.00" の場合に **setSeconds(150)** メソッドを使用すると、日付は "Jan 5, 1996 00:02:30.00" に変更されます。

使用例

setSeconds メソッドの使用例を次に示します。

```
function SetUTCSecondsDemo(nsec, nmsec){
   var d, s;                        //Declare variables.
   d = new Date();                  //Create Date object.
   d.setUTCSeconds(nsec, nmsec);    //Set UTC seconds and milliseconds.
   s = "Current UTC milliseconds setting is ";
   s += d.getUTCMilliseconds();     //Get new setting.
   return(s);                       //Return new setting.
}
```

必要条件

Version 3

対象

Date オブジェクト

参照
関連項目
getSeconds メソッド
getUTCSeconds メソッド
setSeconds メソッド

setYear メソッド

Date オブジェクトの日付の年の部分を設定します。

```
function setYear(numYear : Number)
```

引数
numYear

必ず指定します。設定する年から 1900 を引いた数値を指定します。

解説

このメソッドは廃止されました。現在、以前のバージョンとの互換性のためだけに残されています。代わりに、**setFullYear** メソッドの使用をお勧めします。

Date オブジェクトに 1997 年を設定するには、**setYear(97)** メソッドを実行します。また、2010 年を設定するには、**setYear(2010)** メソッドを実行します。0 ～ 99 の範囲で年を設定するには、**setFullYear** メソッドを使用します。

メモ：

JScript Version 1.0 では、**setYear** メソッドは、年を表す値にかかわらず *numYear* に指定した値に 1900 を加算した結果を使用します。たとえば、1899 年に設定するときは *numYear* に -1 を指定し、2000 年に設定するときは *numYear* に 100 を指定します。

必要条件

Version 1

対象

Date オブジェクト

参照
関連項目
getFullYear メソッド
getUTCFullYear メソッド
getYear メソッド
setFullYear メソッド
setUTCFullYear メソッド

JScript 8.0

shift メソッド

配列の先頭の要素を削除し、削除した要素を返します。

```
function shift() : Object
```

解説

shift メソッドは、配列の先頭の要素を削除し、削除した要素を返します。

必要条件

Version 5.5

対象：

Array オブジェクト

参照

関連項目

unshift メソッド

sin メソッド

指定された数値のサインの値を返します。

```
function sin(number : Number) : Number
```

引数
number

必ず指定します。サインを求める数式を指定します。

解説

戻り値は、引数 *number* のサインの値です。

必要条件

Version 1

対象：

Math オブジェクト

参照
関連項目
acos メソッド
asin メソッド
atan メソッド
cos メソッド
tan メソッド

slice メソッド (Array)

配列の一部を返します。

```
function slice(start : Number [, end : Number]) : Array
```

引数
start

必ず指定します。配列から抽出する部分の先頭位置をインデックス番号で指定します。

end

省略可能です。配列から抽出する部分の終端位置をインデックス番号で指定します。

解説

slice メソッドは、配列内の指定した部分が格納された **Array** オブジェクトを返します。

slice メソッドは、引数 *end* で指定した要素の 1 つ前の要素までをコピーします。引数 *start* に負の値を指定した場合、*length* + *start* として処理されます。*length* は配列の長さです。引数 *end* に負の値を指定した場合、*length* + *end* として処理されます。*length* は配列の長さです。引数 *end* を省略した場合は、配列の最後までが抽出されます。*end* で指定した抽出終了位置が *start* で指定した抽出開始位置より前に来る場合、要素は新しい配列にコピーされません。

使用例

次のコードは、*myArray* の最後の要素を除くすべての要素を *newArray* オブジェクトにコピーする例です。

```
var myArray = new Array(4,3,5,65);
var newArray = myArray.slice(0, -1)
```

必要条件

Version 3

対象

Array オブジェクト

参照
関連項目
slice メソッド (String)
String オブジェクト

slice メソッド (String)

文字列の一部分を返します。

```
function slice(start : Number [, end : Number]) : String
```

引数
start

必ず指定します。文字列から抽出する部分の先頭位置をインデックス番号で指定します。

end

省略可能です。文字列から抽出する部分の終端位置をインデックス番号で指定します。

解説

slice メソッドは、文字列内の指定した部分が格納された **String** オブジェクトを返します。

slice メソッドは、引数 *end* で指定した要素の 1 つ前の要素までをコピーします。引数 *start* に負の値を指定した場合、*length* + *start* として処理されます。*length* は文字列の長さです。引数 *end* に負の値を指定した場合、*length* + *end* として処理されます。*length* は文字列の長さです。引数 *end* を省略した場合は、文字列の最後までが抽出されます。*end* で指定した抽出終了位置が *start* で指定した抽出開始位置より前に来る場合、文字は新しい文字列にコピーされません。

使用例

以下のコード例では、**slice** メソッドの最初の呼び出しで、`str` の最初の 5 文字を含む文字列が返されます。**slice** メソッドの 2 番目の呼び出しでは、`str` の最後の 5 文字を含む文字列が返されます。

```
var str = "hello world";
var firstfive = str.slice(0,5); // Contains "hello".
var lastfive = str.slice(-5);   // Contains "world".
```

必要条件

Version 3

対象

String オブジェクト

参照
関連項目
Array オブジェクト
slice メソッド (Array)

small メソッド

String オブジェクトに格納されているテキストを HTML の <SMALL> タグと </SMALL> タグで囲み、文字列として返します。

```
function small() : String
```

解説

このメソッドの実行時には、タグが既に元の文字列内にあるかどうかの確認は行われません。

使用例

small メソッドの使用例を次に示します。

```
var strVariable = "This is a string";
strVariable = strVariable.small();
```

2 つ目のステートメントの実行後、変数 `strVariable` は次の値になります。

```
<SMALL>This is a string</SMALL>
```

必要条件

Version 1

対象：

String オブジェクト

参照

関連項目

big メソッド

sort メソッド

要素の順序を並べ替えた **Array** オブジェクトを返します。

```
function sort(sortFunction : Function ) : Array
```

引数
sortFunction

省略可能です。要素の順序を決定するために使用する関数の名前を指定します。

解説

sort メソッドは、指定された **Array** オブジェクト内の要素を並べ替えます。このメソッドを実行しても、新しい **Array** オブジェクトは作成されません。

引数 *sortFunction* を指定する場合は、次の戻り値を返すような関数を指定する必要があります。

- 1 つ目の引数が 2 つ目の引数よりも小さい場合は、負の値を返す関数。
- 2 つの引数が等しい場合は 0 を返す関数。
- 1 つ目の引数が 2 つ目の引数よりも大きい場合は、正の値を返す関数。

引数 *sortFunction* を省略すると、要素は ASCII コードの昇順で並べ替えられます。

使用例

sort メソッドの使用例を次に示します。

```
function SortDemo(){
   var a, l;                        //Declare variables.
   a = new Array("X" ,"y" ,"d", "Z", "v","m","r");
   l = a.sort();                    //Sort the array.
   return(l);                       //Return sorted array.
}
```

必要条件

Version 2

対象

Array オブジェクト

参照
その他の技術情報
オブジェクト (JScript)

splice メソッド

配列から要素を削除し、必要に応じて新しい要素を削除位置に挿入し、削除した要素を返します。配列から削除された要素を返します。

```
function splice(start : Number, deleteCount : Number [, item1 : Object [, ... [, itemN : Ob
ject]]]]) : Array
```

引数
start

必ず指定します。要素を削除する配列の削除開始位置を 0 から始まる番号で指定します。

deleteCount

必ず指定します。削除する要素の数を指定します。

item1, ..., itemN

省略可能です。削除した要素の代わりに挿入する要素を指定します。

解説

splice メソッドは、指定された数の要素を *start* の位置から削除し、新しい要素を挿入して配列を修正します。削除した要素は、新しい **array** オブジェクトに格納して返します。

必要条件

Version 5.5

対象

Array オブジェクト

参照
関連項目
slice メソッド (Array)

split メソッド

文字列が複数の部分文字列に分割されたときの文字列の配列を返します。

```
function split([ separator : { String | RegExp } [, limit : Number]]) : Array
```

引数

separator

省略可能です。文字列を区切る 1 つ以上の文字を表す、**Regular Expression** オブジェクトの文字列またはインスタンスを指定します。省略した場合、文字列全体を含む単一要素の配列が返されます。

limit

省略可能です。配列に返される要素の数を制限する値を指定します。

解説

split メソッドの結果は、文字列を *separator* の位置で分割してできた文字列の配列です。*separator* は、配列要素の一部としては返されません。

使用例

split メソッドの使用例を次に示します。

```
function SplitDemo(){
   var s, ss;
   var s = "The rain in Spain falls mainly in the plain.";
   // Split at each space character.
   ss = s.split(" ");
   return(ss);
}
```

必要条件

Version 3

対象

String オブジェクト

参照

関連項目

concat メソッド (String)
RegExp オブジェクト
Regular Expression オブジェクト

概念

正規表現の構文

sqrt メソッド

指定された数値の平方根を返します。

```
function sqrt(number : Number) : Number
```

引数
number

必ず指定します。平方根を求める数式を指定します。

解説

引数 *number* に負の値を指定すると、戻り値は **NaN** になります。

必要条件

Version 1

対象：

Math オブジェクト

参照
関連項目
SQRT1_2 プロパティ
SQRT2 プロパティ

strike メソッド

String オブジェクトに格納されているテキストを HTML の <STRIKE> タグと </STRIKE> タグで囲み、文字列として返します。

```
function strike() : String
```

解説

このメソッドの実行時には、タグが既に元の文字列内にあるかどうかの確認は行われません。

使用例

strike メソッドの使用例を次に示します。

```
var strVariable = "This is a string object";
strVariable = strVariable.strike();
```

2 つ目のステートメントの実行後、変数 `strVariable` は次の値になります。

```
<STRIKE>This is a string object</STRIKE>
```

必要条件

Version 1

対象：

String オブジェクト

参照

その他の技術情報

メソッド

sub メソッド

String オブジェクトに格納されているテキストを HTML の _{タグと} タグで囲み、文字列として返します。

```
function sub() : String
```

解説

このメソッドの実行時には、タグが既に元の文字列内にあるかどうかの確認は行われません。

使用例

sub メソッドの使用例を次に示します。

```
var strVariable = "This is a string object";
strVariable = strVariable.sub();
```

2 つ目のステートメントの実行後、変数 `strVariable` は次の値になります。

```
<SUB>This is a string object</SUB>
```

必要条件

Version 1

対象：

String オブジェクト

参照

関連項目

sup メソッド

substr メソッド

文字列内の、指定位置からの指定された長さを持つ部分を返します。

```
function substr(start : Number [, length : Number]) : String
```

引数
start

必ず指定します。テキストの取り出し開始位置を指定します。文字列の 1 文字目のインデックス番号は 0 です。

length

省略可能です。取り出す部分の文字数を指定します。

解説

引数 *length* に 0 または負の値を指定すると、長さ 0 の文字列 ("") が返されます。この引数を省略すると、文字列の最後まで取り出されます。

使用例

substr メソッドの使用例を次に示します。

```
function SubstrDemo(){
   var s, ss;                //Declare variables.
   var s = "The rain in Spain falls mainly in the plain.";
   ss = s.substr(12, 5);  //Get substring.
   return(ss);               // Returns "Spain".
}
```

必要条件

Version 3

対象

String オブジェクト

参照
関連項目
substring メソッド

substring メソッド

String オブジェクトに格納されている文字列内の指定された位置にある文字列を返します

```
function substring(start : Number, end : Number) : String
```

引数
start

必ず指定します。必ず指定します。取得する文字列の先頭文字の位置を 0 から始まる値で指定します

end

必ず指定します。必ず指定します。取得する文字列の終了文字の位置を 0 から始まる値で指定します。

解説

substring メソッドの戻り値は、*start* から *end* の前の文字までを含む文字列です。

substring メソッドでは、引数 *start* と引数 *end* のうち値の小さい方が取得する文字列の先頭位置になります。たとえば *strvar*.**substring(**0, 3**)** メソッドと *strvar*.**substring(**3, 0**)** メソッドは同じ文字列を返します。

引数 *start* または *end* のいずれかが **NaN** または負である場合は、0 に置き換えられます。

取得した文字列の長さは、2 つの引数の差の絶対値になります。たとえば、*strvar*.**substring(**0, 3**)** メソッドと *strvar*.**substring(**3, 0**)** メソッドで返される文字列の長さは 3 になります。

使用例

substring メソッドの使用例を次に示します。

```
function SubstringDemo(){
   var ss;                         //Declare variables.
   var s = "The rain in Spain falls mainly in the plain..";
   ss = s.substring(12, 17);   //Get substring.
   return(ss);                     //Return substring.
}
```

必要条件

Version 1

対象：

String オブジェクト

参照
関連項目
substr メソッド

sup メソッド

String オブジェクトに格納されているテキストを HTML の ^{タグと} タグで囲み、文字列として返します。

```
function sup() : String
```

解説

このメソッドの実行時には、タグが既に元の文字列内にあるかどうかの確認は行われません。

使用例

sup メソッドの使用例を次に示します。

```
var strVariable = "This is a string object";
strVariable = strVariable.sup();
```

2 つ目のステートメントの実行後、変数 `strVariable` は次の値になります。

```
<SUP>This is a string object</SUP>
```

必要条件

Version 1

対象：

String オブジェクト

参照

関連項目

sub メソッド

メソッド (T ～ Z)

メソッドは、オブジェクトのメンバとなっている関数です。ここでは、名前が T ～ Z で始まるメソッドを示します。

このセクションの内容

tan メソッド

指定された数値のタンジェントの値を返します。

test メソッド

文字列内で検索パターンに一致する部分が存在するかどうかを調べ、結果を示すブール値を返します。

toArray メソッド

VBArray を JScript の標準配列に変換します。

toDateString メソッド

日付を文字列値として返します。

toExponential メソッド

数値を指数表記の文字列として返します。

toFixed メソッド

数値を固定小数点表記の文字列として返します。

toGMTString メソッド

グリニッジ標準時 (GMT) を使用して日付データを文字列に変換します。

toLocaleDateString メソッド

ホスト環境の現在のロケールに対応した日付を文字列値で返します。

toLocaleLowerCase メソッド

ホスト環境の現在のロケールに従って、すべての英字を小文字に変換した文字列を返します。

toLocaleString メソッド

現在のロケールでの既定の書式を使用して、日付データを文字列に変換します。

toLocaleTimeString メソッド

ホスト環境の現在のロケールに対応した時刻を文字列値で返します。

toLocaleUpperCase メソッド

ホスト環境の現在のロケールに従って、すべての英字を大文字に変換した文字列を返します。

toLowerCase メソッド

すべての英字を小文字に変換した文字列を返します。

toPrecision メソッド

指定された桁数の指数表記または固定小数点表記の数値を文字列で返します。

toString メソッド

オブジェクトの値を表す文字列を返します。

toTimeString メソッド

時刻を文字列値として返します。

toUpperCase メソッド

すべての英字を大文字に変換した文字列を返します。

toUTCString メソッド

日付を世界協定時刻 (UTC) の文字列に変換します。

ubound メソッド

VBArray の指定された次元での最も大きいインデックス値を返します。

unescape メソッド

escape メソッドを使ってエスケープコード付きの文字にエンコードした **String** オブジェクトの文字列を元の形にデコードします。

unshift メソッド

指定された要素を配列の先頭に挿入し、その配列を返します。

UTC メソッド

世界協定時刻 (UTC) (つまり GMT) の 1970 年 1 月 1 日 0 時 0 分 0 秒と、指定した日付との間をミリ秒単位で返します。

valueOf メソッド

指定されたオブジェクトのプリミティブ値を返します。

関連するセクション

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

メソッド

JScript で使用できるメソッドのアルファベット順の一覧と、メソッドの各カテゴリへのリンクを示します。

オブジェクト

JScript のオブジェクトの概念と、オブジェクトがプロパティとメソッドにどのように関連するかについて説明します。JScript がサポートするオブジェクトの詳細を説明するトピックへのリンクも示します。

tan メソッド

指定された数値のタンジェントの値を返します。

```
function tan(number : Number) : Number
```

引数
number

必ず指定します。タンジェントを求める数式を指定します。

解説

戻り値は、引数 *number* のタンジェントの値です。

必要条件

Version 1

対象：

Math オブジェクト

参照
関連項目
acos メソッド
asin メソッド
atan メソッド
atan2 メソッド
cos メソッド
sin メソッド

test メソッド

文字列内に正規表現パターンに一致する部分が存在するかどうかを示すブール値を返します。

```
function test(str : String) : Boolean
```

引数
str

必ず指定します。検索対象となる文字列を指定します。

解説

test メソッドは、文字列内にパターンに一致する部分が存在するかどうかを調べ、存在する場合は真 (**true**) を返し、存在しない場合は偽 (**false**) を返します。一致する文字列が見つかった場合は、グローバルな **RegExp** オブジェクトのプロパティが検索結果を反映して更新されます。

正規表現でグローバル フラグが設定されている場合は、**lastIndex** の値で指定された位置から文字列の検索が開始されます。グローバル フラグが設定されていない場合は、**lastIndex** の値に関係なく、検索は文字列の先頭から開始されます。

使用例

test メソッドの使用例を次に示します。この例を使用するには、正規表現パターンおよび文字列に関数を渡します。関数は、文字列内に正規表現パターンに一致する部分が存在するかどうかを調べ、検索結果を示す文字列を返します。

```
function TestDemo(re, s){
   var s1;                          //Declare variable.
   // Test string for existence of regular expression.
   if (re.test(s))                  //Test for existence.
      s1 = " contains ";            //s contains pattern.
   else
      s1 = " does not contain ";    //s does not contain pattern.
   return("'" + s + "'" + s1 + "'"+ re.source + "'"); //Return string.
}
```

必要条件

Version 3

対象：

Regular Expression オブジェクト

参照
関連項目
RegExp オブジェクト
概念
正規表現の構文

toArray メソッド

VBArray を JScript の標準配列に変換します。

```
function toArray() : Array
```

解説

この変換では、多次元の VBArray も 1 次元の JScript 配列に変換されます。**toArray** メソッドは各次元を順に追加します。たとえば、各次元に要素を 3 つずつ持つ次のような 3 次元 VBArray があったとすると、次のように JScript 配列に変換されます。

(1, 2, 3), (4, 5, 6), (7, 8, 9) という VBArray は、1, 2, 3, 4, 5, 6, 7, 8, 9 という JScript 配列に変換されます。

現在、JScript 配列を VBArray に変換する方法はありません。

使用例

次のコードは、3 つの部分から構成されます。最初の部分は、Visual Basic のセーフ配列を作成する VBScript のコードです。2 番目の部分は、Visual Basic のセーフ配列を JScript の配列に変換する JScript のコードです。どちらのコードも、HTML ページの <HEAD> セクションに記述します。3 番目の部分は、他の 2 つのコードを実行するために <BODY> セクションに記述する JScript のコードです。

```
<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Function CreateVBArray()
   Dim i, j, k
   Dim a(2, 2)
   k = 1
   For i = 0 To 2
      For j = 0 To 2
         a(j, i) = k
         document.writeln(k)
         k = k + 1
      Next
      document.writeln("<BR>")
   Next
   CreateVBArray = a
End Function
-->
</SCRIPT>

<SCRIPT LANGUAGE="JScript">
<!--
function VBArrayTest(vbarray)
{
   var a = new VBArray(vbarray);
   var b = a.toArray();
   var i;
   for (i = 0; i < 9; i++)
   {
      document.writeln(b[i]);
   }
}
-->
</SCRIPT>
</HEAD>

<BODY>
<SCRIPT LANGUAGE="JScript">
<!--
   VBArrayTest(CreateVBArray());
-->
</SCRIPT>
</BODY>
```

必要条件

Version 3

対象：

VBArray オブジェクト

参照

関連項目

dimensions メソッド
getItem メソッド
lbound メソッド
ubound メソッド

toDateString メソッド

日付を文字列値として返します。

```
function toDateString() : String
```

解説

toDateString メソッドは、現在のタイムゾーンに従って、日付を便利で読みやすい形式の文字列値にして返します。

必要条件

Version 5.5

対象：

Date オブジェクト

参照

関連項目

toTimeString メソッド
toLocaleDateString メソッド

toExponential メソッド

数値を指数表記の文字列として返します。

```
function toExponential( [fractionDigits : Number] ) : String
```

引数
fractionDigits

省略可能です。小数点以下の桁数を指定します。0 ～ 20 の範囲で指定します。

解説

toExponential メソッドは、数値を指数表記で表した文字列を返します。文字列には、1 桁の整数、小数点、および *fractionDigits* で指定した桁数の小数が格納されます。

fractionDigits を指定しない場合は、**toExponential** メソッドは、数値を一意に表すのに必要な桁数だけ返します。

必要条件

Version 5.5

対象

Number オブジェクト

参照
関連項目
toFixed メソッド
toPrecision メソッド

toFixed メソッド

数値を固定小数点表記の文字列として返します。

```
function toFixed( [fractionDigits : Number] ) : String
```

引数
fractionDigits

省略可能です。小数点以下の桁数を指定します。0 ～ 20 の範囲で指定します。

解説

toFixed メソッドは、数値を固定小数点表記で表した文字列を返します。文字列には、1 桁の整数、小数点、および *fractionDigits* で指定した桁数の小数部が格納されます。

fractionDigits を指定していない場合、または **undefined** である場合は、**toFixed** メソッドは値を 0 と見なします。

必要条件

Version 5.5

対象

Number オブジェクト

参照
関連項目
toExponential メソッド
toPrecision メソッド

toGMTString メソッド

グリニッジ標準時 (GMT) を使用して日付データを文字列に変換します。

```
function toGMTString() : String
```

解説

toGMTString メソッドは廃止されました。現在、以前のバージョンとの互換性のためだけに残されています。代わりに、**toUTCString** メソッドの使用をお勧めします。

toGMTString メソッドは、日付を GMT 規約に基づいた表記に変換し、**String** オブジェクトに格納して返します。戻り値は、"05 Jan 1996 00:00:00 GMT" などの形式になります。

必要条件

Version 1

対象：

Date オブジェクト

参照

関連項目

toUTCString メソッド

toLocaleDateString メソッド

ホスト環境の現在のロケールに対応した日付を文字列値で返します。

```
function toLocaleDateString() : String
```

解説

toLocaleDateString メソッドは、現在のタイムゾーンに従って、日付を読みやすい形式の文字列値にして返します。日付の形式は、ホスト環境の現在のロケールの既定形式になります。このメソッドの戻り値はコンピュータ間で一致しないため、スクリプト用には適していません。**toLocalDateString** メソッドは、表示をフォーマットするためだけに使用し、計算には使用しないようにしてください。

必要条件

Version 5.5

対象：

Date オブジェクト

参照

関連項目

toDateString メソッド
toLocaleTimeString メソッド

toLocaleLowerCase メソッド

ホスト環境の現在のロケールに従って、すべての英字を小文字に変換した文字列を返します。

```
function tolocaleLowerCase() : String
```

解説

toLocaleLowerCase メソッドは、ホスト環境の現在のロケールに従って、文字列の文字を変換します。ほとんどの場合、**toLowerCase** メソッドの実行結果と同じになります。結果が異なるのは、言語規則が通常の Unicode の大文字、小文字のマッピングと競合する場合です。

必要条件

Version 5.5

対象：

String オブジェクト

参照

関連項目

toLocaleUpperCase メソッド
toLowerCase メソッド

toLocaleString メソッド

ホスト環境の現在のロケールに対応した値を文字列値で返します。

```
function toLocaleString() : String
```

解説

Array オブジェクトの場合、配列要素を文字列に変換し、各文字列をホスト環境の現在のロケールで指定された区切り文字で連結して返します。

Date オブジェクトの場合、**toLocaleString** メソッドは、現在のロケールの長い既定形式に日付データを変換し、**String** オブジェクトに格納して返します。

- 西暦 1601 から 9999 年の場合、日付はユーザーのコントロール パネルの地域設定に基づいた形式になります。
- この範囲以外の日付の場合、**toString** メソッドの既定の形式が使用されます。

Number オブジェクトの場合、**toLocaleString** メソッドは、ホスト環境の現在のロケールに対応する形式の **Number** オブジェクトの値を表す文字列値を生成します。

Object オブジェクトの場合、**toLocaleString** メソッドは、オブジェクトが使用するかどうかにかかわらず、すべてのオブジェクトに汎用的な **toLocaleString** 機能を提供するために用意されています。

メモ :

toLocaleString メソッドは、ユーザーに結果を表示する目的だけで使用してください。この関数の結果はコンピュータによって異なるため、スクリプト内での処理の基準としては使用しないでください。

使用例

Array オブジェクト、**Date** オブジェクト、および **Number** オブジェクトに対して **toLocaleString** メソッドを実行するクライアント側コード例を次に示します。

```
function toLocaleStringArray() {
   // Declare variables.
   var myArray = new Array(6);
   var i;
   // Initialize string.
   var s = "The array contains: ";
   // Populate array with values.
   for(i = 0;i < 7; i++)
   {
      // Make value same as index.
      myArray[i] = i;
   }
   s += myArray.toLocaleString();
   return(s);
}
function toLocaleStringDate() {
   // Declare variables.
   var d = new Date();
   var s = "Current date setting is ";
   // Convert to current locale.
   s += d.toLocaleString();
   return(s);
}
function toLocaleStringNumber() {
   var n = Math.PI;
   var s = "The value of Pi is: ";
   s+= n.toLocaleString();
   return(s);
}
```

必要条件

Version 1

対象

Array オブジェクト| Date オブジェクト| Number オブジェクト| Object オブジェクト

参照

その他の技術情報

メソッド

toLocaleTimeString メソッド

ホスト環境の現在のロケールに対応した時刻を文字列値で返します。

```
function toLocaleTimeString() : String
```

解説

toLocaleTimeString メソッドは、現在のタイムゾーンに従って、時刻を読みやすい形式の文字列値にして返します。時刻の形式は、ホスト環境の現在のロケールの既定形式になります。このメソッドの戻り値はコンピュータ間で一致しないため、スクリプト用には適していません。**toLocalTimeString** メソッドは、表示をフォーマットするためだけに使用し、計算には使用しないようにしてください。

必要条件

Version 5.5

対象

Date オブジェクト

参照

関連項目

toTimeString メソッド
toLocaleDateString メソッド

toLocaleUpperCase メソッド

ホスト環境の現在のロケールに従って、すべての英字を大文字に変換した文字列を返します。

```
function tolocaleUpperCase() : String
```

解説

toLocaleUpperCase メソッドは、ホスト環境の現在のロケールに従って、文字列の文字を変換します。ほとんどの場合、**toUpperCase** メソッドの実行結果と同じになります。結果が異なるのは、言語規則が通常の Unicode の大文字、小文字のマッピングと競合する場合です。

必要条件

Version 5.5

対象

String オブジェクト

参照
関連項目
toLocaleLowerCase メソッド
toUpperCase メソッド

toLowerCase メソッド

すべての英字を小文字に変換した文字列を返します。

```
function toLowerCase() : String
```

解説

toLowerCase メソッドは、英字だけに有効です。

使用例

toLowerCase メソッドの使用例を次に示します。

```
var strVariable = "This is a STRING object";
strVariable = strVariable.toLowerCase();
```

2 つ目のステートメントの実行後、変数 `strVariable` は次の値になります。

```
this is a string object
```

必要条件

Version 1

対象

String オブジェクト

参照

関連項目

toUpperCase メソッド

toPrecision メソッド

指定された桁数の指数表記または固定小数点表記の数値を文字列で返します。

```
function toPrecision ( [precision : Number] ) : String
```

引数
precision

省略可能です。有効桁数を指定します。1 ～ 21 の範囲で指定します。

解説

指数表記の数値では、(*precision* - *1*) 桁を小数点以下の数値として返します。固定小数点表記の数値では、*precision* が有効桁数となります。

precision を指定しない場合、または **undefined** である場合は、**toString** メソッドが代わりに呼び出されます。

必要条件

Version 5.5

対象

Number オブジェクト

参照
関連項目
toFixed メソッド
toExponential メソッド

toString メソッド

オブジェクトの値を表す文字列を返します。

```
function toString( [radix : Number] ) : String
```

引数
radix

省略可能です。数値を文字列に変換するときの方法を指定します。この値は、数値にのみ使用されます。

解説

toString メソッドは、JScript の組み込みオブジェクトのメンバです。このメソッドの動作は、オブジェクトの種類に応じて異なります。

オブジェクト	動作
Array	文字列に変換された **Array** オブジェクトの要素を返します。文字列は、コンマで接続されます。
Boolean	ブール値が真 (**true**) の場合は、"true" という文字列を返します。それ以外の場合は、"false" を返します。
Date	日付を表すテキストを返します。
Error	対応するエラーメッセージの文字列を返します。
Function	次の形式の文字列を返します。*functionname* は、指定した関数の名前です。 "function functionname() { [ネイティブ コード] }"
Number	数値を表すテキストを返します。
String	**String** オブジェクトの値を返します。
既定	"[object objectname]" という文字列を返します。objectname は、オブジェクト型の名前です。

使用例

引数 *radix* を使った **toString** メソッドの使用例を次に示します。この関数の戻り値は、基数変換テーブルです。

```
function CreateRadixTable (){
   var s, s1, s2, s3, x;                    //Declare variables.
   s = "Hex    Dec   Bin \n";               //Create table heading.
   for (x = 0; x < 16; x++)                 //Establish size of table
   {                                        // in terms of number of
      switch(x)                             // values shown.
      {                                     //Set intercolumn spacing.
         case 0 :
            s1 = "      ";
            s2 = "    ";
            s3 = "   ";
            break;
         case 1 :
            s1 = "      ";
            s2 = "    ";
            s3 = "   ";
            break;
         case 2 :
            s3 = "  ";
            break;
         case 3 :
            s3 = "  ";
```

```
            break;
         case 4 :
            s3 = " ";
            break;
         case 5 :
            s3 = " ";
            break;
         case 6 :
            s3 = " ";
            break;
         case 7 :
            s3 = " ";
            break;
         case 8 :
            s3 = "" ;
            break;
         case 9 :
            s3 = "";
            break;
         default:
            s1 = "     ";
            s2 = "";
            s3 = "    ";
      }                                    //Convert to hex, decimal & binary.
      s += " " + x.toString(16) + s1 + x.toString(10)
      s +=  s2 + s3 + x.toString(2)+ "\n";

   }
   return(s);                              //Return entire radix table.
}
```

必要条件

Version 2

対象：

Array オブジェクト| Boolean オブジェクト| Date オブジェクト| Error オブジェクト| Function オブジェクト| Number オブジェクト| Object オブジェクト| String オブジェクト

参照

関連項目

function ステートメント

toTimeString メソッド

時刻を文字列値として返します。

```
function toTimeString() : String
```

解説

toTimeString メソッドは、現在のタイムゾーンに従って、時刻を便利で読みやすい形式の文字列値にして返します。

必要条件

Version 5.5

対象：

Date オブジェクト

参照

関連項目

toDateString メソッド
toLocaleTimeString メソッド

toUpperCase メソッド

すべての英字を大文字に変換した文字列を返します。

```
function toUpperCase() : String
```

解説

toUpperCase メソッドは、英字だけに有効です。

使用例

toUpperCase メソッドの使用例を次に示します。

```
var strVariable = "This is a STRING object";
strVariable = strVariable.toUpperCase();
```

2 つ目のステートメントの実行後、変数 `strVariable` は次の値になります。

```
THIS IS A STRING OBJECT
```

必要条件

Version 1

String オブジェクト

参照

関連項目

toLowerCase メソッド

toUTCString メソッド

日付を世界協定時刻 (UTC) の文字列に変換します。

```
function toUTCString() : String
```

解説

toUTCString メソッドは、世界協定時刻 (UTC) を表す **String** オブジェクトを返します。

使用例

toUTCString メソッドの使用例を次に示します。

```
function toUTCStrDemo(){
   var d, s;                    //Declare variables.
   d = new Date();              //Create Date object.
   s = "Current setting is ";
   s += d.toUTCString();        //Convert to UTC string.
   return(s);                   //Return UTC string.
}
```

必要条件

Version 3

対象

Date オブジェクト

参照

関連項目

toGMTString メソッド

ubound メソッド

VBArray の指定された次元での最も大きいインデックス値を返します。

```
function ubound( [dimension : Number] ) : Number
```

引数
dimension

省略可能です。VBArray の、最も大きいインデックスを取得する次元を指定します。省略した場合は、1 が指定されたものとします。

解説

VBArray が空の場合、**ubound** メソッドは undefined を返します。引数 *dimension* に VBArray の次元数より大きい値を指定したり、負の値を指定したりすると、"有効範囲外のインデックス" エラーが発生します。

使用例

次のコードは、3 つの部分から構成されます。最初の部分は、Visual Basic のセーフ配列を作成する VBScript のコードです。2 番目の部分は、このセーフ配列の次元数と各次元の上限値を取得する JScript のコードです。どちらのコードも、HTML ページの <HEAD> セクションに記述します。3 番目の部分は、他の 2 つのコードを実行するために <BODY> セクションに記述する JScript のコードです。

```
<HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Function CreateVBArray()
   Dim i, j, k
   Dim a(2, 2)
   k = 1
   For i = 0 To 2
      For j = 0 To 2
         a(j, i) = k
         k = k + 1
      Next
   Next
   CreateVBArray = a
End Function
-->
</SCRIPT>

<SCRIPT LANGUAGE="JScript">
<!--
function VBArrayTest(vba)
{
   var i, s;
   var a = new VBArray(vba);
   for (i = 1; i <= a.dimensions(); i++)
   {
      s = "The upper bound of dimension ";
      s += i + " is ";
      s += a.ubound(i)+ ".<BR>";
      return(s);
   }
}
-->
</SCRIPT>
</HEAD>

<BODY>
<SCRIPT language="jscript">
   document.write(VBArrayTest(CreateVBArray()));
</SCRIPT>
</BODY>
```

必要条件

Version 3

対象

VBArray オブジェクト

参照

関連項目

dimensions メソッド
getItem メソッド
lbound メソッド
toArray メソッド

unescape メソッド

escape メソッドでエンコードされた **String** オブジェクトの文字列をデコードして返します。

```
function unescape(charString : String) : String
```

引数
charString

必ず指定します。デコードする **String** オブジェクトの名前またはリテラルを指定します。

解説

unescape メソッドは、*charString* の内容を文字列値として返します。*%xx* 形式でエンコードされたすべての文字は、ASCII 文字セットの対応する文字に置き換えられます。

%u*xxxx* 形式 (Unicode 文字) にエンコードされた文字は、Unicode 文字に置き換えられます。その際、*xxxx* は 16 進数に変換されます。

メモ：

unescape メソッドを URI (Uniform Resource Identifier) のデコードに使用しないでください。代わりに、**decodeURI** メソッドまたは **decodeURIComponent** メソッドの使用をお勧めします。

必要条件

Version 1

対象：

Global オブジェクト

参照
関連項目
decodeURI メソッド
decodeURIComponent メソッド
escape メソッド
String オブジェクト

unshift メソッド

指定した要素を配列の先頭に挿入します。

```
function unshift([item1 : Object [, ... [, itemN : Object]]]) : Array
```

引数
item1, ..., itemN

省略可能です。**Array** の先頭に挿入する要素を指定します。

解説

unshift メソッドでは、引数の並びと同じ順序で要素を配列の先頭に挿入します。

必要条件

Version 5.5

対象

Array オブジェクト

参照
関連項目
shift メソッド

UTC メソッド

世界協定時刻 (UTC) (つまり GMT) の 1970 年 1 月 1 日 0 時 0 分 0 秒と、指定した日付との間をミリ秒単位で返します。

```
function UTC(year : Number , month : Number , day : Number [, hours : Number [, minutes : N
umber [, seconds : Number [,ms : Number]]]]) : Number
```

引数

year

必ず指定します。日付の年数を必ず 4 桁で指定します。引数 *year* に 0 ～ 99 の値が指定された場合、1900 + *year* の値が指定されたと見なされます。

month

必ず指定します。月を表す 0 ～ 11 (1 ～ 12 月に相当) の範囲内の整数を指定します。

day

必ず指定します。日を表す 1 ～ 31 の範囲内の整数を指定します。

hours

省略可能です。引数 *minutes* を指定する場合は、この引数を指定する必要があります。時を表す 0 ～ 23 (午前 0 時 ～ 午後 11 時に対応) の範囲内の整数を指定します。

minutes

省略可能です。引数 *seconds* を指定する場合は、この引数を指定する必要があります。分を表す 0 ～ 59 の範囲内の整数を指定します。

seconds

省略可能です。引数 *milliseconds* を指定する場合は、この引数を指定する必要があります。秒を表す 0 ～ 59 の範囲内の整数を指定します。

ms

省略可能です。ミリ秒を表す 0 ～ 999 の範囲内の整数を指定します。

解説

UTC メソッドは、世界協定時刻での 1970 年 1 月 1 日 0 時 0 分 0 秒と指定した日付との間をミリ秒単位で返します。この戻り値は、**setTime** メソッドおよび **Date** オブジェクト コンストラクタで使用できます。引数に有効範囲を超える値や負の値を指定すると、値に応じて格納される他の値が変更されます。たとえば、150 秒を指定すると、2 分 30 秒として処理されます。

UTC メソッドと日付を受け取る **Date** オブジェクトのコンストラクタとの違いは、渡した日付が **UTC** メソッドでは世界協定時刻 (UTC) として扱われ、**Date** オブジェクトのコンストラクタでは現地時間として扱われるという点です。

UTC メソッドは、静的なメソッドです。したがって、メソッドを使用するために、あらかじめ **Date** オブジェクトを作成しておく必要はありません。

メモ :
引数 *year* に 0 ～ 99 の整数を指定すると、*1900 + year* が年として使用されます。

使用例

UTC メソッドの使用例を次に示します。

```
function DaysBetweenDateAndNow(yr, mo, dy){
   var d, r, t1, t2, t3;            //Declare variables.
   var MinMilli = 1000 * 60         //Initialize variables.
   var HrMilli = MinMilli * 60
   var DyMilli = HrMilli * 24
   t1 = Date.UTC(yr, mo - 1, dy)    //Get milliseconds since 1/1/1970.
   d = new Date();                  //Create Date object.
   t2 = d.getTime();                //Get current time.
```

```
    if (t2 >= t1)
       t3 = t2 - t1;
    else
       t3 = t1 - t2;
    r = Math.round(t3 / DyMilli);
    return(r);                          //Return difference.
 }
```

必要条件

Version 1

対象

Date オブジェクト

参照

関連項目

setTime メソッド

valueOf メソッド

指定されたオブジェクトのプリミティブ値を返します。

```
function valueOf() : Object
```

解説

valueOf メソッドの動作は、オブジェクトの種類に応じて異なります。

オブジェクト	戻り値
Array	配列のインスタンスです。
Boolean	オブジェクトに格納されているブール値を返します。
Date	世界協定時刻 (UTC) の 1970 年 1 月 1 日 0 時 0 分 0 秒からの経過時間を表すミリ秒単位の数値を返します。
Function	関数自体を返します。
Number	オブジェクトに格納されている数値を返します。
Object	オブジェクト自体を返します。これは、既定の設定です。
String	オブジェクトに格納されている文字列を返します。

Math オブジェクトおよび **Error** オブジェクトには、**valueOf** メソッドはありません。

必要条件

Version 2

対象

Array オブジェクト| Boolean オブジェクト| Date オブジェクト| Function オブジェクト| Number オブジェクト| Object オブジェクト| String オブジェクト

参照

関連項目

toString メソッド

修飾子

JScript の修飾子は、クラス、インターフェイス、またはクラスやインターフェイスのメンバの動作や参照可能範囲に影響を与えます。修飾子は、クラスやインターフェイスを定義するときに使用できますが、必須ではありません。

このセクションの内容

abstract 修飾子

クラスおよびクラス メンバを定義する継承の修飾子です。実装を与えることはできません。

expando 修飾子

動的に拡張可能なクラス、または expando オブジェクト コンストラクタのメソッドを表す互換性の修飾子です。

final 修飾子

クラスの拡張、またはメソッドやプロパティのオーバーライドを禁止する継承の修飾子です。

hide 修飾子

メソッドやプロパティが、基本クラスのメソッドやプロパティをオーバーライドすることを禁止するバージョン セーフ修飾子です。

internal 修飾子

クラス、インターフェイス、またはメンバの参照可能範囲を現在のパッケージだけにする可視性修飾子です。

override 修飾子

基本クラスのメソッドを明示的にオーバーライドするバージョン セーフ修飾子です。

private 修飾子

クラス メンバの参照可能範囲を同じクラスのメンバだけにする可視性修飾子です。

protected 修飾子

クラスまたはインターフェイスのメンバの参照可能範囲を、現在のクラスまたはインターフェイス、および現在のクラスの派生クラスだけにする可視性修飾子です。

public 修飾子

クラスまたはインターフェイスのメンバを、クラスまたはインターフェイスにアクセスするすべてのコードから参照できるようにする可視性修飾子です。

static 修飾子

クラス メンバをクラス自体に属するようにする修飾子です。

関連するセクション

JScript の修飾子

JScript の修飾子の目的と使用方法に関する概要を示します。

abstract 修飾子

クラスの拡張が必要であること、またはメソッドやプロパティの実装が派生クラスで提供される必要があることを宣言します。

```
abstract statement
```

引数
statement

必ず指定します。クラス、メソッド、またはプロパティの定義。

解説

abstract 修飾子は、実装を持たないクラスのメソッドまたはプロパティ、またはこれらのメソッドを含むクラスに対して使用します。抽象メンバを持つクラスは、**new** 演算子ではインスタンス化できません。抽象基本クラスから、抽象クラスおよび非抽象クラスの両方を派生できます。

abstract 修飾子は、クラスのメソッドとプロパティ、およびクラスに指定できます。**abstract** メンバを含むクラスには、**abstract** 修飾子を指定する必要があります。インターフェイスとそのメンバは暗黙的に抽象であり、**abstract** 修飾子を使用できません。フィールドには **abstract** を指定できません。

abstract 修飾子は、ほかの継承の修飾子 (**final**) と共に使用することはできません。既定では、クラスのメンバは **abstract** と **final** のどちらでもありません。継承の修飾子は、**static** 修飾子と共に使用することはできません。

使用例

次のコードは、**abstract** 修飾子の使用例です。

```
// CAnimal is an abstract base class.
abstract class CAnimal {
   abstract function printQualities();
}
// CDog and CKangaroo are derived classes of CAnimal.
class CDog extends CAnimal {
   function printQualities() {
      print("A dog has four legs.");
   }
}
class CKangaroo extends CAnimal {
   function printQualities() {
      print("A kangaroo has a pouch.");
   }
}

// Define animal of type CAnimal.
var animal : CAnimal;

animal = new CDog;
// animal uses printQualities from CDog.
animal.printQualities();

animal = new CKangaroo;
// animal uses printQualities from CKangaroo.
animal.printQualities();
```

このプログラムの出力は次のようになります。

```
A dog has four legs.
A kangaroo has a pouch.
```

必要条件

バージョン .NET

参照

関連項目

final 修飾子
static 修飾子
var ステートメント
function ステートメント
class ステートメント
new 演算子

概念

変数と定数のスコープ

その他の技術情報

修飾子

expando 修飾子

クラスのインスタンスが expando プロパティをサポートすること、またはメソッドが expando オブジェクト コンストラクタであることを宣言します。

```
expando statement
```

引数
statement

必ず指定します。クラスまたはメソッドの定義。

解説

expando 修飾子は、クラスを動的に拡張できる (expando プロパティをサポートする) ことを示します。**expando** クラス インスタンスの expando プロパティは、**[]** 表記を使ってアクセスする必要があります。ドット演算子ではアクセスできません。**expando** 修飾子は、メソッドが **expando** オブジェクト コンストラクタであることを示します。

expando 修飾子は、クラスとクラスのメソッドに指定できます。フィールド、プロパティ、インターフェイス、およびインターフェイスのメンバには、**expando** 修飾子を使用できません。

拡張クラスには **Item** という名前の隠ぺいされたプライベート プロパティがあり、**Object** パラメータを 1 つ受け取り、**Object** を返します。**expando** クラスでは、このシグネチャを持つプロパティは定義できません。

例 1

次のコードは、クラスに対する **expando** 修飾子の使用例です。expando クラスは JScript の **Object** に似ていますが、次に示すような違いがあります。

```
        expando class CExpandoExample {
   var x : int = 10;
}

// New expando class-based object.
var testClass : CExpandoExample = new CExpandoExample;
// New JScript Object.
var testObject : Object = new Object;

// Add expando properties to both objects.
testClass["x"] = "ten";
testObject["x"] = "twelve";

// Access the field of the class-based object.
print(testClass.x);      // Prints 10.
// Access the expando property.
print(testClass["x"]);   // Prints ten.

// Access the property of the class-based object.
print(testObject.x);     // Prints twelve.
// Access the same property using the [] operator.
print(testObject["x"]);  // Prints twelve.
```

このコードの出力は、次のようになります。

```
10
ten
twelve
twelve
```

例 2

次のコードは、メソッドに対する **expando** 修飾子の使用例です。expando メソッドを通常の方法で呼び出すと、メソッドはフィールド *x* にアクセスします。**new** 演算子を使って、メソッドを明示的なコンストラクタとして使用すると、新しいオブジェクトに expando プロパティが追加されま

す。

```
class CExpandoExample {
   var x : int;
   expando function constructor(val : int) {
      this.x = val;
      return "Method called as a function.";
   }
}

var test : CExpandoExample = new CExpandoExample;
// Call the expando method as a function.
var str = test.constructor(123);
print(str);        // The return value is a string.
print(test.x);     // The value of x has changed to 123.

// Call the expando method as a constructor.
var obj = new test.constructor(456);
// The return value is an object, not a string.
print(obj.x);      // The x property of the new object is 456.
print(test.x);     // The x property of the original object is still 123.
```

このコードの出力は、次のようになります。

```
Method called as a function.
123
456
123
```

必要条件

バージョン .NET

参照

関連項目

static 修飾子
var ステートメント
function ステートメント
class ステートメント

概念

変数と定数のスコープ
型の注釈

その他の技術情報

修飾子

final 修飾子

拡張できないクラス、またはオーバーライドできないメソッドまたはプロパティを宣言します。

```
final statement
```

引数
statement

必ず指定します。クラス、メソッド、またはプロパティの定義。

解説

final 修飾子は、拡張できないクラス、またはオーバーライドできないメソッドやプロパティであることを示すために使用します。これにより、重要な関数がほかの関数によってオーバーライドされることはなくなり、クラスの動作は維持されます。**final** 修飾子を指定したメソッドは、派生クラスのメソッドによって隠ぺいまたはオーバーロードされる場合があります。

final 修飾子は、クラスのメソッドとプロパティ、およびクラスに指定できます。インターフェイス、フィールド、およびインターフェイスのメンバには、**final** 修飾子を使用できません。

final 修飾子は、ほかの継承の修飾子 (**abstract**) と共に使用することはできません。既定では、クラスのメンバは **abstract** と **final** のどちらでもありません。継承の修飾子は、**static** 修飾子と共に使用することはできません。

使用例

次のコードは、**final** 修飾子の使用例です。**final** 修飾子は、基本クラスのメソッドが派生クラスのメソッドによってオーバーライドされることを防ぎます。

```
class CBase {
   final function methodA() { print("Final methodA of CBase.") };
   function methodB() { print("Non-final methodB of CBase.") };
}

class CDerived extends CBase {
   function methodA() { print("methodA of CDerived.") };
   function methodB() { print("methodB of CDerived.") };
}

var baseInstance : CBase = new CDerived;
baseInstance.methodA();
baseInstance.methodB();
```

このプログラムの出力は、final メソッドがオーバーライドされていないことを示しています。

```
Final methodA of CBase.
methodB of CDerived.
```

必要条件

バージョン .NET

参照
関連項目
abstract 修飾子
hide 修飾子
override 修飾子
var ステートメント
function ステートメント
class ステートメント
概念
変数と定数のスコープ
型の注釈

その他の技術情報

修飾子

hide 修飾子

メソッドまたはプロパティが、基本クラスのメソッドまたはプロパティを隠ぺいすることを宣言します。

```
hide statement
```

引数
statement

必ず指定します。メソッドまたはプロパティの定義。

解説

hide 修飾子は、基本クラスのメソッドを隠ぺいするメソッドに対して使用します。基本クラスに同じシグネチャのメンバがない場合、メソッドに **hide** 修飾子は使用できません。

hide 修飾子は、クラスのメソッドとプロパティに指定できます。クラス、フィールド、インターフェイス、およびインターフェイスのメンバには、**hide** 修飾子を使用できません。

hide 修飾子は、ほかのバージョン セーフ修飾子 (**override**) と共に使用することはできません。バージョン セーフ修飾子は、**static** 修飾子と共に使用することはできません。既定では、基本クラスのメソッドに **final** 修飾子がない場合、メソッドは基本クラスのメソッドをオーバーライドします。abstract を指定した基本メソッドに明示的な実装を与えていない場合は、**abstract** メソッドを隠ぺいできません。バージョン セーフ モードで実行されている場合は、基本クラスのメソッドをオーバーライドするときに、バージョン セーフ修飾子のいずれかを使用する必要があります。

使用例

次のコードは、**hide** 修飾子の使用例です。**hide** 修飾子を指定された派生クラスのメソッドは、基本クラスのメソッドをオーバーライドしません。**override** 修飾子を指定されたメソッドは、基本クラスのメソッドをオーバーライドします。

```
class CBase {
   function methodA() { print("methodA of CBase.") };
   function methodB() { print("methodB of CBase.") };
}

class CDerived extends CBase {
   hide function methodA() { print("Hiding methodA.") };
   override function methodB() { print("Overriding methodB.") };
}


var derivedInstance : CDerived = new CDerived;
derivedInstance.methodA();
derivedInstance.methodB();

var baseInstance : CBase = derivedInstance;
baseInstance.methodA();
baseInstance.methodB();
```

このプログラムの出力は、隠ぺいされたメソッドは基本クラスのメソッドをオーバーライドしないことを示します。

```
Hiding methodA.
Overriding methodB.
methodA of CBase.
Overriding methodB.
```

必要条件

バージョン .NET

参照
関連項目
override 修飾子
static 修飾子

internal 修飾子

クラス、クラス メンバ、インターフェイス、またはインターフェイス メンバの参照可能範囲が内部であることを宣言します。

```
internal statement
```

引数
statement

必ず指定します。クラス、インターフェイス、またはメンバの定義。

解説

internal 修飾子は、クラス、インターフェイス、またはメンバを現在のパッケージ内だけで参照できるようにします。現在のパッケージの外部のコードからは、**internal** メンバにアクセスできません。

internal 修飾子は、クラスおよびインターフェイスに指定できます。グローバル スコープでは、**internal** 修飾子は **public** 修飾子と同じです。**internal** 修飾子は、クラスまたはインターフェイスのメンバにも指定できます。

internal 修飾子は、他の可視性修飾子 (**public**、**private**、または **protected**) と共に使用することはできません。可視性修飾子は、定義されているスコープに対して相対的に機能します。たとえば、**internal** クラスの **public** メソッドに対するアクセスはパブリックではありませんが、このクラスにアクセスするコードの場合は、すべてのコードからこのメソッドにアクセスできます。

必要条件

バージョン .NET

参照
関連項目
public 修飾子
private 修飾子
protected 修飾子
var ステートメント
function ステートメント
class ステートメント
概念
変数と定数のスコープ
その他の技術情報
修飾子

override 修飾子

メソッドまたはプロパティが、基本クラスのメソッドまたはプロパティをオーバーライドすることを宣言します。

```
override statement
```

引数
statement

必ず指定します。メソッドまたはプロパティの定義。

解説

override 修飾子は、基本クラスのメソッドをオーバーライドするメソッドに対して使用します。基本クラスに同じシグネチャのメンバがない場合、**override** 修飾子は使用できません。

override 修飾子は、クラスのメソッドとプロパティに指定できます。クラス、フィールド、インターフェイス、およびインターフェイスのメンバには、**override** 修飾子を使用できません。

override 修飾子は、ほかのバージョン セーフ修飾子 (**hide**) と共に使用することはできません。バージョン セーフ修飾子は、**static** 修飾子と共に使用することはできません。既定では、基本クラスのメソッドに **final** 修飾子がない場合、メソッドは基本クラスのメソッドをオーバーライドします。**final** メソッドはオーバーライドできません。バージョン セーフ モードで実行されている場合は、基本クラスのメソッドをオーバーライドするときに、バージョン セーフ修飾子のいずれかを使用する必要があります。

使用例

次のコードは、**override** 修飾子の使用例です。**override** 修飾子を指定された派生クラスのメソッドは、基本クラスのメソッドをオーバーライドします。**hide** 修飾子を指定されたメソッドは、基本クラスのメソッドをオーバーライドしません。

```
class CBase {
   function methodA() { print("methodA of CBase.") };
   function methodB() { print("methodB of CBase.") };
}

class CDerived extends CBase {
   hide function methodA() { print("Hiding methodA.") };
   override function methodB() { print("Overriding methodB.") };
}


var derivedInstance : CDerived = new CDerived;
derivedInstance.methodA();
derivedInstance.methodB();

var baseInstance : CBase = derivedInstance;
baseInstance.methodA();
baseInstance.methodB();
```

このプログラムの出力は、**override** メソッドが基本クラスのメソッドをオーバーライドすることを示しています。

```
Hiding methodA.
Overriding methodB.
methodA of CBase.
Overriding methodB.
```

必要条件

バージョン .NET

参照
関連項目
hide 修飾子
var ステートメント

private 修飾子

クラス メンバの参照可能範囲がプライベートであることを宣言します。

```
private statement
```

引数
statement

必ず指定します。クラス メンバの定義。

解説

private 修飾子は、クラスのメンバをそのクラス内だけで参照できるようにします。派生クラスを含め、現在のクラスの外部からは **private** メンバにアクセスできません。

グローバル スコープのクラスおよびインターフェイスには、**private** 修飾子を指定できません。**private** 修飾子は、クラスまたはインターフェイスのメンバ (入れ子になったクラスおよびインターフェイスを含む) に指定できます。

private 修飾子は、他の可視性修飾子 (**public**、**protected**、または **internal**) と共に使用することはできません。

必要条件

バージョン .NET

参照
関連項目
public 修飾子
protected 修飾子
internal 修飾子
var ステートメント
function ステートメント
class ステートメント
概念
変数と定数のスコープ
その他の技術情報
修飾子

protected 修飾子

クラス メンバまたはインターフェイス メンバの参照可能範囲が、プロテクトであることを宣言します。

```
protected statement
```

引数
statement

必ず指定します。クラス メンバまたはインターフェイス メンバの定義。

解説

protected 修飾子は、クラスまたはインターフェイスのメンバを、そのクラスまたはインターフェイス、および現在のクラスのすべての派生クラス内だけで参照できるようにします。現在のクラスの外部からは、**protected** メンバにアクセスできません。

グローバル スコープのクラスおよびインターフェイスには、**protected** 修飾子を指定できません。**protected** 修飾子は、クラスまたはインターフェイスのメンバ (入れ子になったクラスおよびインターフェイスを含む) に指定できます。

protected 修飾子は、他の可視性修飾子 (**public**、**private**、または **internal**) と共に使用することはできません。

必要条件

バージョン .NET

参照
関連項目
public 修飾子
private 修飾子
internal 修飾子
var ステートメント
function ステートメント
class ステートメント
概念
変数と定数のスコープ
その他の技術情報
修飾子

public 修飾子

クラス、インターフェイス、またはメンバの参照可能範囲がパブリックであることを宣言します。

```
public statement
```

引数
statement

必ず指定します。クラス、インターフェイス、またはメンバの定義。

解説

public 修飾子は、クラスのメンバを、クラスにアクセスするすべてのコードから参照できるようにします。

すべてのクラスおよびインターフェイスは、既定で **public** です。**public** 修飾子は、クラスまたはインターフェイスのメンバに指定できます。

public 修飾子は、ほかの可視性修飾子 (**private**、**protected**、または **internal**) と共に使用することはできません。

必要条件

バージョン .NET

参照
関連項目
private 修飾子
protected 修飾子
internal 修飾子
var ステートメント
function ステートメント
class ステートメント
概念
変数と定数のスコープ
その他の技術情報
修飾子

static 修飾子

クラス メンバが、クラスのインスタンスではなくクラスに属していることを宣言します。

```
static statement
```

引数
statement

必ず指定します。クラス メンバの定義。

解説

static 修飾子は、メンバがクラスのインスタンスではなくクラス自身に属していることを示します。クラスのインスタンスが複数作成された場合でも、**static** メンバのコピーは、指定したアプリケーションに 1 つしか存在しません。**static** メンバには、インスタンスへの参照ではなく、クラスへの参照を使用することでアクセスできます。ただし、クラス メンバの宣言では、**this** オブジェクトを使用して **static** メンバにアクセスできます。

static 修飾子は、クラスのメンバに指定できます。クラス、インターフェイス、およびインターフェイスのメンバには、**static** 修飾子を使用できません。

static 修飾子は、継承の修飾子 (**abstract** および **final**) またはバージョン セーフ修飾子 (**hide** および **override**) と共に使用することはできません。

static 修飾子と **static** ステートメントを混同しないでください。**static** 修飾子は、メンバがクラスのインスタンスではなくクラス自身に属していることを示します。

使用例

次のコードは、**static** 修飾子の使用例です。

```
class CTest {
   var nonstaticX : int;      // A non-static field belonging to a class instance.
   static var staticX : int;  // A static field belonging to the class.
}

// Initialize staticX. An instance of test is not needed.
CTest.staticX = 42;

// Create an instance of test class.
var a : CTest = new CTest;
a.nonstaticX = 5;
// The static field is not directly accessible from the class instance.

print(a.nonstaticX);
print(CTest.staticX);
```

このプログラムの出力は次のようになります。

```
5
42
```

必要条件

バージョン .NET

参照
関連項目
expando 修飾子
var ステートメント
function ステートメント
class ステートメント
static ステートメント
概念

変数と定数のスコープ
型の注釈
その他の技術情報
修飾子

オブジェクト (JScript)

JScript のオブジェクトは、プロパティとメソッドのコレクションです。ここでは、JScript オブジェクトの使い方を説明する情報へのリンクを示します。

メモ：
JScript ランタイムは、スレッド セーフ機能を持つようにはデザインされていません。このため、JScript のオブジェクトとメソッドをマルチスレッド アプリケーションで使用した場合、予期しない動作をすることがあります。

このセクションの内容

ActiveXObject オブジェクト

オートメーション オブジェクトへの参照を有効にして返します。

arguments オブジェクト

現在の関数へ渡された引数にアクセスする手段を提供します。

Array オブジェクト

任意のデータ型の配列を作成する手段を提供します。

Boolean オブジェクト

新しいブール値を作成します。

Date オブジェクト

日付および時刻を格納しておくオブジェクトです。必要に応じて、データの必要な部分を取り出すことができます。

Enumerator オブジェクト

コレクション内の項目を列挙する手段を提供します。

Error オブジェクト

JScript のコードの実行中に発生するエラーに関する情報が含まれているオブジェクトです。

Function オブジェクト

新しい関数を作成します。

Global オブジェクト

グローバルなメソッドを 1 つのオブジェクトに集めておくための組み込みオブジェクトです。

Math オブジェクト

数値計算のための基本的な演算機能と定数を提供する組み込みオブジェクトです。

Number オブジェクト

数値型のデータを表すオブジェクトです。数値定数のプレースホルダを入れておくこともできます。

Object オブジェクト

すべての JScript オブジェクトに共通の機能を提供します。

RegExp オブジェクト

正規表現パターンの検索に関する情報が格納されるオブジェクトです。

Regular Expression オブジェクト

正規表現パターンを格納するオブジェクトです。

String オブジェクト

テキスト文字列を表すオブジェクトです。このオブジェクトを使用すると、各種文字列操作、文字列の書式設定、文字列内の一部分の取得、文字列内での指定した文字列の検索などを行うことができます。

VBArray オブジェクト

Visual Basic のセーフ配列にアクセスする手段を提供します。

関連するセクション

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

メソッド

JScript で使用できるメソッドのアルファベット順の一覧と、メソッドの各カテゴリへのリンクを示します。

プロパティ

JScript で使用できるプロパティと、各プロパティの正しい構文を説明するトピックへのリンクを示します。

JScript オブジェクト

JScript のオブジェクトの概念と、オブジェクトがプロパティとメソッドにどのように関連するかについて説明します。JScript がサポートするオブジェクトの詳細を説明するトピックへのリンクも示します。

ActiveXObject オブジェクト

オートメーション オブジェクトへのインターフェイスを提供するオブジェクトです。

```
function ActiveXObject(ProgID : String [, location : String])
```

引数
ProgID

必ず指定します。"*serverName.typeName*" 形式の文字列。*serverName* はオブジェクトを提供しているアプリケーションの名前、*typeName* は作成するオブジェクトの型またはクラス名を表します。

location

省略可能です。オブジェクトの作成先のネットワーク サーバーの名前。

解説

一般的に、オートメーション サーバーでは、少なくとも 1 種類のオブジェクトが提供されます。たとえば、ワードプロセッシング アプリケーションでは、アプリケーション オブジェクト、ドキュメント オブジェクト、およびツール バー オブジェクトが提供されます。

次のコードは、**ActiveXObject** オブジェクト コンストラクタを呼び出して、アプリケーション (この場合は、Microsoft Excel ワークシート) を起動しています。**ActiveXObject** を使用して、コード内でアプリケーションを参照できます。次の例を使用すると、オブジェクト変数 `ExcelSheet` と、Application オブジェクトや ActiveSheet.Cells コレクションなどの Excel のオブジェクトを使って、新しいオブジェクトのプロパティやメソッドにアクセスできます。

```
// Declare the variables
var Excel, Book;

// Create the Excel application object.
Excel = new ActiveXObject("Excel.Application");

// Make Excel visible.
Excel.Visible = true;

// Create a new work book.
Book = Excel.Workbooks.Add()

// Place some text in the first cell of the sheet.
Book.ActiveSheet.Cells(1,1).Value = "This is column A, row 1";

// Save the sheet.
Book.SaveAs("C:\\TEST.XLS");

// Close Excel with the Quit method on the Application object.
Excel.Application.Quit();
```

リモート サーバー上でオブジェクトを作成できるのは、インターネット セキュリティが機能していない場合だけです。コンピュータ名を **ActiveXObject** 関数の引数 *servername* に渡すことで、リモート ネットワークで接続されたコンピュータ上にオブジェクトを作成できます。この名前は、共有名のコンピュータ名の部分と同じです。たとえば、"\\MyServer\public" というネットワーク共有名の場合、*servername* は "MyServer" です。さらに、DNS 形式または IP アドレスを使用して *servername* を指定することもできます。

次のコードは、"MyServer" というリモート ネットワーク コンピュータ上で実行されている Excel のインスタンスのバージョン番号を返す例です。

```
function GetAppVersion() {
   var Excel = new ActiveXObject("Excel.Application", "MyServer");
   return(Excel.Version);
}
```

指定したリモート サーバーがネットワーク上に存在しないか見つからない場合は、エラーが発生します。

プロパティおよびメソッド

ActiveXObject オブジェクトには、組み込みのプロパティやメソッドがありません。このため、オートメーション オブジェクトのプロパティおよびメソッドにアクセスできます。

必要条件

Version 1

参照

関連項目

new 演算子

GetObject 関数 (JScript 8.0)

arguments オブジェクト

現在実行中の関数、その引数、およびその関数の呼び出し元の関数を表すオブジェクトです。このオブジェクトを明示的に構築することはできません。

プロパティ

arguments オブジェクトのプロパティ

メソッド

arguments オブジェクトには、メソッドはありません。

必要条件

Version 1

解説

arguments オブジェクトは、各関数について、それぞれの実行が開始されるときにインスタンス化されます。**arguments** オブジェクトには、関連する関数のスコープ内からだけ直接アクセスできます。

関数に渡されるすべてのパラメータとパラメータの数は、**arguments** オブジェクトに格納されます。関数の **arguments** オブジェクトは配列ではありませんが、配列の各要素にアクセスする場合と同じ [ ] 表記を使用して、各引数にアクセスできます。

arguments オブジェクトを使用すると、任意の数の引数を受け取る関数を作成できます。この機能は、関数の定義時にパラメータ配列の構築を使用することでも実現できます。詳細については、**function** ステートメントのトピックを参照してください。

メモ :

arguments オブジェクトは、高速モードで実行されている場合は利用できません。JScript の既定のモードは高速モードです。**arguments** オブジェクトを使用するプログラムをコマンド ラインからコンパイルするには、**/fast-** を使用して fast オプションをオフにする必要があります。ASP.NET で fast オプションをオフにするのは安全ではありません。スレッドに関する問題が発生する場合があります。

使用例

次のコードは、**arguments** オブジェクトの使用例です。

```
function argTest(a, b) : String {
   var i : int;
   var s : String = "The argTest function expected ";
   var numargs : int = arguments.length; // Get number of arguments passed.
   var expargs : int = argTest.length;   // Get number of arguments expected.
   if (expargs < 2)
      s += expargs + " argument. ";
   else
      s += expargs + " arguments. ";
   if (numargs < 2)
      s += numargs + " was passed.";
   else
      s += numargs + " were passed.";
   s += "\n"
   for (i =0 ; i < numargs; i++){        // Get argument contents.
      s += "  Arg " + i + " = " + arguments[i] + "\n";
   }
   return(s);                            // Return list of arguments.
}

print(argTest(42));
print(argTest(new Date(1999,8,7),"Sam",Math.PI));
```

このプログラムの出力は次のようになります。

```
The argTest function expected 2 arguments. 1 was passed.
  Arg 0 = 42
```

```
The argTest function expected 2 arguments. 3 were passed.
  Arg 0 = Tue Sep 7 00:00:00 PDT 1999
  Arg 1 = Sam
  Arg 2 = 3.141592653589793
```

参照

関連項目

new 演算子

function ステートメント

/fast

arguments オブジェクトのプロパティ

arguments オブジェクトは、現在実行中の関数の引数、およびその関数の呼び出し側の関数に渡される引数を表します。

プロパティ

0...n プロパティ

arguments プロパティ

callee プロパティ

caller プロパティ

length プロパティ (arguments)

参照

その他の技術情報

プロパティ (JScript)

JScript リファレンス

Array オブジェクト

すべてのデータ型の expando 配列をサポートする機能を提供します。**Array** コンストラクタには、次の 3 つの形式があります。:

```
function Array( [size : int] )
function Array( [... varargs : Object[]] )
function Array( [array : System.Array )
```

引数

size

省略可能です。配列のサイズ。JScript の配列のインデックスは 0 から始まります。したがって、作成された要素のインデックスは 0 ～ (*size* - 1) となります。

varargs

省略可能です。コンストラクタに渡されたすべてのパラメータを保持する、型指定された配列。これらのパラメータは、配列の最初の要素として使用されます。

array

省略可能です。構築される配列にコピーされる配列。

解説

Array オブジェクトのコンストラクタに渡される引数が 1 つだけで、その引数が数値である場合、その値は符号なしの 32 ビット整数 (約 40 億未満の任意の整数) であることが必要です。渡された値は、配列のサイズに使用されます。その値が 0 より小さいか、整数でない場合は、ランタイム エラーが発生します。

データ型 **System.Array** の変数は、**Array** コンストラクタに渡すことができます。これにより、入力配列のコピーとなる JScript 配列が生成されます。**System.Array** は、1 次元であることが必要です。

Array オブジェクトのコンストラクタに渡される値が 1 つだけで、それが数値または配列でない場合、配列の **length** プロパティは 1 に設定され、配列の最初の要素 (要素 0) の値は渡された引数の値になります。複数の引数がコンストラクタに渡される場合は、配列の長さが引数の数に設定され、それらの引数が新規配列の最初の要素になります。

Script の配列は疎配列であり、1 つの配列に多数の要素を割り当てても、実際にデータを含む要素だけが存在するしくみになっています。このため、配列が使用するメモリ量を抑えることができます。

Array オブジェクトは **System.Array** データ型と相互運用されます。したがって、**Array** オブジェクトは **System.Array** データ型のメソッドとプロパティを呼び出すことができ、**System.Array** データ型は **Array** オブジェクトのメソッドとプロパティを呼び出すことができます。また、**Array** オブジェクトは **System.Array** 型を受け取る関数で使用でき、その逆も可能です。詳細については、「Array メンバ」を参照してください。

Array オブジェクトが **System.Array** を受け取る関数に渡されるとき、または **System.Array** メソッドが **Array** オブジェクトから呼び出されるときに、**Array** の内容がコピーされます。したがって、**System.Array** メソッドを呼び出したり、このメソッドを **System.Array** を受け取る関数に渡すことで、元の **Array** オブジェクトを変更することはできません。**System.Array** では、非破棄的な **Array** メソッドだけを呼び出せます。

ヒント

Array オブジェクトは、ジェネリック スタックや項目の一覧が必要な場合で、パフォーマンスが最優先でない場合に便利です。それ以外の場合は、型指定された配列データ型を使用してください。型指定された配列は、**Array** オブジェクトとほとんど同じ機能を持ち、タイプ セーフです。この配列を使用すると、パフォーマンスが向上し、他の言語とやり取りが簡単になります。

メモ :

JScript 内では、**Array** オブジェクトは .NET Framework の **System.Array** データ型と相互運用されます。ただし、**Array** オブジェクトがサポートされるのは JScript だけです。このオブジェクトは .NET Framework 型から派生していないため、他の共通言語仕様 (CLS: Common Language Specification) 言語では使用できません。したがって、CLS 準拠のメソッドのパラメータと戻り値の型を型の注釈で指定する場合は、**Array** オブジェクトではなく **System.Array** 型を使用してください。ただし、パラメータや戻り値の型以外の識別子では、**Array** オブジェクトを使用して型の注釈を指定できます。詳細については、「CLS 準拠コードの記述」を参照してください。

使用例

配列の各要素には、[ ] 表記を使用してアクセスできます。次に例を示します。

```
var my_array = new Array();
for (var i = 0; i < 10; i++) {
   my_array[i] = i;
}
var x = my_array[4];
```

Microsoft JScript では、配列のインデックスは 0 から始まります。したがって、上の例の最後のステートメントでは、配列内の 5 つ目の要素を取得しています。その要素の値は 4 です。

プロパティおよびメソッド

Array オブジェクトのプロパティとメソッド

必要条件

Version 2

参照

関連項目

new 演算子

概念

型指定された配列

Array オブジェクトのプロパティとメソッド

Array オブジェクトは、任意のデータ型の配列を作成する機能を提供します。

プロパティ

constructor プロパティ

length プロパティ (Array)

prototype プロパティ

メソッド

concat メソッド (Array)

join メソッド

pop メソッド

push メソッド

reverse メソッド

shift メソッド

slice メソッド (Array)

sort メソッド

splice メソッド

toLocaleString メソッド

toString メソッド

unshift メソッド

valueOf メソッド

参照

その他の技術情報

プロパティ (JScript)
メソッド
JScript リファレンス

Boolean オブジェクト

Boolean オブジェクトはブール値を参照します。

```
function Boolean( [boolValue : boolean] )
```

引数
boolValue

省略可能です。新しいオブジェクトの初期ブール値。*boolValue* を省略するか、**false**、0、**null**、**NaN**、および長さ 0 の文字列 ("") のいずれかを指定すると、Boolean オブジェクトの初期値は **false** になります。それ以外の場合、初期値は **true** になります。

解説

Boolean オブジェクトは、Boolean 型のラッパーです。**Boolean** オブジェクトを使用する主な目的は、数値の各種プロパティを 1 つのオブジェクトに集めておくこと、および **toString** メソッドを使用して Boolean 値を文字列に変換できるようにすることです。**Boolean** オブジェクトは **boolean** 型に似ています。ただし、プロパティとメソッドが異なります。

メモ：

ほとんどの場合、Boolean オブジェクトを明示的に作成する必要はありません。通常は boolean 型を使用してください。**Boolean** オブジェクトは boolean 型と相互運用されるため、すべての Boolean オブジェクトのメソッドとプロパティは、Boolean 型の変数に対しても使用できます。詳細については、「boolean 型」を参照してください。

Boolean オブジェクトのデータ型は **Object** で、**boolean** ではありません。

プロパティおよびメソッド

Boolean オブジェクトのプロパティとメソッド

必要条件

Version 2

参照
関連項目
Object オブジェクト
Boolean Structure
new 演算子
var ステートメント

Boolean オブジェクトのプロパティとメソッド

Boolean オブジェクトは、新しいブール値を作成します。

プロパティ

constructor プロパティ

prototype プロパティ

メソッド

toString メソッド

valueOf メソッド

参照

その他の技術情報

プロパティ (JScript)
メソッド
JScript リファレンス

Date オブジェクト

このオブジェクトを使用して、日付と時刻を格納したり、取得したりできます。**Date** コンストラクタには、次の 2 つの形式があります。

```
function Date( [dateVal : { Number | String | System.DateTime } ] )
function Date( year : int, month : int, date : int[, hours : int [, minutes : int [, second
s : int [, ms : int]]]] )
```

引数

dateVal

省略可能です。数値で指定する場合は、指定した日付と 1970 年 1 月 1 日 0 時 0 分 0 秒との間を、ミリ秒単位の数値を使って世界協定時刻 (UTC) で表します。文字列で指定する場合は、*dateVal* は **parse** メソッドでの規則に準じて解析されます。*dateVal* は、.NET データ値でも指定できます。

year

必ず指定します。4 桁の年 (76 ではなく 1976 など)。

month

必ず指定します。月を表す 0 ～ 11 (1 ～ 12 月に相当) の範囲内の整数を指定します。

date

必ず指定します。日を表す 1 ～ 31 の範囲内の整数を指定します。

hours

省略可能です。引数 *minutes* を指定する場合は、この引数を指定する必要があります。時を表す 0 ～ 23 (午前 0 時 ～ 午後 11 時に対応) の範囲内の整数を指定します。

minutes

省略可能です。引数 *seconds* を指定する場合は、この引数を指定する必要があります。分を表す 0 ～ 59 の範囲内の整数を指定します。

seconds

省略可能です。引数 *milliseconds* を指定する場合は、この引数を指定する必要があります。秒を表す 0 ～ 59 の範囲内の整数を指定します。

ms

省略可能です。ミリ秒を表す 0 ～ 999 の範囲内の整数を指定します。

解説

Date オブジェクトには、特定の時刻をミリ秒で表す数値が格納されます。引数に有効範囲を超える値や負の値を指定すると、値に応じて格納される他の値が変更されます。たとえば、150 秒を指定すると、2 分 30 秒として処理されます。

格納されている数値が **NaN** の場合は、オブジェクトが特定の時刻を表していないことを示します。**Date** コンストラクタに渡すパラメータがない場合は、現在の時刻 (UTC) で初期化されます。**Date** 型の変数は、使用する前に初期化する必要があります。

Date オブジェクトで表せる日付の範囲は、1970 年 1 月 1 日の前後の約 285,616 年です。

Date オブジェクトには、**Date** オブジェクトを作成しなくても呼び出すことができる、**parse** と **UTC** の 2 つの静的なメソッドがあります。

Date コンストラクタが **new** 演算子を使用せずに呼び出されると、コンストラクタに渡される引数にかかわらず、返される **Date** オブジェクトには現在の日付が含まれます。

メモ :

JScript 内では、**Date** オブジェクトは .NET Framework の **System.DateTime** データ型と相互運用されます。ただし、**Date** オブジェクトがサポートされるのは JScript だけです。このオブジェクトは .NET Framework 型から派生していないため、他の共通言語仕様 (CLS: Common Language Specification) 言語では使用できません。したがって、CLS 準拠のメソッドのパラメータと戻り値の型を型の注釈で指定する場合は、**Date** オブジェクトではなく **System.DateTime** 型を使用してください。ただし、パラメータや戻り値の型以外の識別子では、**Date** オブジェクトを使用して型の注釈を指定できます。詳細については、「CLS 準拠コードの記述」を参照してください。

使用例

次の例では、**Date** オブジェクトを使用しています。

```
var s : String = "Today's date is: ";   // Declare variables.
var d : Date = new Date();              // Create Date object with today's date.
s += (d.getMonth() + 1) + "/";          // Get month
s += d.getDate() + "/";                 // Get day
s += d.getYear();                       // Get year.
print(s);                               // Print date.
```

このプログラムを 1992 年 1 月 26 日に実行すると、出力は次のようになります。

```
Today's date is: 1/26/1992
```

プロパティおよびメソッド

Date オブジェクトのプロパティとメソッド

必要条件

Version 1

参照

関連項目

new 演算子
var ステートメント

JScript 8.0

Date オブジェクトのプロパティとメソッド

Date オブジェクトを使用して、日付と時刻を格納したり、取得したりできます。

プロパティ

constructor プロパティ

prototype プロパティ

メソッド

getDate メソッド

getDay メソッド

getFullYear メソッド

getHours メソッド

getMilliseconds メソッド

getMinutes メソッド

getMonth メソッド

getSeconds メソッド

getTime メソッド

getTimezoneOffset メソッド

getUTCDate メソッド

getUTCDay メソッド

getUTCFullYear メソッド

getUTCHours メソッド

getUTCMilliseconds メソッド

getUTCMinutes メソッド

getUTCMonth メソッド

getUTCSeconds メソッド

getVarDate メソッド

getYear メソッド

parse メソッド

setDate メソッド

setFullYear メソッド

setHours メソッド

setMilliseconds メソッド

setMinutes メソッド

setMonth メソッド

setSeconds メソッド

setTime メソッド

setUTCDate メソッド

setUTCFullYear メソッド

setUTCHours メソッド

setUTCMilliseconds メソッド

setUTCMinutes メソッド

setUTCMonth メソッド

setUTCSeconds メソッド

setYear メソッド

toDateString メソッド

toGMTString メソッド

toLocaleDateString メソッド

toLocaleString メソッド

toLocaleTimeString メソッド

toString メソッド

toTimeString メソッド

toUTCString メソッド

UTC メソッド

valueOf メソッド

参照

その他の技術情報

プロパティ (JScript)
メソッド
JScript リファレンス

Enumerator オブジェクト

コレクション内の項目を列挙する手段を提供します。

```
varName = new Enumerator([collection])
```

引数
varName

必ず指定します。列挙子を代入する変数名。

collection

省略可能です。配列やコレクションなど、**IEnumerable** インターフェイスを実装するオブジェクト。

解説

コレクションは JScript で自動的に列挙できます。したがって、コレクションのメンバにアクセスするために、**Enumerator** オブジェクトを使用する必要はありません。**for...in** ステートメントを使用して、メンバに直接アクセスできます。**Enumerator** オブジェクトは下位互換性を維持するために用意されています。

コレクションは、そのメンバに直接アクセスできないという点で配列とは異なります。配列の場合はインデックスを使って項目にアクセスできますが、コレクションでは、現在の項目を指すポインタをコレクション内の最初の項目に移動したり、次の項目に移動したりするしかありません。

Enumerator オブジェクトは、コレクション内の任意のメンバにアクセスする手段を提供するオブジェクトで、VBScript の **For...Each** ステートメントと同じように動作します。

IEnumerable を実装するクラスを定義して、JScript でコレクションを作成できます。コレクションは、他の言語 (Visual Basic など) や **ActiveXObject** オブジェクトを使用しても作成できます。

例 1

次のコードは **Enumerator** オブジェクトを使用して、使用可能なドライブ文字と、名前が使用できる場合はその名前を出力します。

```
// Declare variables.
var n, x;
var fso : ActiveXObject = new ActiveXObject("Scripting.FileSystemObject");
// Create Enumerator on Drives.
var e : Enumerator = new Enumerator(fso.Drives);
for (;!e.atEnd();e.moveNext()) {      // Loop over the drives collection.
   x = e.item();
   if (x.DriveType == 3)              // See if network drive.
      n = x.ShareName;                // Get share name
   else if (x.IsReady)                // See if drive is ready.
      n = x.VolumeName;               // Get volume name.
   else
      n = "[Drive not ready]";
   print(x.DriveLetter + " - " + n);
}
```

システムによって異なりますが、出力は次のようになります。

```
A - [Drive not ready]
C - DRV1
D - BACKUP
E - [Drive not ready]
```

例 2

例 1 のコードは、**Enumerator** オブジェクトを使用せずに、次のように記述することもできます。この場合、列挙体のメンバには直接アクセスできます。

```
// Declare variables.
```

```
var n, x;
var fso : ActiveXObject = new ActiveXObject("Scripting.FileSystemObject");
// The following three lines are not needed.
//    var e : Enumerator = new Enumerator(fso.Drives);
//    for (;!e.atEnd();e.moveNext()) {
//       x = e.item();
// Access the members of the enumeration directly.
for (x in fso.Drives) {               // Loop over the drives collection.
   if (x.DriveType == 3)              // See if network drive.
      n = x.ShareName;                // Get share name
   else if (x.IsReady)                // See if drive is ready.
      n = x.VolumeName;               // Get volume name.
   else
      n = "[Drive not ready]";
   print(x.DriveLetter + " - " + n);
}
```

プロパティ

Enumerator オブジェクトには、プロパティはありません。

メソッド

Enumerator オブジェクトのメソッド

必要条件

Version 3

参照

関連項目

new 演算子
for...in ステートメント

Enumerator オブジェクトのメソッド

Enumerator オブジェクトを使用して、コレクション内の項目を列挙できます。

メソッド

atEnd メソッド

item メソッド

moveFirst メソッド

moveNext メソッド

参照

その他の技術情報

メソッド

JScript リファレンス

Error オブジェクト

エラーに関する情報を格納します。**Error** コンストラクタには、次の 2 つの形式があります。

```
function Error([description : String ])
function Error([number : Number [, description : String ]])
```

引数
number

省略可能です。**number** プロパティの値を指定する、エラーに割り当てられた数値。省略した場合は 0 です。

description

省略可能です。エラーを説明する短い文字列。**description** プロパティおよび **message** プロパティの初期値になります。省略した場合は空の文字列です。

解説

Error オブジェクトは、上に示したコンストラクタを使用して明示的に作成できます。**Error** オブジェクトにプロパティを追加して、その機能を拡張できます。ランタイム エラーが発生した場合はいつでも、**Error** オブジェクトを作成してエラーを表すことができます。

一般に、**Error** オブジェクトは **throw** ステートメントでスローされ、**try...catch** ステートメントでキャッチされます。**throw** ステートメントを使用して任意のデータ型をエラーとして渡すことができますが、**throw** ステートメントは暗黙的に **Error** オブジェクトを作成しません。**Error** オブジェクトをスローすることにより、**catch** ブロックが JScript のランタイム エラーおよびユーザー定義エラーを同様に処理できます。

Error オブジェクトには 4 つの組み込みプロパティがあります。エラーの説明 (**description** および **message** プロパティ)、エラー番号 (**number** プロパティ)、およびエラーの名前 (**name** プロパティ) の 4 つです。**description** プロパティおよび **message** プロパティは、同じメッセージを参照します。**description** プロパティは下位互換性を提供し、**message** プロパティは ECMA 規格に準拠しています。

エラー番号は 32 ビット値です。上位の 16 ビット ワードは機能識別符号です。下位のワードは実際のエラー コードです。実際のエラー コードを読み取るには、**&** (ビットごとの And) 演算子を使用して、number プロパティと 16 進数の `0xFFFF` を組み合わせます。

注意

ASP.NET ページで JScript の **Error** オブジェクトを使用した場合、予期しない結果が生じることがあります。これは、JScript の **Error** オブジェクトと ASP.NET ページの **Error** イベントの間にあるあいまいさが原因です。ASP.NET ページのエラー処理には、**Error** オブジェクトではなく **System.Exception** クラスを使用してください。

メモ :

Error オブジェクトがサポートされるのは JScript だけです。このオブジェクトは .NET Framework 型から派生していないため、他の共通言語仕様 (CLS: Common Language Specification) 言語では使用できません。したがって、CLS 準拠のメソッドのパラメータと戻り値の型を型の注釈で指定する場合は、**Error** オブジェクトではなく **System.Exception** 型を使用してください。ただし、パラメータや戻り値の型以外の識別子では、**Error** オブジェクトを使用して型の注釈を指定できます。詳細については、「CLS 準拠コードの記述」を参照してください。

使用例

次のコードは、**Error** オブジェクトの使用例です。

```
try {
   // Throw an error.
   throw new Error(42,"No question");
} catch(e) {
   print(e)
// Extract the error code from the error number.
   print(e.number & 0xFFFF)
   print(e.description)
}
```

このコードの出力は次のようになります。

```
Error: No question
42
No question
```

プロパティおよびメソッド

Error オブジェクトのプロパティとメソッド

必要条件

Version 5

参照

関連項目

new 演算子
throw ステートメント
try...catch...finally ステートメント
var ステートメント
AllMembers.T:System.Web.UI.Page

Error オブジェクトのプロパティとメソッド

Error オブジェクトには、エラーに関する情報が含まれます。

プロパティ

description プロパティ

message プロパティ

name プロパティ

number プロパティ

メソッド

toString メソッド

参照

その他の技術情報

プロパティ (JScript)
メソッド
JScript リファレンス

Function オブジェクト

新しい関数を作成します。

```
function Function( [[param1 : String, [..., paramN : String,]] body : String ])
```

引数
param1, ..., paramN

省略可能です。関数のパラメータ。各パラメータには、型の注釈を指定できます。最後のパラメータは *parameterarray* となる場合があります。この配列は、3 つのピリオド (**...**)、パラメータ配列名、および型指定された配列の型の注釈で表されます。

body

省略可能です。関数が呼び出されたときに実行する JScript コード ブロックを記述した文字列。

解説

Function コンストラクタを使用すると、実行時にスクリプトで関数を作成できます。**Function** コンストラクタに渡されるパラメータは、最後のパラメータを除いて、新しい関数のパラメータとして使用されます。コンストラクタに渡される最後のパラメータは、関数本体のコードとして解釈されます。

JScript は、**Function** コンストラクタが呼びされたときに、コンストラクタで作成されるオブジェクトをコンパイルします。したがって、実行時に関数を柔軟に再定義できますが、コードの実行速度は低下します。スクリプトの速度低下を避けるには、**Function** コンストラクタをできるだけ使用しないようにしてください。

評価する関数を呼び出すときには、必要な引数とかっこを記述してください。かっこを指定せずに関数を呼び出すと、その関数の **Function** オブジェクトが返されます。関数のテキストは、**Function** オブジェクトの **toString** メソッドを使用して取得できます。

メモ :

Function オブジェクトがサポートされるのは JScript だけです。このオブジェクトは .NET Framework 型から派生していないため、他の共通言語仕様 (CLS: Common Language Specification) 言語では使用できません。したがって、CLS 準拠のメソッドのパラメータと戻り値の型を型の注釈で指定する場合は、**Function** オブジェクトではなく **System.EventHandler** 型を使用してください。ただし、パラメータや戻り値の型以外の識別子では、**Function** オブジェクトを使用して型の注釈を指定できます。詳細については、「CLS 準拠コードの記述」を参照してください。

使用例

次のコードは、**Function** オブジェクトの使用例です。

```
var add : Function = new Function("x", "y", "return(x+y)");
print(add(2, 3));
```

このコードの出力は、次のようになります。

```
5
```

プロパティおよびメソッド

Function オブジェクトのプロパティとメソッド

必要条件

Version 2

参照
関連項目
function ステートメント
new 演算子
var ステートメント

Function オブジェクトのプロパティとメソッド

Function オブジェクトは、新しい関数を作成します。

プロパティ

0...n プロパティ

arguments プロパティ

callee プロパティ

caller プロパティ

constructor プロパティ

length プロパティ (Function)

prototype プロパティ

メソッド

apply メソッド

call メソッド

toString メソッド

valueOf メソッド

参照

その他の技術情報

プロパティ (JScript)
メソッド
JScript リファレンス

Global オブジェクト

グローバルなメソッドを 1 つのオブジェクトに集めておくための組み込みオブジェクトです。

Global オブジェクトには、メソッドはありません。直接、メソッドを呼び出します。

プロパティおよびメソッド

Global オブジェクトのプロパティとメソッド

必要条件

Version 5

解説

Global オブジェクトは、直接使用されることはなく、**new** 演算子では作成できません。このオブジェクトは、スクリプト エンジンが初期化されるときに作成され、そのメソッドとプロパティはその時点からすぐに使用可能になります。

参照

関連項目

Object オブジェクト

Global オブジェクトのプロパティとメソッド

Global オブジェクトは、グローバルなメソッドを 1 つのオブジェクトに集めておくための、組み込みオブジェクトです。

プロパティ

Infinity プロパティ

NaN プロパティ (Global)

undefined プロパティ

メソッド

decodeURI メソッド

decodeURIComponent メソッド

encodeURI メソッド

encodeURIComponent メソッド

escape メソッド

eval メソッド

isFinite メソッド

isNaN メソッド

parseFloat メソッド

parseInt メソッド

unescape メソッド

参照

その他の技術情報

プロパティ (JScript)
メソッド
JScript リファレンス

Math オブジェクト

数値計算のための基本的な演算機能と定数を提供する組み込みオブジェクトです。このオブジェクトを明示的に構築することはできません。

プロパティおよびメソッド

Math オブジェクトのプロパティとメソッド

必要条件

Version 1

解説

Math オブジェクトは、**new** 演算子では作成できません。**new** 演算子を使って作成しようとすると、エラーが返されます。スクリプト エンジンは、エンジンが読み込まれるときに **Math** オブジェクトを作成します。Math オブジェクトのメソッドやプロパティはすべて、スクリプト内でいつでも使用できます。

次のコードは、**Math** オブジェクトの使用例です。浮動小数点数の精度には限界があるため、これらを使用する計算では、微小な丸め誤差が蓄積する可能性があります。**Number** オブジェクトの **toFixed** メソッドを使用すると、丸め誤差のない数値を表示できます。

使用例

```
var pi : double = Math.PI;         // Should be about 3.14.
print(pi);
var cosPi : double = Math.cos(pi); // Should be minus one.
print(cosPi);
var sinPi : double = Math.sin(pi); // Should be zero.
print(sinPi.toFixed(10));
```

このコードの出力は次のようになります。

```
3.141592653589793
-1
0.0000000000
```

参照

関連項目

Number オブジェクト

Math オブジェクトのプロパティとメソッド

Math オブジェクトは、数値計算のための基本的な演算機能と定数を提供する組み込みオブジェクトです。

プロパティ

E プロパティ

LN10 プロパティ

LN2 プロパティ

LOG10E プロパティ

LOG2E プロパティ

PI プロパティ

SQRT1_2 プロパティ

SQRT2 プロパティ

メソッド

abs メソッド

acos メソッド

asin メソッド

atan メソッド

atan2 メソッド

ceil メソッド

cos メソッド

exp メソッド

floor メソッド

log メソッド

max メソッド

min メソッド

pow メソッド

random メソッド

round メソッド

sin メソッド

sqrt メソッド

tan メソッド

参照

その他の技術情報

プロパティ (JScript)
メソッド
JScript リファレンス

Number オブジェクト

数値データを表すオブジェクトです。数値定数のプレースホルダでもあります。

```
function Number( [value : Number] )
```

引数
value

必ず指定します。作成する **Number** オブジェクトの数値。

解説

Number オブジェクトは、数値データのラッパーです。**Number** オブジェクトを使用する主な目的は、数値の各種プロパティを 1 つのオブジェクトに集めておくこと、および **toString** メソッドを使用して数値を文字列へ変換できるようにすることです。**Number** オブジェクトは **Number** 型に似ています。ただし、プロパティとメソッドが異なります。

メモ：

ほとんどの場合、**Number** オブジェクトを明示的に作成する必要はありません。通常は **Number** 型を使用してください。**Number** オブジェクトは **Number** 型と相互運用されるため、**Number** オブジェクトのメソッドとプロパティはすべて、**Number** 型の変数に対しても使用できます。詳細については、「Number 型」を参照してください。

Number オブジェクトは、数値データを 8 バイトの倍精度浮動小数点数として保持します。倍精度の 64 ビット IEEE 754 値を表します。**Number** オブジェクトは、-1.79769313486231570E+308 ～ +1.79769313486231570E+308 の範囲の数値を表すことができます。表すことのできる最小の数値は 4.94065645841247E-324 です。**Number** オブジェクトは非数 (**NaN**)、正の無限大、負の無限大、正の 0、および負の 0 を表すことができます。

Number オブジェクトのデータ型は **Object** で、**Number** ではありません。

プロパティおよびメソッド

Number オブジェクトのプロパティとメソッド

必要条件

Version 1

参照
関連項目
Object オブジェクト
Number 型
Math オブジェクト
new 演算子

Number オブジェクトのプロパティとメソッド

Number オブジェクトは、数値型のデータを表すオブジェクトです。数値定数のプレースホルダでもあります。

プロパティ

constructor プロパティ

MAX_VALUE プロパティ

MIN_VALUE プロパティ

NaN プロパティ

NEGATIVE_INFINITY プロパティ

POSITIVE_INFINITY プロパティ

prototype プロパティ

メソッド

toExponential メソッド

toFixed メソッド

toLocaleString メソッド

toPrecision メソッド

toString メソッド

valueOf メソッド

参照

その他の技術情報

プロパティ (JScript)
メソッド
JScript リファレンス

Object オブジェクト

すべての JScript オブジェクトに共通の機能を提供します。

```
function Object([value : { ActiveXObject | Array | Boolean | Date | Enumerator | Error | Fu
nction | Number | Object | RegExp | String | VBArray ])
```

引数
value

省略可能です。JScript の任意のプリミティブ データ型。オブジェクトの場合、そのオブジェクトは変更されずに返されます。**null**、**undefined**、または省略した場合は、何も入っていないオブジェクトが作成されます。

解説

Object オブジェクトは、JScript のその他のオブジェクトすべての基礎となります。このオブジェクトのメソッドとプロパティはいずれも、その他のすべてのオブジェクトで使用できます。メソッドはユーザー定義オブジェクト内で再定義でき、必要なときに JScript から呼び出されます。たとえば、**toString** メソッドは、再定義されることの多い **Object** メソッドの 1 つです。

型の注釈を指定せずに定義した変数は、暗黙的に **Object** 型になります。JScript オブジェクトはそれぞれ、独自のプロパティとメソッドに加えて、**Object** オブジェクトのすべてのプロパティとメソッドを持ちます。

プロパティおよびメソッド

Object オブジェクトのプロパティとメソッド

必要条件

Version 3

参照
関連項目
new 演算子
Function オブジェクト
Global オブジェクト

Object オブジェクトのプロパティとメソッド

Object オブジェクトは、すべての JScript オブジェクトに共通の機能を提供します。

プロパティ

constructor プロパティ

prototype プロパティ

propertyIsEnumerable プロパティ

メソッド

isPrototypeOf メソッド

hasOwnProperty メソッド

toLocaleString メソッド

toString メソッド

valueOf メソッド

参照

その他の技術情報

プロパティ (JScript)
メソッド
JScript リファレンス

RegExp オブジェクト

正規表現パターン一致の結果についての情報を保存する組み込みのグローバルなオブジェクトです。このオブジェクトを明示的に構築することはできません。

プロパティ

RegExp オブジェクトのプロパティ

メソッド

RegExp オブジェクトには、メソッドはありません。

必要条件

Version 3

解説

RegExp オブジェクトは、直接作成することはできませんが、いつでも使用できます。次の表は、正規表現の検索が成功するまでの、**RegExp** オブジェクトのさまざまなプロパティの初期値を示したものです。

プロパティ	略式	初期値
index		-1
input	$_	空の文字列
lastIndex		-1
lastMatch	$&	空の文字列
lastParen	$+	空の文字列
leftContext	$`	空の文字列
rightContext	$'	空の文字列
$1 - $9		空の文字列

グローバルな **RegExp** オブジェクトを **Regular Expression** オブジェクトと混同しないようにしてください。名前は似ていますが、これらの 2 つのオブジェクトには明確な違いがあります。グローバルな **RegExp** オブジェクトのプロパティには、一致が検出されるたびに更新される情報が格納されるのに対し、**Regular Expression** オブジェクトのプロパティには、**Regular Expression** の 1 つのインスタンスによる一致に関する情報だけが格納されます。

メモ：

RegExp のプロパティは、高速モードで実行されている場合は利用できません。JScript の既定のモードは高速モードです。これらのプロパティを使用するプログラムをコマンド ラインからコンパイルするには、**/fast-** を使用して fast オプションをオフにする必要があります。ASP.NET で fast オプションをオフにするのは安全ではありません。スレッドに関する問題が発生する場合があります。

次のコードは、グローバルな **RegExp** オブジェクトの使用例です。この例は、**/fast-** オプションを使用してコンパイルする必要があります。

使用例

```
var re : RegExp = new RegExp("d(b+)(d)","ig");
var arr : Array = re.exec("cdbBdbsbdbdz");
print("$1 contains: " + RegExp.$1);
print("$2 contains: " + RegExp.$2);
print("$3 contains: " + RegExp.$3);
```

このコードの出力は次のようになります。

```
$1 contains: bB
$2 contains: d
$3 contains:
```

参照

関連項目

Regular Expression オブジェクト

String オブジェクト

/fast

概念

正規表現の構文

RegExp オブジェクトのプロパティ

RegExp オブジェクトは、正規表現パターン一致の結果情報を保存する、組み込みのグローバル オブジェクトです。

プロパティ

$1...$9 プロパティ

index プロパティ

input プロパティ ($_)

lastIndex プロパティ

lastMatch プロパティ ($&)

lastParen プロパティ ($+)

leftContext プロパティ ($`)

rightContext プロパティ ($')

参照

その他の技術情報

プロパティ (JScript)
JScript リファレンス

Regular Expression オブジェクト

正規表現パターンとそのパターンの適用方法を識別するフラグを含むオブジェクトです。

```
// The explicit constructor for a Regular Expression object.
function RegExp(pattern : String [,flags : String])
function RegExp(regexObj : System.Text.RegularExpressions.Regex)

// The implicit constructor for a Regular Expression object.
/pattern/[flags]
```

引数

pattern

必ず指定します。使用する正規表現パターン。構文 1 では、パターンは文字列である必要があります。構文 2 では、パターンを "/" 文字で区切ります。

flags

省略可能です。構文 1 では、フラグは文字列で指定する必要があります。構文 2 では、フラグ文字は最後の "/" のすぐ後に指定する必要があります。指定できるフラグは、次のとおりです。

- g (引数 *pattern* に指定したパターンと一致する文字列をすべて検索するグローバル検索)
- i (大文字小文字を区別しない)
- m (複数行検索)

regexObj

必ず指定します。使用する正規表現パターンを含む **Regex** オブジェクト。

解説

Regular Expression オブジェクトを **RegExp** オブジェクトと混同しないようにしてください。名前は似ていますが、これらの 2 つのオブジェクトには明確な違いがあります。**Regular Expression** オブジェクトのプロパティには、特定の Regular Expression のインスタンスによる一致に関する情報だけが格納されるのに対し、グローバルな **RegExp** オブジェクトのプロパティには一致が検出されるたびに更新される情報が格納されます。

Regular Expression オブジェクトは、文字を組み合わせた文字列検索 (正規表現による検索) に使用するパターンを格納します。Regular Expression オブジェクトが作成されると、このオブジェクトが文字列のメソッドに渡されるか、または文字列が Regular Expression オブジェクトのいずれかのメソッドに渡されます。最後に実行した検索に関する情報が、グローバルな **RegExp** オブジェクトに格納されます。

ユーザー入力から文字列を取得する場合のように、検索文字列が頻繁に変更されたり、コーディングの時点では検索文字列がわからないときには、構文 1 を使用します。検索文字列があらかじめわかっている場合は、構文 2 を使用します。

JScript では、引数 *pattern* に指定したパターンは、使用前に内部形式にコンパイルされます。構文 1 の場合は、使用する直前、または **compile** メソッドが呼び出されたときにパターンがコンパイルされます。構文 2 の場合は、スクリプトが読み込まれたときにパターンがコンパイルされます。

メモ :

JScript 内では、Regular Expression オブジェクトは .NET Framework の **System.Text.RegularExpressions.Regex** データ型と相互運用されます。ただし、Regular Expression オブジェクトがサポートされるのは JScript だけです。このオブジェクトは .NET Framework 型から派生していないため、他の共通言語仕様 (CLS: Common Language Specification) 言語では使用できません。したがって、CLS 準拠のメソッドのパラメータと戻り値の型を型の注釈で指定する場合は、**Regular Expression** オブジェクトではなく **System.Text.RegularExpressions.Regex** 型を使用してください。ただし、パラメータや戻り値の型以外の識別子では、**Regular Expression** オブジェクトを使用して型の注釈を指定できます。詳細については、「CLS 準拠コードの記述」を参照してください。

使用例

次のコードは、**Regular Expression** オブジェクトの使用例です。関連するフラグを持つ正規表現パターンを含むオブジェクト `re1` と `re2` が作成されます。この場合、**Regular Expression** オブジェクトの結果は **match** メソッドによって使用されます。

```
var s : String = "The rain in Spain falls mainly in the plain";
// Create regular expression object using Syntax 1.
var re1 : RegExp = new RegExp("Spain","i");
// Create regular expression object using Syntax 2.
var re2 : RegExp = /IN/i;

// Find a match within string s.
print(s.match(re1));
print(s.match(re2));
```

このスクリプトの出力は次のようになります。

```
Spain
in
```

必要条件

Version 3

プロパティおよびメソッド

Regular Expression オブジェクトのプロパティとメソッド

参照

関連項目

new 演算子
RegExp オブジェクト
String オブジェクト
Regex

概念

正規表現の構文

Regular Expression オブジェクトのプロパティとメソッド

Regular Expression オブジェクトは、正規表現パターンと、パターンの適用方法を識別するフラグを含むオブジェクトです。

プロパティ

global プロパティ

ignoreCase プロパティ

multiline プロパティ

source プロパティ

メソッド

compile メソッド

exec メソッド

test メソッド

参照

その他の技術情報

プロパティ (JScript)
メソッド
JScript リファレンス

String オブジェクト

このオブジェクトを使用すると、文字列の操作と書式設定を実行できます。また、文字列内の一部分を取得したり、文字列内で指定した文字列を検索できます。

```
function String([stringLiteral : String])
```

引数
stringLiteral

省略可能です。任意の Unicode 文字グループ。

解説

String オブジェクトは、リテラル文字列を使用して自動的に作成できます。この方法で作成された **String** オブジェクト ("プリミティブ" 文字列) は、**new** 演算子で作成された **String** オブジェクトとは使い方が異なります。プリミティブ文字列では、プロパティの読み取りやメソッドの呼び出しは実行できますが、新しいプロパティの作成やメソッドの追加は実行できません。

リテラル文字列にエスケープ シーケンスを使用すると、改行文字や Unicode 文字など、文字列で直接使用できない特殊文字を表すことができます。スクリプトのコンパイル時に、リテラル文字列の各エスケープ シーケンスは、それぞれが表している文字に変換されます。詳細については、「文字列データ」を参照してください。

JScript では、**String** 型も定義されています。このデータ型は、**String** オブジェクトとは異なるプロパティとメソッドを提供します。**String** 型の変数にプロパティを作成したり、メソッドを追加したりすることはできません。**String** オブジェクトのインスタンスでは、これらの処理が可能です。

String オブジェクトは、**String** 型 (**System.String** 型と同じ) と相互運用されます。つまり、**String** オブジェクトは **String** 型のメソッドとプロパティを呼び出すことができ、**String** 型は **String** オブジェクトのメソッドとプロパティを呼び出すことができます。詳細については、「AllMembers.T:System.String」を参照してください。また、**String** オブジェクトは **String** 型を受け取る関数で使用でき、その逆も可能です。

String オブジェクトのデータ型は **Object** で、**String** ではありません。

例 1

このスクリプトでは、length プロパティは読み取り可能で、**toUpperCase** メソッドは呼び出し可能ですが、カスタム プロパティの `myProperty` にはプリミティブ文字列を設定できないことを示しています。

```
var primStr : Object = "This is a string";
print(primStr.length);           // Read the length property.
print(primStr.toUpperCase());    // Call a method.
primStr.myProperty = 42;         // Set a new property.
print(primStr.myProperty);       // Try to read it back.
```

このスクリプトの出力は次のようになります。

```
16
THIS IS A STRING
undefined
```

例 2

new ステートメントで作成された **String** オブジェクトの場合は、カスタム プロパティを設定できます。

```
var newStr : Object = new String("This is also a string");
print(newStr.length);           // Read the length property.
print(newStr.toUpperCase());    // Call a method.
newStr.myProperty = 42;         // Set a new property.
print(newStr.myProperty);       // Try to read it back.
```

このスクリプトの出力は次のようになります。

```
21
```

```
THIS IS ALSO A STRING
42
```

プロパティおよびメソッド

String オブジェクトのプロパティとメソッド

必要条件

Version 1

参照

関連項目

Object オブジェクト
String 型 (JScript)
new 演算子

概念

文字列データ

String オブジェクトのプロパティとメソッド

String オブジェクトは、テキスト文字列を表すオブジェクトです。このオブジェクトを使用すると、各種文字列操作、文字列の書式設定、文字列内の一部分の取得、文字列内での指定した文字列の検索などを行うことができます。

プロパティ

constructor プロパティ

length プロパティ (String)

prototype プロパティ

メソッド

anchor メソッド

big メソッド

blink メソッド

bold メソッド

charAt メソッド

charCodeAt メソッド

concat メソッド (String)

fixed メソッド

fontcolor メソッド

fontsize メソッド

fromCharCode メソッド

indexOf メソッド

italics メソッド

lastIndexOf メソッド

link メソッド

localeCompare メソッド

match メソッド

replace メソッド

search メソッド

slice メソッド (String)

small メソッド

split メソッド

strike メソッド

sub メソッド

substr メソッド

substring メソッド

sup メソッド

toLocaleLowerCase メソッド

toLocaleUpperCase メソッド

toLowerCase メソッド

toString メソッド

toUpperCase メソッド

valueOf メソッド

参照

その他の技術情報

プロパティ (JScript)
メソッド
JScript リファレンス

VBArray オブジェクト

Visual Basic のセーフ配列にアクセスする手段を提供します。

```
varName = new VBArray(safeArray)
```

引数
varName

必ず指定します。VBArray を代入する変数名。

safeArray

必ず指定します。VBArray の値。

解説

safeArray 引数は、VBArray コンストラクタに渡される前に、VBArray 値を持つ必要があります。この値は、既存の ActiveX か別のオブジェクトから取得できます。

メモ：

JScript で作成された配列、および Visual Basic で作成された配列は、どちらも .NET Framework 配列と相互運用されます。したがって、Visual Basic で作成された配列の要素は、JScript で直接アクセスできます。**VBArray** オブジェクトは下位互換性を維持するためだけに用意されています。配列の詳細については、「Array オブジェクト」、「Dim ステートメント (Visual Basic)」、および AllMembers.T:System.Array を参照してください。

VBArray には多次元を指定できます。各次元のインデックスを個別に指定できます。配列の次元数は、**dimensions** メソッドにより取得できます。各次元で使用されるインデックスの範囲は、**lbound** メソッドおよび **ubound** メソッドにより取得できます。

プロパティ

VBArray オブジェクトには、プロパティはありません。

メソッド

VBArray オブジェクトのメソッド

必要条件

Version 3

参照
関連項目
new 演算子
Array オブジェクト
Array

VBArray オブジェクトのメソッド

VBArray オブジェクトは、Visual Basic のセーフ配列へアクセスする手段を提供します。

メソッド

dimensions メソッド

getItem メソッド

lbound メソッド

toArray メソッド

ubound メソッド

参照

その他の技術情報

メソッド

JScript リファレンス

演算子 (JScript)

JScript には、算術演算子、論理演算子、ビット処理演算子、代入演算子などの多くの演算子が用意されています。ここでは、演算子の使用方法に関する情報へのリンクを示します。

このセクションの内容

加算代入演算子 (+=)

2 つの数値を加算するか、2 つの文字列を連結し、結果を 1 番目の引数に代入します。

加算演算子 (+)

2 つの数値を加算します。または 2 つの文字列を連結します。

代入演算子 (=)

値を変数に代入します。

ビットごとの AND 代入演算子 (&=)

2 つの式でビットごとの AND 演算を実行し、結果を 1 番目の引数に代入します。

ビットごとの AND 演算子 (&)

2 つの式のビットごとの AND 演算を実行します。

ビットごとの左シフト演算子 (<<)

変数の値を、式で指定されたビット数分だけ左へシフトします。

ビットごとの NOT 演算子 (~)

式で指定された値のビットごとの NOT (否定) 演算を実行します。

ビットごとの OR 代入演算子 (|=)

2 つの式でビットごとの OR を実行し、結果を 1 番目の引数に代入します。

ビットごとの OR 演算子 (|)

2 つの式のビットごとの OR 演算を実行します。

ビットごとの右シフト演算子 (>>)

変数の値を式で指定されたビット数分だけ右へシフトします。符号は保持されます。

ビットごとの XOR 代入演算子 (^=)

2 つの式でビットごとの排他的 OR 演算を実行し、結果を 1 番目の引数に代入します。

ビットごとの XOR 演算子 (^)

2 つの式のビットごとの XOR (排他的論理和) 演算を実行します。

コンマ演算子 (,)

2 つの式を順番に実行します。

比較演算子

比較結果を示すブール値を返す、各種の演算子 (==、>、>=、===、!=、<、<=、および !==) です。

条件 (三項) 演算子 (?:)

条件に応じて 2 つのステートメントのどちらを実行するか選択します。

delete 演算子

オブジェクトのプロパティ、または配列の要素を削除します。

除算代入演算子 (/=)

2 つの数値で除算を行い、結果を 1 番目の引数に代入します。

除算演算子 (/)

指定された 2 つの数値で除算を行い、結果を返します。

in 演算子

オブジェクトにプロパティがあるかどうかを調べます。

インクリメント演算子 (++)、デクリメント演算子 (--)

インクリメント演算子 (++) は変数に 1 ずつ加算し、デクリメント演算子 (--) は変数を 1 ずつ減算します。

instanceof 演算子

オブジェクトが特定のクラスのインスタンスかどうかを示すブール値を返します。

左シフト代入演算子 (<<=)

式の各ビットを左にシフトし、結果を 1 番目の引数に代入します。

論理 AND 演算子 (&&)

2 つの式の論理積を求めます。

論理 NOT 演算子 (!)

式で指定された値の論理否定を求めます。

論理 OR 演算子 (||)

2 つの式の論理和を求めます。

剰余代入演算子 (%=)

2 つの数値で除算を行い、剰余を 1 番目の引数に代入します。

剰余演算子 (%)

2 つの数値で除算を行い、剰余を返します。

乗算代入演算子 (*=)

2 つの数値を乗算し、結果を 1 番目の引数に代入します。

乗算演算子 (*)

2 つの数値の積を返します。

new 演算子

新しいオブジェクトを作成します。

参照演算子 (&)

参照パラメータまたは出力パラメータを持つメソッドに、変数の参照を渡します。

右シフト代入演算子 (>>=)

式の各ビットを右にシフトし、結果を 1 番目の引数に代入します。式の符号は維持されます。

減算代入演算子 (-=)

一方の数値からもう一方の値を減算し、結果を 1 番目の引数に代入します。

減算演算子 (-)

数式の負の値を示すか、一方の数値からもう一方の値を減算します。

typeof 演算子

式のデータ型を識別する文字列を返します。

符号なし右シフト代入演算子 (>>>=)

式の各ビットに符号なし右シフトを実行し、結果を 1 番目の引数に代入します。

符号なし右シフト演算子 (>>>)

式の各ビットを指定されたビット数分だけ右へシフトします。

void 演算子

式が値を返すことができないようにします。

関連するセクション

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

JScript の演算子

JScript で使用される演算子の概要を示します。また、各演算子の正しい構文を説明するトピック、および演算子の優先順位の重要性を説明するトピックへのリンクを示します。

演算子の優先順位

JScript の演算子の実行時の優先順位に関する情報の一覧を示します。

演算子の一覧

JScript の演算子と、各演算子の正しい使用方法を説明するリンクを示します。

加算代入演算子 (+=)

変数の値に式で指定された値を加算し、その結果を変数に代入します。

```
result += expression
```

引数

result

任意の変数。

expression

任意の式。

解説

この演算子は、`result = result + expression` と指定する場合とほぼ同じ結果になります。ただし、*result* は一度しか評価されません。

+= 演算子の動作は、指定する式の型によって決まります。

result	***expression***	**動作**
char	**char**	エラー
char 型	数値型	加算
char 型	**String**	エラー
数値型	**char**	加算
数値型	数値型	加算
数値型	**String**	連結
String	**char**	連結
文字列型	数値型	連結
String	**String**	連結

連結の場合、数字は数値の文字列表現に変換され、文字は長さが 1 の文字列と見なされます。文字と数字の加算の場合は、文字が数値に変換され、2 つの値が加算されます。いくつかの型の組み合わせでは、加算の結果を必要な出力の型に変換できないためエラーが発生します。

使用例

次の例は、加算代入演算子が異なる型の式を処理するようすを示しています。

```
var str : String = "42";
var n : int = 20;
var c : char = "A";  // The numeric value of "A" is 65.
var result;
c += n;         // The result is the char "U".
n += c;         // The result is the number 105.
n += n;         // The result is the number 210.
n += str;       // The result is the number 21042.
str += c;       // The result is the string "42U".
str += n;       // The result is the string "42U21042".
str += str;     // The result is the string "42U2104242U21042".
c += c;         // This returns a runtime error.
```

```
c += str;       // This returns a runtime error.
n += "string";  // This returns a runtime error.
```

必要条件

Version 1

参照

関連項目

加算演算子 (+)

代入演算子 (=)

概念

演算子の優先順位

演算子の一覧

加算演算子 (+)

1 つの数式の値を他方に加算します。または文字列の連結を行います。

```
expression1 + expression2
```

引数

expression1

任意の式を指定します。

expression2

任意の式を指定します。

解説

+ 演算子の動作は、指定する式の型によって決まります。

指定された式	動作	結果の型
両方とも文字の場合	連結	**String**
両方とも数値の場合	加算	数値型
両方とも文字列の場合	連結	**String**
一方が文字で他方が数値の場合	加算	**char**
一方が文字で他方が文字列の場合	連結	**String**
一方が数値で他方が文字列の場合	連結	**String**

連結の場合、数字は数値の文字列表現に変換され、文字は長さが 1 の文字列と見なされます。文字と数字の加算の場合は、文字が数値に変換され、2 つの値が加算されます。

メモ：

型の注釈が使用されない場合は、数値データが文字列として格納されることがあります。明示的型変換または型の注釈変数を使用して、加算演算子が数値を文字列として、または文字列を数値として扱わないようにしてください。

使用例

次の例は、加算演算子が異なる型の式を処理するようすを示しています。

```
var str : String = "42";
var n : double = 20;
var c : char = "A";  // the numeric value of "A" is 65
var result;
result = str + str;  // result is the string "4242"
result = n + n;      // result is the number 40
result = c + c;      // result is the string "AA"
result = c + n;      // result is the char "U"
result = c + str;    // result is the string "A42"
result = n + str;    // result is the string "2042"
// Use explicit type coversion to use numbers as strings, or vice versa.
result = int(str) + int(str);    // result is the number 84
result = String(n) + String(n);  // result is the string "2020"
result = c + int(str);           // result is the char "k"
```

必要条件

Version 1

参照

関連項目

加算代入演算子 (+=)

概念

演算子の優先順位

演算子の一覧

型変換

代入演算子 (=)

値を変数に代入します。

```
result = expression
```

引数
result

任意の変数。

expression

任意の式。

解説

= 演算子は、*expression* の値を返し、その値を *variable* に代入します。このため、代入演算子で次のような使い方をすることもできます。

```
j = k = l = 0;
```

このステートメントを実行すると、j、k、および l のすべての変数の値は 0 になります。

expression のデータ型は、*result* のデータ型に変換できる必要があります。

必要条件

Version 1

参照
概念
演算子の優先順位
演算子の一覧

ビットごとの AND 代入演算子 (&=)

変数の値と式で指定された値のビットごとの AND 演算を行い、その結果を変数に代入します。

```
result &= expression
```

引数
result

任意の数値変数を指定します。

expression

任意の数式を指定します。

解説

この演算子は、`result = result & expression` と指定する場合とほぼ同じ結果になります。ただし、*result* は一度しか評価されません。

&= 演算子は、引数を一致するデータ型に変換します。続いて、**&=** 演算子は *result* と *expression* の値を 2 進数形式で取り込み、それに対してビットごとに AND 演算を行います。

この演算の結果は次のようになります。

```
0101    (result)
1100    (expression)
----
0100    (output)
```

両方の式でビットの値が 1 の場合、演算結果のそのビット値は必ず 1 になります。それ以外の場合は 0 になります。

必要条件

Version 1

参照
関連項目
ビットごとの AND 演算子 (&)
代入演算子 (=)
概念
演算子の優先順位
演算子の一覧
ビット処理演算子による型の強制変換

ビットごとの AND 演算子 (&)

2 つの式のビットごとの AND 演算を実行します。

```
expression1 & expression2
```

引数

expression1

任意の数式を指定します。

expression2

任意の数式を指定します。

解説

& 演算子は、引数を一致するデータ型に変換します。続いて、**&** 演算子は 2 つの式の値を 2 進数形式で取り込み、それに対してビットごとに AND 演算を行います。この演算子が返すデータ型は、引数のデータ型で決まります。

この演算の結果は次のようになります。

```
0101   (expression1)
1100   (expression2)
----
0100   (result)
```

両方の式でビットの値が 1 の場合、演算結果のそのビット値は必ず 1 になります。それ以外の場合は 0 になります。

必要条件

Version 1

参照

関連項目

ビットごとの AND 代入演算子 (&=)

概念

演算子の優先順位

演算子の一覧

ビット処理演算子による型の強制変換

ビットごとの左シフト演算子 (<<)

式の各ビットを指定されたビット数分だけ左へシフトします。

```
expression1 << expression2
```

引数
expression1

任意の数式を指定します。

expression2

任意の数式を指定します。

解説

<< 演算子は、*expression1* の各ビットを *expression2* で指定されたビット数分だけ左へシフトします。この演算子が返すデータ型は、*expression1* のデータ型で決まります。

<< 演算子は *expression2* をマスクして、*expression1* のビットが過剰にシフトされるのを防ぎます。マスクを行わないと、シフト量が *expression1* のデータ型のビット数より大きい場合、単純な結果を得るために元のビットがすべてシフトされることになります。元のビットが少なくとも 1 ビットは残るように、シフト演算子は *expression2* を *expression1* のビット数より 1 小さい値で (ビットごとの AND 演算子を使用して) マスクし、実際のシフト量を計算します。

使用例

次に例を示します。

```
var temp
temp = 14 << 2
```

変数 temp の値は、14 (2 進数で 00001110) から 2 ビット分だけ左シフトされて 56 (2 進数で 00111000) になります。

マスクの動作を次の例に示します。

```
var x : byte = 15;
// A byte stores 8 bits.
// The bits stored in x are 00001111
var y : byte = x << 10;
// Actual shift is 10 & (8-1) = 2
// The bits stored in y are 00111100
// The value of y is 60
print(y); // Prints 60
```

必要条件

Version 1

参照
関連項目
左シフト代入演算子 (<<=)
ビットごとの右シフト演算子 (>>)
符号なし右シフト演算子 (>>>)
概念
演算子の優先順位
演算子の一覧
ビット処理演算子による型の強制変換

ビットごとの NOT 演算子 (~)

式で指定された値のビットごとの NOT (否定) 演算を実行します。

```
~ expression
```

引数
expression

任意の数式を指定します。

解説

~ 演算子は、式の値を 2 進数形式で取り込み、その各ビットを反転させます。この演算の結果は次のようになります。

```
0101   (expression)
----
1010   (result)
```

元の式でビットが 1 の場合は必ず 0 になります。元の式でビットが 0 の場合は必ず 1 になります。

~ 演算子を整数型のオペランドに使用すると、演算子は型の変換を行わず、オペランドと同じ型の値を返します。オペランドが整数以外の型である場合は、演算が実行される前に値が **int** に変換され、演算子の戻り値は **int** 型になります。

必要条件

Version 1

参照
関連項目
論理 NOT 演算子 (!)
概念
演算子の優先順位
演算子の一覧

ビットごとの OR 代入演算子 (|=)

変数の値と式で指定された値のビットごとに OR 演算を行い、その結果を変数に代入します。

```
result |= expression
```

引数
result

任意の数値変数を指定します。

expression

任意の数式を指定します。

解説

この演算子は、`result = result | expression` と指定する場合とほぼ同じ結果になります。ただし、*result* は一度しか評価されません。

|= 演算子は、引数を一致するデータ型に変換します。続いて、**|** = 演算子は *result* と *expression* の値を 2 進数形式で取り込み、それに対してビットごとに OR 演算を行います。この演算の結果は次のようになります。

```
0101    (result)
1100    (expression)
----
1101    (output)
```

どちらか一方の式でビットの値が 1 の場合、演算結果のそのビットの値は 1 になります。それ以外の場合は 0 になります。

必要条件

Version 1

参照
関連項目
ビットごとの OR 演算子 (|)
代入演算子 (=)
概念
演算子の優先順位
演算子の一覧
ビット処理演算子による型の強制変換

ビットごとの OR 演算子 (|)

2 つの式のビットごとの OR 演算を実行します。

```
expression1 | expression2
```

引数
expression1

任意の数式を指定します。

expression2

任意の数式を指定します。

解説

| 演算子は、引数を一致するデータ型に変換します。続いて、**|** 演算子は 2 つの式の値を 2 進数形式で取り込み、それに対してビットごとの OR 演算を行います。この演算子が返すデータ型は、引数のデータ型で決まります。

この演算の結果は次のようになります。

```
0101   (expression1)
1100   (expression2)
----
1101   (result)
```

指定された 2 つの値で、どちらかの桁が 1 である場合、演算結果のその桁は必ず 1 になります。それ以外の場合は 0 になります。

必要条件

Version 1

参照
関連項目
ビットごとの OR 代入演算子 (|=)
概念
演算子の優先順位
演算子の一覧
ビット処理演算子による型の強制変換

ビットごとの右シフト演算子 (>>)

式の各ビットを指定されたビット数分だけ右へシフトします。ただし、符号は保持されます。

```
expression1 >> expression2
```

引数
expression1

任意の数式を指定します。

expression2

任意の数式を指定します。

解説

>> 演算子は、*expression1* の各ビットを *expression2* で指定されたビット数分だけ右へシフトします。上位ビットは、*expression1* の符号ビットで埋められます。シフトされて最下位ビットより右へ移動した桁は破棄されます。この演算子が返すデータ型は、*expression1* のデータ型で決まります。

>> 演算子は *expression2* をマスクして、*expression1* のビットが過剰にシフトされるのを防ぎます。マスクを行わないと、シフト量が *expression1* のデータ型のビット数より大きい場合、単純な結果を得るために元のビットがすべてシフトされることになります。元のビットが少なくとも 1 ビットは残るように、シフト演算子は *expression2* を *expression1* のビット数より 1 小さい値で (ビットごとの AND 演算子を使用して) マスクし、実際のシフト量を計算します。

使用例

たとえば次に示すコードでは、変数 *temp* の値は、-14 (2 進数で 11110010) から 2 ビット分だけ右へシフトされて -4 (2 進数で 11111100) になります。

```
var temp
temp = -14 >> 2
```

マスクの動作を次の例に示します。

```
var x : byte = 15;
// A byte stores 8 bits.
// The bits stored in x are 00001111
var y : byte = x >> 10;
// Actual shift is 10 & (8-1) = 2
// The bits stored in y are 00000011
// The value of y is 3
print(y); // Prints 3
```

必要条件

Version 1

参照
関連項目
ビットごとの左シフト演算子 (<<)
右シフト代入演算子 (>>=)
符号なし右シフト演算子 (>>>)
概念
演算子の優先順位
演算子の一覧
ビット処理演算子による型の強制変換

ビットごとの XOR 代入演算子 (^=)

変数の値と式で指定された値のビットごとに排他的 OR を行い、その結果を変数に代入します。

```
result ^= expression
```

引数

result

任意の数値変数を指定します。

expression

任意の数式を指定します。

解説

この演算子は、`result = result ^ expression` と指定する場合とほぼ同じ結果になります。ただし、*result* は一度しか評価されません。

^= 演算子は、引数を適切なデータ型に変換します。続いて、^= 演算子は 2 つの式の値を 2 進数形式で取り込み、それに対してビットごとに排他的 OR の演算を行います。この演算の結果は次のようになります。

```
0101    (result)
1100    (expression)
----
1001    (result)
```

一方の式だけでビットの値が 1 の場合、演算結果のそのビットの値は 1 になります。それ以外の場合は 0 になります。

必要条件

Version 1

参照

関連項目

ビットごとの XOR 演算子 (^)

代入演算子 (=)

概念

演算子の優先順位

演算子の一覧

ビット処理演算子による型の強制変換

ビットごとの XOR 演算子 (^)

2 つの式のビットごとの XOR (排他的論理和) 演算を実行します。

```
expression1 ^ expression2
```

引数
expression1

任意の数式を指定します。

expression2

任意の数式を指定します。

解説

^ 演算子は、引数を一致するデータ型に変換します。続いて、^ 演算子は 2 つの式の値を 2 進数形式で取り込み、それに対してビットごとに排他的 OR 演算を行います。この演算子が返すデータ型は、引数のデータ型で決まります。

この演算の結果は次のようになります。

```
0101   (expression1)
1100   (expression2)
----
1001   (result)
```

一方の式だけでビットの値が 1 の場合、演算結果のそのビットの値は 1 になります。それ以外の場合は 0 になります。

必要条件

Version 1

参照
関連項目
ビットごとの XOR 代入演算子 (^=)
概念
演算子の優先順位
演算子の一覧
ビット処理演算子による型の強制変換

コンマ演算子 (,)

2 つの式を順番に実行します。

```
expression1, expression2
```

引数
expression1

任意の式を指定します。

expression2

任意の式を指定します。

解説

, 演算子は、その前後の式を左から順に実行させて、右側の式の結果を返します。**,** 演算子が最もよく使用されるのは、**for** ループのインクリメント式の中です。次に例を示します。

```
var i, j, k;
for (i = 0; i < 10; i++, j++) {
   k = i + j;
}
```

for ステートメントでは、ループ内の処理の最後に実行する式として指定できるのは単一の式だけです。**,** 演算子を使用すると、複数の式を単一の式として扱うことができるため、この制限を回避できます。

必要条件

Version 1

参照
関連項目
for ステートメント
概念
演算子の優先順位
演算子の一覧

比較演算子

2 つの式を比較した結果を示すブール値を返します。

```
expression1 comparisonoperator expression2
```

引数

expression1

任意の式。

comparisonoperator

任意の比較演算子 (<、>、<=、>=、==、!=、===、!==)。

expression2

任意の式。

解説

文字列を比較する場合、JScript では Unicode のコード順に基づいて比較を行います。

比較演算子の種類によって、*expression1* と *expression2* がどのように比較されるかを次に示します。

関係演算子 (<、>、<=、>=)

- *expression1* と *expression2* の両方を数値に変換しようとします。
- 式が両方とも文字列の場合は Unicode 順で比較が行われます。
- どちらか一方の式が **NaN** の場合は偽 (**false**) が返されます。
- 負の 0 は、正の 0 と等しいと評価されます。
- 負の無限大は、それ自身を含むすべての式よりも小さいと評価されます。
- 正の無限大は、それ自身を含むすべての式よりも大きいと評価されます。

等価演算子 (==、!=)

- 2 つの式のデータ型が異なる場合は、文字列型、数値型、ブール型の順に変換を試みます。
- **NaN** は、それ自身を含む、どのような値とも等しくないと評価されます。
- 負の 0 は、正の 0 と等しいと評価されます。
- **NULL** は、**NULL** と **undefined** の両方と等しいと評価されます。
- 2 つの値が同一の文字列である場合、等価な数値である場合、同じオブジェクトである場合、および同一のブール値である場合、これらの値は等しいと評価されます。また、2 つの値のデータ型が異なる場合に、データ型の変換を行ってこのうちのいずれかに当てはまれば、2 つの値は等しいと評価されます。
- 上記以外の比較演算は等しくないと評価されます。

ID (===、!==)

これらの演算子は、等価演算子と同様の比較を行います。ただし、データ型の変換は行われず、同じデータ型でないと等しいとは評価されません。

必要条件

Version 1

参照

概念

演算子の優先順位
演算子の一覧

条件 (三項) 演算子 (?:)

条件に応じて 2 つの式のどちらかを返します。

```
test ? expression1 : expression2
```

引数
test

任意のブール式。

expression1

test が真 (**true**) の場合に返される式。コンマ式も使用できます。

expression2

test が偽 (**false**) の場合に返される式。コンマ式も使用できます。

解説

?: 演算子を使用して、**if...else** ステートメントと同じ処理を簡単に実行できます。**?:** 演算子は通常、**if...else** ステートメントが記述しづらい長い式の 1 部として使用されます。次に例を示します。

```
var now = new Date();
var greeting = "Good" + ((now.getHours() > 17) ? " evening." : " day.");
```

この例は、午後 6 時より前では "皆さん、こんにちは"、午後 6 時以降では "皆さん、こんばんは" という文字列を作成します。上記の例は、**if...else** ステートメントを使用すると、次のようになります。

```
var now = new Date();
var greeting = "Good";
if (now.getHours() > 17)
   greeting += " evening.";
else
   greeting += " day.";
```

必要条件

Version 1

参照
関連項目
if...else ステートメント
概念
演算子の優先順位
演算子の一覧

delete 演算子

オブジェクトのプロパティ、配列の要素、または IDictionary オブジェクトのエントリを削除します。

```
delete expression
```

引数
expression

必ず指定します。プロパティ参照、配列要素、または IDictionary オブジェクトとなる式を指定します。

解説

expression の結果がオブジェクトで、*expression* に指定したプロパティが存在し、さらにオブジェクトがそのプロパティの削除を禁止している場合は、偽 (**false**) が返されます。

その他の場合は、真 (**true**) が返されます。

使用例

delete 演算子の使用例を次に示します。

```
// Make an object with city names and an index letter.
var cities : Object = {"a" : "Athens" , "b" : "Belgrade", "c" : "Cairo"}

// List the elements in the object.
var key : String;
for (key in cities) {
   print(key + " is in cities, with value " + cities[key]);
}

print("Deleting property b");
delete cities.b;

// List the remaining elements in the object.
for (key in cities) {
   print(key + " is in cities, with value " + cities[key]);
}
```

このコードの出力は次のようになります。

```
a is in cities, with value Athens
b is in cities, with value Belgrade
c is in cities, with value Cairo
Deleting property b
a is in cities, with value Athens
c is in cities, with value Cairo
```

必要条件

Version 3

参照
関連項目
IDictionary Interface
概念
演算子の優先順位
演算子の一覧

除算代入演算子 (/=)

変数の値を式で指定された値で除算し、その結果を変数に代入します。

```
result /= expression
```

引数
result

任意の数値変数を指定します。

expression

任意の数式を指定します。

解説

この演算子は、`result = result / expression` と指定する場合とほぼ同じ結果になります。ただし、*result* は一度しか評価されません。

必要条件

Version 1

参照
関連項目
除算演算子 (/)
代入演算子 (=)
概念
演算子の優先順位
演算子の一覧

除算演算子 (/)

2 つの数式の値の除算を行います。

```
number1 / number2
```

引数
number1

任意の数式を指定します。

number2

任意の数式を指定します。

解説

number1 が有限の 0 以外の数字で *number2* が 0 の場合、除算結果は、*number1* が正の場合は **Infinity** に、負の場合は **-Infinity** になります。*number1* と *number2* の両方が 0 の場合、結果は **NaN** になります。

必要条件

Version 1

参照
関連項目
除算代入演算子 (/=)
概念
演算子の優先順位
演算子の一覧

in 演算子

オブジェクトにプロパティがあるかどうかを調べます。

```
property in object
```

引数

property

必ず指定します。文字列型に評価される式を指定します。

object

必ず指定します。任意のオブジェクトを指定します。

解説

in 演算子は、オブジェクトに property という名前のプロパティがあるかどうかを調べます。また、オブジェクトのプロトタイプも調べるため、*property* がプロトタイプ チェインの一部かどうかもわかります。*property* がオブジェクトかプロトタイプ チェインの一部である場合、**in** 演算子は **true** を返し、そうでない場合は **false** を返します。

in 演算子と **for...in** ステートメントを混同しないようにしてください。

メモ :

オブジェクト自身にプロパティがあるかどうか、およびオブジェクトがプロトタイプ チェインからプロパティを継承していないかを確認するには、オブジェクトの **hasOwnProperty** メソッドを使用します。

使用例

in 演算子の使用例を次に示します。

```
function cityName(key : String, cities : Object) : String {
   // Returns a city name associated with an index letter.
   var ret : String = "Key '" + key + "'";
   if( key in cities )
      return ret + " represents " + cities[key] + ".";
   else  // no city indexed by the key
      return ret + " does not represent a city."
}

// Make an object with city names and an index letter.
var cities : Object = {"a" : "Athens" , "b" : "Belgrade", "c" : "Cairo"}

// Look up cities with an index letter.
print(cityName("a",cities));
print(cityName("z",cities));
```

このコードの出力は次のようになります。

```
Key 'a' represents Athens.
Key 'z' does not represent a city.
```

必要条件

Version 1

参照

関連項目

for...in ステートメント

hasOwnProperty メソッド

概念

演算子の優先順位

演算子の一覧

インクリメント演算子 (++)、デクリメント演算子 (--)

変数の値をインクリメント (1 だけ増加) またはデクリメント (1 だけ減少) します。

```
//prefix syntax
++variable
--variable
//postfix syntax
variable++
variable--
```

引数
variable

任意の数値変数を指定します。

解説

インクリメント演算子およびデクリメント演算子は、変数の値を修正したり、値にアクセスする処理を簡単にするための演算子です。どちらの演算子も、前置構文または後置構文で使用できます。

If	等価な演算	戻り値
++ *variable*	*variable* += 1	インクリメントした後の *variable* の値
variable ++	*variable* += 1	インクリメントする前の *variable* の値
-- *variable*	*variable* -= 1	デクリメントした後の *variable* の値
variable --	*variable* -= 1	デクリメントする前の *variable* の値

使用例

次の例は、++ 演算子の前置構文と後置構文の違いを示しています。

```
// Example of prefix increment operator
var j1 : int = 2;
var k1 : int;
k1 = ++j1;          // k1 is 3, the value of j1 after incrementing

// Example of postfix increment operator
var j2 : int = 2;
var k2 : int;
k2 = j2++;          // k2 is 2, the value of j2 before incrementing
```

必要条件

Version 1

参照
概念
演算子の優先順位
演算子の一覧

instanceof 演算子

オブジェクトが特定のクラスまたは作成した関数のインスタンスかどうかを示すブール値を返します。

```
object instanceof class
```

引数
object

必ず指定します。任意のオブジェクト式を指定します。

class

必ず指定します。任意のオブジェクト クラスまたは作成した関数を指定します。

解説

引数 *object* が引数 *class* または作成した関数のインスタンスである場合、**instanceof** 演算子は真 (**true**) を返します。*object* が指定されたクラスのインスタンスでない場合、または *object* が **NULL** 値である場合は、偽 (**false**) を返します。

JScript の **Object** は特殊です。オブジェクトは **Object** コンストラクタで作成された場合にだけ、**Object** のインスタンスであると見なされます。

例 1

instanceof 演算子を使用して変数の型を確認する方法を次に示します。

```
// This program uses System.DateTime, which must be imported.
import System

function isDate(ob) : String {
   if (ob instanceof Date)
      return "It's a JScript Date"
   if (ob instanceof DateTime)
      return "It's a .NET Framework Date"
   return "It's not a date"
}

var d1 : DateTime = DateTime.Now
var d2 : Date = new Date
print(isDate(d1))
print(isDate(d2))
```

このコードの出力は次のようになります。

```
It's a .NET Date
It's a JScript Date
```

例 2

instanceof 演算子を使用して、作成された関数のインスタンスを確認する方法を次に示します。

```
function square(x : int) : int {
   return x*x
}

function bracket(s : String) : String{
   return("[" + s + "]");
}

var f = new square
print(f instanceof square)
print(f instanceof bracket)
```

このコードの出力は次のようになります。

```
true
false
```

例 3

instanceof 演算子を使用して、オブジェクトが **Object** のインスタンスかどうかを確認する方法を次に示します。

```
class CDerived extends Object {
   var x : double;
}

var f : CDerived = new CDerived;
var ob : Object = f;
print(ob instanceof Object);

ob = new Object;
print(ob instanceof Object);
```

このコードの出力は次のようになります。

```
false
true
```

必要条件

Version 5

参照

概念

演算子の優先順位

演算子の一覧

左シフト代入演算子 (<<=)

変数の値を式で指定されたビット数分だけ左へシフトし、その結果を変数に代入します。

```
result <<= expression
```

引数
result

任意の数値変数を指定します。

expression

任意の数式を指定します。

解説

この演算子は、`result = result << expression` と指定する場合とほぼ同じ結果になります。ただし、*result* は一度しか評価されません。

<<= 演算子は、*result* の各ビットを *expression* で指定されたビット数分だけ左へシフトします。演算子は *expression* をマスクして、*result* のビットが過剰にシフトされるのを防ぎます。マスクを行わないと、シフト量が *result* のデータ型のビット数より大きい場合、単純な結果を得るために元のビットがすべてシフトされることになります。元のビットが少なくとも 1 ビットは残るように、シフト演算子は *expression* を *result* のビット数より 1 小さい値で (ビットごとの AND 演算子を使用して) マスクし、実際のシフト量を計算します。

使用例

次に例を示します。

```
var temp
temp = 14
temp <<= 2
```

変数 temp の値は、14 (2 進数で 00001110) から 2 ビット分だけ左シフトされて 56 (2 進数で 00111000) になります。下位ビットは 0 で埋められます。

マスクの動作を次の例に示します。

```
var x : byte = 15;
// A byte stores 8 bits.
// The bits stored in x are 00001111
x <<= 10;
// Actual shift is 10 & (8-1) = 2
// The bits stored in x are 00111100
// The value of x is 60
print(x); // Prints 60
```

必要条件

Version 1

参照
関連項目
ビットごとの左シフト演算子 (<<)
ビットごとの右シフト演算子 (>>)
符号なし右シフト演算子 (>>>)
代入演算子 (=)
概念
演算子の優先順位
演算子の一覧

論理 AND 演算子 (&&)

2 つの式の論理積を求めます。

```
expression1 && expression2
```

引数
expression1

任意の式を指定します。

expression2

任意の式を指定します。

解説

指定された 2 つの式が両方とも真 (**true**) の場合だけ、結果も真 (**true**) になります。元の式のどちらかが偽 (**false**) の場合、結果は偽 (**false**) になります。次の表は、2 つの式の値と演算結果の値の対応を示しています。

expression1 を変換した値	**expression2 を変換した値**	**演算結果**	**結果を変換した値**
true	**true**	expression2	**true**
true	**false**	expression2	**false**
false	**true**	expression1	**false**
false	**false**	expression1	**false**

JScript では、非ブール値がブール値に変換される場合は、次の規則が適用されます。

- オブジェクトはすべて真 (**true**) となります。
- 文字列は、長さ 0 の文字列の場合だけ偽 (**false**) となります。
- **null** と undefined は偽 (**false**) となります。
- 数値は、0 の場合だけ偽 (**false**) となります。

必要条件

Version 1

参照
概念
演算子の優先順位
演算子の一覧

論理 NOT 演算子 (!)

式で指定された値の論理否定を求めます。

```
!expression
```

引数
expression

任意の式を指定します。

解説

次の表は、式の値と演算結果の値の対応を示しています。

expression を変換した値	**演算結果**
true	**false**
false	**true**

単項演算子での式の評価は、**!** 演算子も含めてすべて次のように行われます。

- undefined または **NULL** を持つ式を指定すると、実行時エラーが発生します。
- オブジェクトは文字列に変換されます。
- 文字列は、数値に変換されます。数値に変換できない場合は、実行時エラーが発生します。
- ブール値は数値として扱われます (**false** の場合は 0、**true** の場合は 1)。

演算子は、結果として導かれた数値に適用されます。

! 演算子では、*expression* が 0 以外の値の場合、*result* は 0 になります。*expression* が 0 の場合、*result* は 1 になります。

必要条件

Version 1

参照
関連項目
ビットごとの NOT 演算子 (~)
概念
演算子の優先順位
演算子の一覧

論理 OR 演算子 (||)

2 つの式の論理和を求めます。

```
expression1 || expression2
```

引数

expression1

任意の式を指定します。

expression2

任意の式を指定します。

解説

2 つの式のどちらか一方、または両方が真 (**true**) の場合、演算結果は真 (**true**) になります。次の表は、2 つの式の値と演算結果の値の対応を示しています。

expression1 を変換した値	**expression2 を変換した値**	**演算結果**	**結果を変換した値**
true	**true**	expression1	**true**
true	**false**	expression1	**true**
false	**true**	expression2	**true**
false	**false**	expression2	**false**

JScript では、非ブール値がブール値に変換される場合は、次の規則が適用されます。

- オブジェクトはすべて真 (**true**) となります。
- 文字列は、長さ 0 の文字列の場合だけ偽 (**false**) となります。
- **null** と undefined は偽 (**false**) となります。
- 数値は、0 の場合だけ偽 (**false**) となります。

必要条件

Version 1

参照

概念

演算子の優先順位

演算子の一覧

剰余代入演算子 (%=)

変数の値を式で指定された値で除算し、その剰余を変数に代入します。

```
result %= expression
```

引数
result

任意の数値変数を指定します。

expression

任意の数式を指定します。

解説

この演算子は、`result = result % expression` と指定する場合とほぼ同じ結果になります。ただし、*result* は一度しか評価されません。

必要条件

Version 1

参照
関連項目
剰余演算子 (%)
代入演算子 (=)
概念
演算子の優先順位
演算子の一覧

剰余演算子 (%)

1 つの式の値を他方で除算し、その剰余を返します。

```
number1 % number2
```

引数
number1

任意の数式を指定します。

number2

任意の数式を指定します。

解説

剰余演算子は、引数 *number1* を引数 *number2* で除算し、その剰余だけを返します。結果の符号は、*number1* の符号と同じです。結果の値は、0 から *number2* の絶対値の範囲になります。

剰余演算子の引数には、浮動小数点数も指定できます。`5.6 % 0.5` では `0.1` が返されます。

使用例

剰余演算子の使用例を次に示します。

```
var myMoney : int = 128;
var cookiePrice : int = 33;
// Calculate the change if the maximum number of cookies are bought.
var change : int = myMoney % cookiePrice;
// Calculate number of cookies bought.
var numCookies : int = Math.round((myMoney-change)/cookiePrice);
```

必要条件

Version 1

参照
関連項目
剰余代入演算子 (%=)
概念
演算子の優先順位
演算子の一覧

乗算代入演算子 (*=)

変数の値を式で指定された値で乗算し、その結果を変数に代入します。

```
result *= expression
```

引数
result

任意の数値変数を指定します。

expression

任意の数式を指定します。

解説

この演算子は、`result = result * expression` と指定する場合とほぼ同じ結果になります。ただし、*result* は一度しか評価されません。

必要条件

Version 1

参照
関連項目
乗算演算子 (*)
代入演算子 (=)
概念
演算子の優先順位
演算子の一覧

乗算演算子 (*)

2 つの式の値の積を返します。

```
number1 * number2
```

引数
number1

任意の数式を指定します。

number2

任意の数式を指定します。

解説

乗算演算子は、引数 *number1* に引数 *number2* を乗算し、その結果を返します。引数のどちらかが **NaN** の場合は、結果も **NaN** となります。**Infinity** と 0 を乗算すると、結果は **NaN** になります。また、**Infinity** と 0 以外の数値 (**Infinity** を含む) を乗算すると、結果は **Infinity** になります。

必要条件

Version 1

参照
関連項目
乗算代入演算子 (*=)
概念
演算子の優先順位
演算子の一覧

new 演算子

新しいオブジェクトを作成します。

```
new constructor[( [arguments] )]
```

引数

constructor

必ず指定します。オブジェクトのコンストラクタを指定します。引数がない場合は、かっこを省略できます。

arguments

省略可能です。新しいオブジェクトのコンストラクタに渡す任意の引数を指定します。

解説

new 演算子は次の処理を実行します。

- メンバを持たないオブジェクトを作成します。
- 新しく作成したオブジェクトのコンストラクタを、そのオブジェクトへの参照を **this** ポインタとして渡して呼び出します。
- コンストラクタは、受け取った引数に従って、オブジェクトを初期化します。

使用例

new 演算子の使用例を次に示します。

```
var myObject : Object = new Object;
var myArray : Array = new Array();
var myDate : Date = new Date("Jan 5 1996");
```

必要条件

Version 1

参照

関連項目

function ステートメント

参照演算子 (&)

& 演算子は、参照パラメータまたは出力パラメータを持つメソッドに、変数の参照を渡すために使用します。呼び出し先のメソッド内でパラメータに加えられた変更は、呼び出し元のメソッドに制御が返されたときに、参照渡しされた変数に反映されます。

```
&expression
```

パラメータ

expression

メソッドに渡す変数を指定します。

解説

JScript では、参照パラメータおよび出力パラメータを持つメソッドを呼び出すことができますが、定義することはできません。

使用例

参照 (&) 演算子の使用例を次に示します。

```
// Define Compute method in C# code.
public class C
{
    public static void Compute(ref int sum, out int product, int a, int b)
    {
        sum = a + b;
        product = a * b;
    }
}

// Call Compute method from your JScript code.
var a : int, b: int;
C.Compute(&a, &b, 2, 3)
print(a);
print(b);
```

参照

関連項目

ref (C# リファレンス)

out (C# リファレンス)

右シフト代入演算子 (>>=)

変数の値を式で指定されたビット数分だけ右へシフトし、その結果を変数に代入します。ただし、変数の符号は保持されます。

```
result >>= expression
```

引数
result

任意の数値変数を指定します。

expression

任意の数式を指定します。

解説

この演算子は、`result = result >> expression` と指定する場合とほぼ同じ結果になります。ただし、*result* は一度しか評価されません。

>>= 演算子は、*result* の各ビットを *expression* で指定されたビット数分だけ右へシフトします。上位ビットは、*result* の符号ビットで埋められます。シフトされて最下位ビットより右へ移動した桁は破棄されます。演算子は *expression* をマスクして、*result* のビットが過剰にシフトされるのを防ぎます。マスクを行わないと、シフト量が *result* のデータ型のビット数より大きい場合、単純な結果を得るために元のビットがすべてシフトされることになります。元のビットが少なくとも 1 ビットは残るように、シフト演算子は *expression* を *result* のビット数より 1 小さい値で (ビットごとの AND 演算子を使用して) マスクし、実際のシフト量を計算します。

使用例

たとえば次に示すコードでは、変数 *temp* の値は、14 (2 進数で 11110010) から 2 ビット分だけ右へシフトされて -4 (2 進数で 11111100) になります。

```
var temp
temp = -14
temp >>= 2
```

マスクの動作を次の例に示します。

```
var x : byte = 15;
// A byte stores 8 bits.
// The bits stored in x are 00001111
x >>= 10;
// Actual shift is 10 & (8-1) = 2
// The bits stored in x are 00000011
// The value of x is 3
print(x); // Prints 3
```

必要条件

Version 1

参照
関連項目
ビットごとの左シフト演算子 (<<)
ビットごとの右シフト演算子 (>>)
符号なし右シフト演算子 (>>>)
代入演算子 (=)
概念
演算子の優先順位
演算子の一覧
ビット処理演算子による型の強制変換

減算代入演算子 (-=)

変数の値から式で指定された値を減算し、その結果を変数に代入します。

```
result -= expression
```

引数
result

任意の数値変数を指定します。

expression

任意の数式を指定します。

解説

この演算子は、`result = result - expression` と指定する場合とほぼ同じ結果になります。ただし、*result* は一度しか評価されません。

必要条件

Version 1

参照
関連項目
減算演算子 (-)
代入演算子 (=)
概念
演算子の優先順位
演算子の一覧

減算演算子 (-)

1 つの式の値から他方を減算します。または、式の符号を反転した値 (数値の負の値) を求めます。

```
number1 - number2
```

```
-number
```

引数
number1

任意の数式を指定します。

number2

任意の数式を指定します。

number

任意の数式を指定します。

解説

構文 1 では、**-** 演算子は、減算演算子として 2 つの数値の差を求めるために使用されます。構文 2 では、**-** 演算子は、単項マイナス符号演算子として式の符号を反転した値 (数値の負の値) を指定するために使用されます。

構文 2 の場合には、他のすべての単項演算子の場合と同様、式の評価は次のように行われます。

- undefined または **NULL** を持つ式を指定すると、実行時エラーが発生します。
- オブジェクトは文字列に変換されます。
- 文字列は、数値に変換されます。数値に変換できない場合は、実行時エラーが発生します。
- ブール値は数値として扱われます (偽の場合は 0、真の場合は 1)。

演算子は、結果として導かれた数値に適用されます。構文 2 では、対象の数値が 0 以外の場合は *result* が元の値の符号を反転させた値になります。対象の数値が 0 の場合は、結果も 0 となります。

必要条件

Version 1

参照
関連項目
減算代入演算子 (-=)
概念
演算子の優先順位
演算子の一覧

typeof 演算子

式のデータ型を識別する文字列を返します。

```
typeof[(]expression[)] ;
```

引数
expression

必ず指定します。任意の式を指定します。

解説

typeof 演算子は、型情報を文字列で返します。**typeof** 演算子が返す文字列は、"number"、"string"、"boolean"、"object"、"function"、"date"、"undefined"、"unknown" の 8 つです。

typeof の構文のかっこは省略できます。

メモ：

JScript のすべての式で、**GetType** メソッドを使用できます。このメソッドは、式のデータ型を返します (データ型を表す文字列ではありません)。**typeof** 演算子よりも、**GetType** メソッドの方がより詳細な情報を得られます。

使用例

typeof 演算子の使用例を次に示します。

```
var x : double = Math.PI;
var y : String = "Hello";
var z : int[] = new int[10];

print("The type of x (a double) is " + typeof(x)  );
print("The type of y (a String) is " + typeof(y) );
print("The type of z (an int[]) is " + typeof(z) );
```

このコードの出力は次のようになります。

```
The type of x (a double) is number

The type of y (a String) is string

The type of z (an int[]) is object
```

必要条件

Version 1

参照
関連項目
GetType
概念
演算子の優先順位
演算子の一覧

符号なし右シフト代入演算子 (>>>=)

変数の値を式で指定されたビット数分だけ右へシフトし、その結果を変数に代入します。ただし、変数の符号は保持されません。

```
result >>>= expression
```

引数

result

任意の数値変数を指定します。

expression

任意の数式を指定します。

解説

この演算子は、`result = result >>> expression` と指定する場合とほぼ同じ結果になります。ただし、*result* は一度しか評価されません。

>>>= 演算子は、*result* の各ビットを *expression* で指定されたビット数分だけ右へシフトします。上位ビットは、0 で埋められます。シフトされて最下位ビットより右へ移動した桁は破棄されます。演算子は *expression* をマスクして、*result* のビットが過剰にシフトされるのを防ぎます。マスクを行わないと、シフト量が *result* のデータ型のビット数より大きい場合、単純な結果を得るために元のビットがすべてシフトされることになります。元のビットが少なくとも 1 ビットは残るように、シフト演算子は *expression* を *result* のビット数より 1 小さい値で (ビットごとの AND 演算子を使用して) マスクし、実際のシフト量を計算します。

使用例

次に例を示します。

```
var temp
temp = -14
temp >>>= 2
```

変数 *temp* の値は、-14 (2 進数で 11111111 11111111 11111111 11110010) から 2 ビット分だけ右シフトされて 1073741820 (2 進数で 00111111 11111111 11111111 11111100) になります。

マスクの動作を次の例に示します。

```
var x : byte = 15;
// A byte stores 8 bits.
// The bits stored in x are 00001111
x >>>= 10;
// Actual shift is 10 & (8-1) = 2
// The bits stored in x are 00000011
// The value of x is 3
print(x); // Prints 3
```

必要条件

Version 1

参照

関連項目

符号なし右シフト演算子 (>>>)
ビットごとの左シフト演算子 (<<)
ビットごとの右シフト演算子 (>>)
代入演算子 (=)

概念

演算子の優先順位
演算子の一覧
ビット処理演算子による型の強制変換

符号なし右シフト演算子 (>>>)

式の各ビットを指定されたビット数分だけ右へシフトします。ただし、符号は保持されません。

```
expression1 >>> expression2
```

引数
expression1

任意の数式を指定します。

expression2

任意の数式を指定します。

解説

>>> 演算子は、*expression1* の各ビットを *expression2* で指定されたビット数分だけ右へシフトします。上位ビットは、0 で埋められます。シフトされて最下位ビットより右へ移動した桁は破棄されます。この演算子が返すデータ型は、*expression1* のデータ型で決まります。

>>> 演算子は *expression2* をマスクして、*expression1* のビットが過剰にシフトされるのを防ぎます。マスクを行わないと、シフト量が *expression1* のデータ型のビット数より大きい場合、単純な結果を得るために元のビットがすべてシフトされることになります。元のビットが少なくとも 1 ビットは残るように、シフト演算子は *expression2* を *expression1* のビット数より 1 小さい値で (ビットごとの AND 演算子を使用して) マスクし、実際のシフト量を計算します。

使用例

次に例を示します。

```
var temp
temp = -14 >>> 2
```

変数 *temp* の値は、-14 (2 進数で 11111111 11111111 11111111 11110010) から 2 ビット分だけ右シフトされて 1073741820 (2 進数で 00111111 11111111 11111111 11111100) になります。

マスクの動作を次の例に示します。

```
var x : byte = 15;
// A byte stores 8 bits.
// The bits stored in x are 00001111
var y : byte = x >>> 10;
// Actual shift is 10 & (8-1) = 2
// The bits stored in y are 00000011
// The value of y is 3
print(y); // Prints 3
```

必要条件

Version 1

参照
関連項目
符号なし右シフト代入演算子 (>>>=)
ビットごとの左シフト演算子 (<<)
ビットごとの右シフト演算子 (>>)
概念
演算子の優先順位
演算子の一覧
ビット処理演算子による型の強制変換

void 演算子

式が値を返すことができないようにします。

```
void expression
```

引数
expression

必ず指定します。任意の式を指定します。

解説

void 演算子は、式を評価し、undefined を返します。式の評価が必要で、スクリプトの残りの部分でその結果が参照される必要がない場合に使用すると便利です。

必要条件

Version 2

参照
概念
演算子の優先順位
演算子の一覧

プロパティ (JScript)

プロパティは、オブジェクトのメンバである値または値のセット (配列やオブジェクト) です。ここでは、JScript のプロパティの使用方法に関する情報へのリンクを示します。

このセクションの内容

0...n プロパティ

実行した関数の **arguments** プロパティで返される **arguments** オブジェクトから、各引数の実際の値を返します。

$1...$9 プロパティ

パターン一致で検出された、記憶されている最新の 9 つの部分を返します。

arguments プロパティ

現在実行中の Function オブジェクトに渡された arguments オブジェクトを返します。

callee プロパティ

指定した **Function** オブジェクトの本体である、実行される **Function** オブジェクトを返します。

caller プロパティ

現在の関数を呼び出した関数への参照を返します。

constructor プロパティ

オブジェクトを作成する関数を指定します。

description プロパティ

特定のエラーと関連付けられたエラーを説明する文字列を設定するか、または返します。

E プロパティ

自然対数の底を表す数理定数 *e* を返します。

global プロパティ

正規表現で使用する global フラグ (**g**) の状態を表すブール値を返します。

ignoreCase プロパティ

正規表現で使用する ignoreCase フラグ (**i**) の状態を表すブール値を返します。

index プロパティ

検索文字列と一致する最初の文字について、対象文字列内の先頭からの位置を返します。

Infinity プロパティ

Number.POSITIVE_INFINITY の初期値を返します。

input プロパティ ($_)

正規表現検索の対象となった文字列を返します。

lastIndex プロパティ

検索文字列内で次に文字が一致する位置を返します。

lastMatch プロパティ ($&)

正規表現による検索で最後に一致した文字を返します。

lastParen プロパティ ($+)

正規表現による検索で、かっこで囲まれた最後のサブマッチを返します。

leftContext プロパティ ($`)

検索した文字列の先頭から、最後に一致した先頭の前までの文字を返します。

length プロパティ (arguments)

関数を呼び出すときに実際に渡された引数の数を返します。

length プロパティ (Array)

配列内で定義されている最後の要素のインデックスより 1 だけ大きい整数値を示します。

length プロパティ (Function)

関数に定義されている引数の数を返します。

length プロパティ (String)

String オブジェクトの長さを返します。

LN10 プロパティ

10 の自然対数の値を返します。

LN2 プロパティ

2 の自然対数の値を返します。

LOG10E プロパティ

10 を底とする *e* (自然対数の底) の対数の値を返します。

LOG2E プロパティ

2 を底とする *e* (自然対数の底) の対数の値を返します。

MAX_VALUE プロパティ

JScript で表現できる最大の数値を返します。この値は、およそ 1.79E+308 です。

message プロパティ

エラー メッセージの文字列を返します。

MIN_VALUE プロパティ

JScript で表現できる最も 0 に近い数値を返します。この値は、およそ 5.00E-324 です。

multiline プロパティ

正規表現で使用する multiline フラグ (**m**) の状態を表すブール値を返します。

name プロパティ

エラー名を返します。

NaN プロパティ

算術式が数値以外の値を返したことを表す特別な値です。

NaN プロパティ (Global)

式が数値ではないことを示す特別な値 (**NaN)** を返します。

NEGATIVE_INFINITY プロパティ

JScript で表現できる最も小さい負の値 (**-Number.MAX_VALUE**) よりも小さい値を返します。

number プロパティ

JScript で表現できる最も小さい負の値 (**-Number.MAX_VALUE**) よりも小さい値を返します。

PI プロパティ

円の円周を直径で割った値 (円周率) を返します。この値は、およそ 3.141592653589793 です。

POSITIVE_INFINITY プロパティ

JScript で表現できる最も大きい値 (**Number.MAX_VALUE**) よりも大きい値を返します。

propertyIsEnumerable プロパティ

指定したプロパティがオブジェクトの一部であるかどうか、および加算できるかどうかを表すブール値を返します。

prototype プロパティ

指定されたオブジェクトのクラスのプロトタイプへの参照を返します。

rightContext プロパティ ($')

検索した文字列内で、最後に一致した位置から最後までの文字を返します。

source プロパティ

正規表現パターンのテキストを返します。

SQRT1_2 プロパティ

2 の平方根の 1/2 の値を返します。

SQRT2 プロパティ

2 の平方根の値を返します。

undefined プロパティ

undefined の初期値を返します。

関連するセクション

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

.NET Framework リファレンス

.NET Framework クラス ライブラリおよびその他の基本要素の構文と構造を説明するトピックへのリンクを示します。

0...n プロパティ

実行した関数の **arguments** プロパティで返される **arguments** オブジェクトから、各引数の値を返します。

```
[function.]arguments[[n]]
```

引数
function

省略可能です。現在実行中の **Function** オブジェクトの名前を指定します。

n

必ず指定します。0 ～ **arguments.length-1** の範囲で正の整数を指定します。0 が 1 番目の引数に対応し、**arguments.length-1** が最後の引数に対応します。

解説

0 ～ n の各プロパティが返す値は、実行している関数に渡す値です。**arguments** オブジェクトは配列ではありませんが、**arguments** オブジェクトを構成する各引数は、配列要素にアクセスする方法と同じ方法でアクセスされます。

メモ :

arguments オブジェクトは、高速モードで実行されている場合は利用できません。JScript の既定のモードは高速モードです。**arguments** オブジェクトを使用するプログラムをコマンド ラインからコンパイルするには、**/fast-** を使用して fast オプションをオフにする必要があります。ASP.NET で fast オプションをオフにするのは安全ではありません。スレッドに関する問題が発生する場合があります。詳細については、「arguments オブジェクト」を参照してください。

使用例

arguments オブジェクトの **0 . . .** n の各プロパティの使用例を次に示します。

```
function argTest(){
   var s = "";
   s += "The individual arguments are:\n"
   for (var n=0; n< arguments.length; n++){
      s += "argument " + n;
      s += " is " + argTest.arguments[n] + "\n";
   }
   return(s);
}
print(argTest(1, 2, "hello", new Date()));
```

/fast- オプションを指定してコンパイルすると、このプログラムの出力は次のようになります。

```
The individual arguments are:
argument 0 is 1
argument 1 is 2
argument 2 is hello
argument 3 is Sat Jan 1 00:00:00 PST 2000
```

必要条件

Version 5.5

対象

arguments オブジェクト| Function オブジェクト

参照
その他の技術情報
プロパティ (JScript)

JScript 8.0

$1...$9 プロパティ

パターン一致で検出された、記憶されている最新の 9 つの部分を返します。読み取り専用です。

```
RegExp.$n
```

引数

RegExp

必ず指定します。グローバルな **RegExp** オブジェクトを指定します。

n

必ず指定します。1 ～ 9 の整数を指定します。

解説

$1...$9 プロパティの値は、かっこで囲まれたパターンの検索が成功するたびに変更されます。正規表現パターン内に指定できるかっこで囲んだ部分の数に制限はありませんが、保存されるのは最後に見つかった 9 つだけです。

メモ：

RegExp オブジェクトのプロパティは、高速モードで実行されている場合は利用できません。JScript の既定のモードは高速モードです。これらのプロパティを使用するプログラムをコマンド ラインからコンパイルするには、**/fast-** を使用して fast オプションをオフにする必要があります。ASP.NET で fast オプションをオフにするのは安全ではありません。スレッドに関する問題が発生する場合があります。

使用例

$1...$9 プロパティの使用例を次に示します。

```
var s : String;
var re : RegExp = new RegExp("d(b+)(d)","ig");
var str : String = "cdbBdbsbdbdz";
var arr : Array = re.exec(str);
s = "$1 contains: " + RegExp.$1 + "\n";
s += "$2 contains: " + RegExp.$2 + "\n";
s += "$3 contains: " + RegExp.$3;
print(s);
```

/fast- オプションを指定してコンパイルすると、このプログラムの出力は次のようになります。

```
$1 contains: bB
$2 contains: d
$3 contains:
```

必要条件

Version 1

対象

RegExp オブジェクト

参照

概念

正規表現の構文

arguments プロパティ

現在実行中の **Function** オブジェクトに渡された **arguments** オブジェクトを格納した配列を返します。

```
[function.]arguments
```

引数
function

省略可能です。現在実行中の **Function** オブジェクトの名前を指定します。

解説

arguments プロパティを使用すると、関数に異なる数の引数を扱わせることができます。**arguments** オブジェクトの **length** プロパティには、関数に渡された引数の数が設定されています。**arguments** オブジェクトの各引数には、各要素にアクセスするのと同じ方法でアクセスできます。

メモ：

arguments オブジェクトは、高速モードで実行されている場合は利用できません。JScript の既定のモードは高速モードです。**arguments** オブジェクトを使用するプログラムをコマンド ラインからコンパイルするには、**/fast-** を使用して fast オプションをオフにする必要があります。ASP.NET で fast オプションをオフにするのは安全ではありません。スレッドに関する問題が発生する場合があります。詳細については、「arguments オブジェクト」を参照してください。

使用例

arguments プロパティの使用例を次に示します。

```
function argTest(){
   var s = "";
   s += "The individual arguments are:\n"
   for (var n=0; n< arguments.length; n++){
      s += "argument " + n;
      s += " is " + argTest.arguments[n] + "\n";
   }
   return(s);
}
print(argTest(1, 2, "hello", new Date()));
```

/fast- オプションを指定してコンパイルすると、このプログラムの出力は次のようになります。

```
The individual arguments are:
argument 0 is 1
argument 1 is 2
argument 2 is hello
argument 3 is Sat Jan 1 00:00:00 PST 2000
```

必要条件

Version 2

対象

Function オブジェクト

参照
関連項目
arguments オブジェクト
function ステートメント

callee プロパティ

指定した **Function** オブジェクトの本体である実行中の **Function** オブジェクトを返します。

```
[function.]arguments.callee
```

引数
function

省略可能です。現在実行中の **Function** オブジェクトの名前を指定します。

解説

callee プロパティは **arguments** オブジェクトのメンバで、対応する関数が実行されているときにだけ使用できます。

callee プロパティの初期値は、実行中の **Function** オブジェクトになります。したがって、無名関数を再帰的に使用できます。

メモ：

arguments オブジェクトは、高速モードで実行されている場合は利用できません。JScript の既定のモードは高速モードです。**arguments** オブジェクトを使用するプログラムをコマンド ラインからコンパイルするには、**/fast-** を使用して fast オプションをオフにする必要があります。ASP.NET で fast オプションをオフにするのは安全ではありません。スレッドに関する問題が発生する場合があります。詳細については、「arguments オブジェクト」を参照してください。

使用例

callee プロパティの使用例を次に示します。

```
function factorial(n) {
   if (n <= 0)
      return 1;
   else
      return n * arguments.callee(n - 1)
}
print(factorial(3));
```

/fast- オプションを指定してコンパイルすると、このプログラムの出力は次のようになります。

```
6
```

必要条件

Version 5.5

対象：

arguments オブジェクト| Function オブジェクト

参照
関連項目
function ステートメント

caller プロパティ

現在の関数を呼び出した関数への参照を返します。

```
function.caller
```

引数
function

必ず指定します。現在実行中の **Function** オブジェクトの名前を指定します。

解説

caller プロパティは、実行中の関数にしか定義されません。また、JScript プログラムのトップ レベルから呼び出された関数の **caller** プロパティは、**null** が設定されます。

文字列コンテキストの中で **caller** プロパティを使用すると、*functionname*.**toString** と同じ結果になります。つまり、関数の逆コンパイルされたテキストが表示されます。

メモ :

caller プロパティは、高速モードで実行されている場合は利用できません。JScript の既定のモードは高速モードです。**caller** プロパティを使用するプログラムをコマンド ラインからコンパイルするには、**/fast-** を使用して fast オプションをオフにする必要があります。ASP.NET で fast オプションをオフにするのは安全ではありません。スレッドに関する問題が発生する場合があります。

使用例

caller プロパティの使用例を次に示します。

```
function callLevel(){
   if (callLevel.caller == null)
      print("callLevel was called from the top level.");
   else {
      print("callLevel was called by:");
      print(callLevel.caller);
   }
}
function testCall() {
   callLevel()
}
// Call callLevel directly.
callLevel();
// Call callLevel indirectly.
testCall();
```

/fast- オプションを指定してコンパイルすると、このプログラムの出力は次のようになります。

```
callLevel was called from the top level.
callLevel was called by:
function testCall() {
   callLevel()
}
```

必要条件

Version 2

対象

arguments オブジェクト| Function オブジェクト

参照
関連項目

function ステートメント

constructor プロパティ

オブジェクトを作成する関数を指定します。

```
object.constructor
```

引数
object

必ず指定します。オブジェクトまたは関数の名前を指定します。

解説

constructor プロパティは、プロトタイプを持つあらゆるオブジェクトのプロトタイプのメンバです。これには、**arguments**、**Enumerator**、**Error**、**Global**、**Math**、**RegExp**、**Regular Expression**、および **VBArray** オブジェクト以外の、JScript の組み込みオブジェクトがすべて含まれます。**constructor** プロパティには、特定のオブジェクトのインスタンスを作成する関数への参照が格納されます。

クラス ベースのオブジェクトには、**constructor** プロパティはありません。

使用例

constructor プロパティの使用例を次に示します。

```
function testObject(ob) {
   if (ob.constructor == String)
      print("Object is a String.");
   else if (ob.constructor == MyFunc)
      print("Object is constructed from MyFunc.");
   else
      print("Object is neither a String or constructed from MyFunc.");
}
// A constructor function.
function MyFunc() {
   // Body of function.
}

var x = new String("Hi");
testObject(x)
var y = new MyFunc;
testObject(y);
```

このプログラムの出力は次のようになります。

```
Object is a String.
Object is constructed from MyFunc.
```

必要条件

Version 2

対象：

Array オブジェクト| Boolean オブジェクト| Date オブジェクト| Function オブジェクト| Number オブジェクト| Object オブジェクト| String オブジェクト

参照
関連項目
prototype プロパティ

description プロパティ

特定のエラーと関連付けられたエラーを説明する文字列を設定するか、または返します。

```
object.description
```

引数
object

必ず指定します。**Error** オブジェクトのインスタンスを指定します。

解説

description プロパティは、特定のエラーに関連付けられたエラー メッセージを格納する文字列です。このプロパティを使用して、スクリプトが処理できないエラーをユーザーに通知します。

description プロパティおよび **message** プロパティは、同じメッセージを参照します。**description** プロパティは下位互換性を提供し、**message** プロパティは ECMA 規格に準拠しています。

使用例

例外を発生させ、エラーの説明を表示する例を次に示します。

```
function getAge(age) {
   if(age < 0)
      throw new Error("An age cannot be negative.")
   print("Age is "+age+".");
}

// Pass the getAge an invalid argument.
try {
   getAge(-5);
} catch(e) {
   print(e.description);
}
```

このコードの出力は次のようになります。

```
An age cannot be negative.
```

必要条件

Version 5

対象：

Error オブジェクト

参照
関連項目
number プロパティ
message プロパティ (JScript)
name プロパティ

E プロパティ

自然対数の底を表す数理定数 *e* を返します。

```
Math.E
```

引数

Math

必ず指定します。グローバルな **Math** オブジェクトを指定します。

解説

E プロパティの値は、約 2.718 です。

必要条件

Version 1

対象

Math オブジェクト

参照

関連項目

exp メソッド

global プロパティ

正規表現で使用する global フラグ (**g**) の状態を表すブール値を返します。

```
rgExp.global
```

引数

rgExp

必ず指定します。**Regular Expression** オブジェクトのインスタンスを指定します。

解説

global プロパティは読み取り専用で、正規表現のグローバル フラグが設定されているときは **true** を返し、設定されていないときは **false** を返します。既定値は **false** です。

global フラグを使用すると、検索文字列内のパターンの最初の 1 候補だけではなく、すべての候補を検索できます。これはグローバル マッチングとも呼びます。

使用例

global プロパティの使用例を次に示します。

```
function RegExpPropDemo(re : RegExp) {
   print("Regular expression: " + re);
   print("global:     " + re.global);
   print("ignoreCase: " + re.ignoreCase);
   print("multiline:  " + re.multiline);
   print();
};

// Some regular expression to test the function.
var re1 : RegExp = new RegExp("the","i");  // Use the constructor.
var re2 = /\w+/gm;                         // Use a literal.
RegExpPropDemo(re1);
RegExpPropDemo(re2);
RegExpPropDemo(/^\s*$/im);
```

このプログラムの出力は次のようになります。

```
Regular expression: /the/i
global:     false
ignoreCase: true
multiline:  false

Regular expression: /\w+/gm
global:     true
ignoreCase: false
multiline:  true

Regular expression: /^\s*$/im
global:     false
ignoreCase: true
multiline:  true
```

必要条件

Version 5.5

対象：

Regular Expression オブジェクト

参照

関連項目
ignoreCase プロパティ
multiline プロパティ
概念
正規表現の構文

ignoreCase プロパティ

正規表現で使用する ignoreCase フラグ (**i**) の状態を表すブール値を返します。

```
rgExp.ignoreCase
```

引数
rgExp

必ず指定します。**Regular Expression** オブジェクトのインスタンスを指定します。

解説

ignoreCase プロパティは読み取り専用で、正規表現の ignoreCase フラグが設定されているときは **true** を返し、設定されていないときは **false** を返します。既定値は **false** です。

ignoreCase フラグを使用すると、検索文字列内のパターンをマッチングさせるときに、大文字小文字の区別をしません。

使用例

ignoreCase プロパティの使用例を次に示します。

```
function RegExpPropDemo(re : RegExp) {
   print("Regular expression: " + re);
   print("global:     " + re.global);
   print("ignoreCase: " + re.ignoreCase);
   print("multiline:  " + re.multiline);
   print();
};

// Some regular expression to test the function.
var re1 : RegExp = new RegExp("the","i");  // Use the constructor.
var re2 = /\w+/gm;                         // Use a literal.
RegExpPropDemo(re1);
RegExpPropDemo(re2);
RegExpPropDemo(/^\s*$/im);
```

このプログラムの出力は次のようになります。

```
Regular expression: /the/i
global:     false
ignoreCase: true
multiline:  false

Regular expression: /\w+/gm
global:     true
ignoreCase: false
multiline:  true

Regular expression: /^\s*$/im
global:     false
ignoreCase: true
multiline:  true
```

必要条件

Version 5.5

対象

Regular Expression オブジェクト

参照
関連項目

index プロパティ

検索文字列と一致する最初の文字について、対象文字列内の先頭からの位置を返します。

```
{RegExp | reArray}.index
```

引数

RegExp

必ず指定します。グローバルな **RegExp** オブジェクトを指定します。

reArray

必ず指定します。**Regular Expression** オブジェクトの **exec** メソッドによって返される配列を指定します。

解説

index プロパティの値は、0 から始まるインデックス番号です。

RegExp.index プロパティの初期値は –1 です。プロパティの値は読み取り専用で、検索が成功するたびに変更されます。

メモ :

RegExp オブジェクトのプロパティは、高速モードで実行されている場合は利用できません。JScript の既定のモードは高速モードです。これらのプロパティを使用するプログラムをコマンド ラインからコンパイルするには、**/fast-** を使用して fast オプションをオフにする必要があります。ASP.NET で fast オプションをオフにするのは安全ではありません。スレッドに関する問題が発生する場合があります。

使用例

index プロパティの使用例を次に示します。この関数は、文字列の検索を繰り返し、文字列内にある各文字の **index** 値および **lastIndex** 値を出力します。

```
var src : String = "The rain in Spain falls mainly in the plain.";
var re : RegExp = /\w+/g;
var arr : Array;
while ((arr = re.exec(src)) != null)
   print(arr.index + "-" + arr.lastIndex + "\t" + arr);
```

このプログラムの出力は次のようになります。

```
0-3     The
4-8     rain
9-11    in
12-17   Spain
18-23   falls
24-30   mainly
31-33   in
34-37   the
38-43   plain
```

必要条件

Version 3

対象

RegExp オブジェクト

参照

関連項目

exec メソッド

概念

正規表現の構文

Infinity プロパティ

Number.POSITIVE_INFINITY の値を返します。

```
Infinity
```

解説

Infinity プロパティは **Global** オブジェクトのメンバであり、スクリプト エンジンが初期化された時点で使用可能になります。

必要条件

Version 3

対象：

Global オブジェクト

参照

関連項目

POSITIVE_INFINITY プロパティ
NEGATIVE_INFINITY プロパティ

input プロパティ ($_)

正規表現検索の対象となった文字列を返します。

```
//Syntax 1
{RegExp | reArray}.input

//Syntax 2
RegExp.$_
//The $_ property may be used as shorthand for the input property
//for the RegExp object.
```

引数

RegExp

必ず指定します。グローバルな **RegExp** オブジェクトを指定します。

reArray

必ず指定します。**Regular Expression** オブジェクトの **exec** メソッドによって返される配列を指定します。

解説

input プロパティの値は、正規表現検索の対象となった文字列です。

RegExp.input プロパティの初期値は空の文字列 "" です。値は読み取り専用で、検索が成功するたびに変更されます。

メモ :

RegExp オブジェクトのプロパティは、高速モードで実行されている場合は利用できません。JScript の既定のモードは高速モードです。これらのプロパティを使用するプログラムをコマンド ラインからコンパイルするには、**/fast-** を使用して fast オプションをオフにする必要があります。ASP.NET で fast オプションをオフにするのは安全ではありません。スレッドに関する問題が発生する場合があります。

使用例

input プロパティの使用例を次に示します。

```
var str : String = "A test string.";
var re : RegExp = new RegExp("\\w+","ig");
var arr : Array = re.exec(str);
print("The string used for the match was: " + arr.input);
```

このプログラムの出力は次のようになります。

```
The string used for the match was: A test string.
```

必要条件

Version 3

対象

RegExp オブジェクト

参照

関連項目

exec メソッド

概念

正規表現の構文

lastIndex プロパティ

検索文字列内で次に文字が一致する位置を返します。

```
{RegExp | reArray}.lastIndex
```

引数

RegExp

必ず指定します。グローバルな **RegExp** オブジェクトを指定します。

reArray

必ず指定します。**Regular Expression** オブジェクトの **exec** メソッドによって返される配列を指定します。

解説

lastIndex プロパティの値は、文字列の先頭位置の 0 を基にしています。初期値は –1 です。この値は、検索が成功するたびに変更されます。

RegExp オブジェクトの **lastIndex** プロパティは、**RegExp** オブジェクトの **exec** メソッドと **test** メソッド、および **String** オブジェクトの **match**、**replace**、**split** の各メソッドによって変更されます。

lastIndex プロパティの値には、次の規則が適用されます。

- 一致する文字列がない場合は、-1 に設定されます。
- **lastIndex** プロパティの値を文字列よりも長く設定してから **test** メソッドまたは **exec** メソッドを実行すると、メソッドの実行は失敗し、**lastIndex** プロパティに -1 が設定されます。
- **lastIndex** プロパティの値が文字列の長さと等しい場合、パターンが空の文字列であれば、正規表現パターンが一致します。それ以外の場合、検索は失敗し、**lastIndex** プロパティに -1 が再設定されます。
- 上記以外の場合、**lastIndex** プロパティは、パターンに一致する最後に見つかった文字列の直後の位置を示す値が設定されます。

RegExp.lastIndex プロパティの初期値は –1 です。プロパティの値は読み取り専用で、検索が成功するたびに変更されます。

メモ :

RegExp オブジェクトのプロパティは、高速モードで実行されている場合は利用できません。JScript の既定のモードは高速モードです。これらのプロパティを使用するプログラムをコマンド ラインからコンパイルするには、**/fast-** を使用して fast オプションをオフにする必要があります。ASP.NET で fast オプションをオフにするのは安全ではありません。スレッドに関する問題が発生する場合があります。

使用例

lastIndex プロパティの使用例を次に示します。この関数は、文字列の検索を繰り返し、文字列内にある各文字の **index** 値および **lastIndex** 値を出力します。

```
var src : String = "The rain in Spain falls mainly in the plain.";
var re : RegExp = /\w+/g;
var arr : Array;
while ((arr = re.exec(src)) != null)
   print(arr.index + "-" + arr.lastIndex + "\t" + arr);
```

このプログラムの出力は次のようになります。

```
0-3     The
4-8     rain
9-11    in
12-17   Spain
18-23   falls
24-30   mainly
31-33   in
34-37   the
```

```
38-43   plain
```

必要条件

Version 3

対象：

RegExp オブジェクト

参照

関連項目

exec メソッド

概念

正規表現の構文

lastMatch プロパティ ($&)

正規表現による検索で最後に一致した文字を返します。読み取り専用です。

```
RegExp.lastMatch
```

引数

RegExp

必ず指定します。グローバルな **RegExp** オブジェクトを指定します。

解説

lastMatch プロパティの初期値は空の文字列です。**lastMatch** プロパティの値は、検索が成功するたびに変更されます。

メモ :

RegExp オブジェクトのプロパティは、高速モードで実行されている場合は利用できません。JScript の既定のモードは高速モードです。これらのプロパティを使用するプログラムをコマンド ラインからコンパイルするには、**/fast-** を使用して fast オプションをオフにする必要があります。ASP.NET で fast オプションをオフにするのは安全ではありません。スレッドに関する問題が発生する場合があります。

lastMatch プロパティの短縮形は **$&** です。式 **RegExp["$&"]** と式 **RegExp.lastMatch** は同義です。

使用例

lastMatch プロパティの使用例を次に示します。

```
var s;                                //Declare variable.
var re = new RegExp("d(b+)(d)","ig"); //Regular expression pattern.
var str = "cdbBdbsbdbdz";             //String to be searched.
var arr = re.exec(str);               //Perform the search.
s = "$1 returns: " + RegExp.$1 + "\n";
s += "$2 returns: " + RegExp.$2 + "\n";
s += "$3 returns: " + RegExp.$3 + "\n";
s += "input returns : " + RegExp.input + "\n";
s += "lastMatch returns: " + RegExp.lastMatch + "\n";
s += "leftContext returns: " + RegExp.leftContext + "\n";
s += "rightContext returns: " + RegExp.rightContext + "\n";
s += "lastParen returns: " + RegExp.lastParen + "\n";
print(s);                             //Return results.
```

/fast- オプションを指定してコンパイルすると、このプログラムの出力は次のようになります。

```
$1 returns: bB
$2 returns: d
$3 returns:
input returns : cdbBdbsbdbdz
lastMatch returns: dbBd
leftContext returns: c
rightContext returns: bsbdbdz
lastParen returns: d
```

必要条件

Version 5.5

対象

RegExp オブジェクト

参照

関連項目

$1...$9 プロパティ

lastParen プロパティ ($+)

正規表現による検索で、最後にパターン化されたサブマッチがある場合に、それを返します。読み取り専用です。

```
RegExp.lastParen
```

引数
RegExp

必ず指定します。グローバルな **RegExp** オブジェクトを指定します。

解説

lastParen プロパティの初期値は空の文字列です。**lastParen** プロパティの値は、検索が成功するたびに変更されます。

メモ :

RegExp オブジェクトのプロパティは、高速モードで実行されている場合は利用できません。JScript の既定のモードは高速モードです。これらのプロパティを使用するプログラムをコマンド ラインからコンパイルするには、**/fast-** を使用して fast オプションをオフにする必要があります。ASP.NET で fast オプションをオフにするのは安全ではありません。スレッドに関する問題が発生する場合があります。

lastParen プロパティの短縮形は **$+** です。式 **RegExp["$+"]** と式 **RegExp.lastParen** は同義です。

使用例

lastParen プロパティの使用例を次に示します。

```
var s;                                //Declare variable.
var re = new RegExp("d(b+)(d)","ig"); //Regular expression pattern.
var str = "cdbBdbsbdbdz";             //String to be searched.
var arr = re.exec(str);               //Perform the search.
s = "$1 returns: " + RegExp.$1 + "\n";
s += "$2 returns: " + RegExp.$2 + "\n";
s += "$3 returns: " + RegExp.$3 + "\n";
s += "input returns : " + RegExp.input + "\n";
s += "lastMatch returns: " + RegExp.lastMatch + "\n";
s += "leftContext returns: " + RegExp.leftContext + "\n";
s += "rightContext returns: " + RegExp.rightContext + "\n";
s += "lastParen returns: " + RegExp.lastParen + "\n";
print(s);                            //Return results.
```

/fast- オプションを指定してコンパイルすると、このプログラムの出力は次のようになります。

```
$1 returns: bB
$2 returns: d
$3 returns:
input returns : cdbBdbsbdbdz
lastMatch returns: dbBd
leftContext returns: c
rightContext returns: bsbdbdz
lastParen returns: d
```

必要条件

Version 5.5

対象

RegExp オブジェクト

参照
関連項目
$1...$9 プロパティ

leftContext プロパティ ($`)

検索した文字列の先頭から、最後に一致した先頭の前までの文字を返します。読み取り専用です。

```
RegExp.leftContext
```

引数
RegExp

必ず指定します。グローバルな **RegExp** オブジェクトを指定します。

解説

leftContext プロパティの初期値は空の文字列です。**leftContext** プロパティの値は、検索が成功するたびに変更されます。

メモ :

RegExp オブジェクトのプロパティは、高速モードで実行されている場合は利用できません。JScript の既定のモードは高速モードです。これらのプロパティを使用するプログラムをコマンド ラインからコンパイルするには、**/fast-** を使用して fast オプションをオフにする必要があります。ASP.NET で fast オプションをオフにするのは安全ではありません。スレッドに関する問題が発生する場合があります。

leftContext プロパティの短縮形は **$`** です。式 **RegExp["$`"]** と式 **RegExp.leftContext** は同義です。

使用例

leftContext プロパティの使用例を次に示します。

```
var s;                                //Declare variable.
var re = new RegExp("d(b+)(d)","ig"); //Regular expression pattern.
var str = "cdbBdbsbdbdz";             //String to be searched.
var arr = re.exec(str);               //Perform the search.
s = "$1 returns: " + RegExp.$1 + "\n";
s += "$2 returns: " + RegExp.$2 + "\n";
s += "$3 returns: " + RegExp.$3 + "\n";
s += "input returns : " + RegExp.input + "\n";
s += "lastMatch returns: " + RegExp.lastMatch + "\n";
s += "leftContext returns: " + RegExp.leftContext + "\n";
s += "rightContext returns: " + RegExp.rightContext + "\n";
s += "lastParen returns: " + RegExp.lastParen + "\n";
print(s);                            //Return results.
```

/fast- オプションを指定してコンパイルすると、このプログラムの出力は次のようになります。

```
$1 returns: bB
$2 returns: d
$3 returns:
input returns : cdbBdbsbdbdz
lastMatch returns: dbBd
leftContext returns: c
rightContext returns: bsbdbdz
lastParen returns: d
```

必要条件

Version 5.5

対象

RegExp オブジェクト

参照
関連項目
$1...$9 プロパティ

length プロパティ (arguments)

関数を呼び出すときに実際に渡された引数の数を返します。

```
[function.]arguments.length
```

引数
function

省略可能です。現在実行中の **Function** オブジェクトの名前を指定します。

解説

arguments オブジェクトの **length** プロパティは、スクリプト エンジンによって、関数が実行されるときにオブジェクトに渡される引数の実際の数に初期化されます。

使用例

arguments オブジェクトの **length** プロパティの使用例を次に示します。

```
function argTest(a, b) : String {
   var i : int;
   var s : String = "The argTest function expected ";
   var numargs : int = arguments.length; // Get number of arguments passed.
   var expargs : int = argTest.length;   // Get number of arguments expected.
   if (expargs < 2)
      s += expargs + " argument. ";
   else
      s += expargs + " arguments. ";
   if (numargs < 2)
      s += numargs + " was passed.";
   else
      s += numargs + " were passed.";
   s += "\n"
   for (i =0 ; i < numargs; i++){        // Get argument contents.
      s += "  Arg " + i + " = " + arguments[i] + "\n";
   }
   return(s);                            // Return list of arguments.
}

print(argTest(42));
print(argTest(new Date(1999,8,7),"Sam",Math.PI));
```

/fast- オプションを指定してコンパイルすると、このプログラムの出力は次のようになります。

```
The argTest function expected 2 arguments. 1 was passed.
  Arg 0 = 42

The argTest function expected 2 arguments. 3 were passed.
  Arg 0 = Tue Sep 7 00:00:00 PDT 1999
  Arg 1 = Sam
  Arg 2 = 3.141592653589793
```

必要条件

Version 5.5

対象

arguments オブジェクト

参照
関連項目
arguments プロパティ

length プロパティ (Array)
length プロパティ (String)

length プロパティ (Array)

配列内で定義されている最後の要素のインデックスより 1 だけ大きい整数値を示します。

```
arrayObj.length
```

引数
arrayObj

必ず指定します。任意の **Array** オブジェクトを指定します。

解説

JScript 配列内の要素のインデックスは必ずしも連続している必要はありません。したがって、**length** プロパティの値が配列内の要素の数と一致するとは限りません。

length プロパティに代入される値が既存の値よりも小さい場合、配列は切り詰められ、**length** プロパティに新しく設定した値以上のインデックスを持つ要素はすべて失われます。

前回よりも大きい値を **length** プロパティに設定すると、配列は形式的に拡張されますが、新しい要素は作成されません。

使用例

length プロパティの使用例を次に示します。配列を宣言し、2 つの要素を追加します。配列の最大のインデックスは 6 であるため、length の値は 7 になります。

```
var my_array : Array = new Array();
my_array[2] = "Test";
my_array[6] = "Another Test";
print(my_array.length); // Prints 7.
```

必要条件

Version 2

対象：

Array オブジェクト

参照
関連項目
length プロパティ (Function)
length プロパティ (String)

length プロパティ (Function)

関数に定義されている引数の数を返します。

```
function.length
```

引数
function

必ず指定します。現在実行中の **Function** オブジェクトの名前を指定します。

解説

関数のインスタンスが作成されると、スクリプト エンジンによって、関数の定義に含まれている引数の数が関数の **length** プロパティに初期設定されます。

length プロパティと異なる数の引数を指定して関数を呼び出した場合に何が起こるかは、各関数に依存します。

使用例

次のコードは、**length** プロパティの使用例です。

```
function argTest(a, b) : String {
   var s : String = "The argTest function expected " ;
   var expargs : int = argTest.length;
   s += expargs;
   if (expargs < 2)
      s += " argument.";
   else
      s += " arguments.";
   return(s);
}
// Display the function output.
print(argTest(42,"Hello"));
```

このプログラムの出力は次のようになります。

```
The argTest function expected 2 arguments.
```

必要条件

Version 2

対象：

Function オブジェクト

参照
関連項目
arguments プロパティ
length プロパティ (Array)
length プロパティ (String)

JScript 8.0

length プロパティ (String)

文字列の長さを返します。

```
str.length
```

引数
str

必ず指定します。リテラル文字列または **String** オブジェクトの名前を指定します。

解説

length プロパティには、**String** オブジェクトの文字数を示す整数が格納されています。**String** オブジェクト内の末尾の文字のインデックス番号は、**length** - 1 となります。

必要条件

Version 1

対象：

String オブジェクト

参照
関連項目
length プロパティ (Array)
length プロパティ (Function)

LN10 プロパティ

10 の自然対数の値を返します。

```
Math.LN10
```

引数

Math

必ず指定します。グローバルな **Math** オブジェクトを指定します。

解説

LN10 プロパティは、約 2.302 の値を持つ定数です。

必要条件

Version 1

対象

Math オブジェクト

参照

関連項目

length プロパティ (Array)
length プロパティ (Function)

LN2 プロパティ

2 の自然対数の値を返します。

```
Math.LN2
```

引数

Math

必ず指定します。グローバルな **Math** オブジェクトを指定します。

解説

LN2 プロパティは、約 0.693 の値を持つ定数です。

必要条件

Version 1

対象：

Math オブジェクト

参照

関連項目

length プロパティ (Array)
length プロパティ (Function)

LOG10E プロパティ

10 を底とする *e* (自然対数の底) の対数の値を返します。

```
Math.LOG10E
```

引数

Math

必ず指定します。グローバルな **Math** オブジェクトを指定します。

解説

LOG10E プロパティは、約 0.434 の値を持つ定数です。

必要条件

Version 1

対象

Math オブジェクト

参照

関連項目

length プロパティ (Array)
length プロパティ (Function)

LOG2E プロパティ

2 を底とする *e* (自然対数の底) の対数の値を返します。

```
Math.LOG2E
```

引数

Math

必ず指定します。グローバルな **Math** オブジェクトを指定します。

解説

LOG2E プロパティは、約 1.442 の値を持つ定数です。

必要条件

Version 1

対象

Math オブジェクト

参照

関連項目

length プロパティ (Array)
length プロパティ (Function)

MAX_VALUE プロパティ

JScript で表現できる最大の数値を返します。この値は、およそ 1.79E+308 です。

```
Number.MAX_VALUE
```

引数

Number

必ず指定します。グローバルな **Number** オブジェクトを指定します。

解説

MAX_VALUE プロパティの値を取得するために、あらかじめ **Number** オブジェクトを作成しておく必要はありません。

必要条件

Version 2

対象

Number オブジェクト

参照

関連項目

MIN_VALUE プロパティ
NaN プロパティ
NEGATIVE_INFINITY プロパティ
POSITIVE_INFINITY プロパティ
toString メソッド

message プロパティ (JScript)

エラー メッセージの文字列を返します。

```
errorObj.message
```

引数
errorObj

必ず指定します。**Error** オブジェクトのインスタンスを指定します。

解説

message プロパティは、特定のエラーに関連付けられたエラー メッセージを格納する文字列です。このプロパティを使用して、エラー処理できない場合やエラー処理したくない場合に、ユーザーに注意を促します。

description プロパティおよび **message** プロパティは、同じメッセージを参照します。**description** プロパティは下位互換性を提供し、**message** プロパティは ECMA 規格に準拠しています。

使用例

例外を発生させ、エラーのメッセージを表示する例を次に示します。

```
function getAge(age) {
   if(age < 0)
      throw new Error("An age cannot be negative.")
   print("Age is "+age+".");
}

// Pass the getAge an invalid argument.
try {
   getAge(-5);
} catch(e) {
   print(e.message);
}
```

このコードの出力は次のようになります。

```
An age cannot be negative.
```

必要条件

Version 5.5

対象

Error オブジェクト

参照
関連項目
description プロパティ
name プロパティ

MIN_VALUE プロパティ

JScript で表現できる最も 0 に近い数値を返します。この値は、およそ 5.00E-324 です。

```
Number.MIN_VALUE
```

引数

Number

必ず指定します。グローバルな **Number** オブジェクトを指定します。

解説

MIN_VALUE プロパティの値を取得するために、あらかじめ **Number** オブジェクトを作成しておく必要はありません。

必要条件

Version 2

対象

Number オブジェクト

参照

関連項目

MAX_VALUE プロパティ
NaN プロパティ
NEGATIVE_INFINITY プロパティ
POSITIVE_INFINITY プロパティ
toString メソッド

multiline プロパティ

正規表現で使用する multiline フラグ (**m**) の状態を表すブール値を返します。

```
rgExp.multiline
```

引数
rgExp

必ず指定します。**Regular Expression** オブジェクトのインスタンスを指定します。

解説

multiline プロパティは読み取り専用で、正規表現の複数行フラグが設定されているときは **true** を返し、設定されていないときは **false** を返します。**m** フラグを使用した正規表現オブジェクトを作成すると、**multiline** プロパティが **true** になります。既定値は **false** です。

multiline が **false** のとき、"^" で文字列の先頭位置に一致し、"$" で文字列の最終位置に一致します。**multiline** が **true** の場合は、"^" で "\n" または "\r" の直後を含む文字列の先頭位置に一致し、"$" で "\n" または "\r" の直前を含む文字列の最終位置に一致します。

使用例

multiline プロパティの使用例を次に示します。

```
function RegExpPropDemo(re : RegExp) {
   print("Regular expression: " + re);
   print("global:     " + re.global);
   print("ignoreCase: " + re.ignoreCase);
   print("multiline:  " + re.multiline);
   print();
};

// Some regular expression to test the function.
var re1 : RegExp = new RegExp("the","i");  // Use the constructor.
var re2 = /\w+/gm;                         // Use a literal.
RegExpPropDemo(re1);
RegExpPropDemo(re2);
RegExpPropDemo(/^\s*$/im);
```

このプログラムの出力は次のようになります。

```
Regular expression: /the/i
global:     false
ignoreCase: true
multiline:  false

Regular expression: /\w+/gm
global:     true
ignoreCase: false
multiline:  true

Regular expression: /^\s*$/im
global:     false
ignoreCase: true
multiline:  true
```

必要条件

Version 5.5

対象

Regular Expression オブジェクト

参照

関連項目
global プロパティ
ignoreCase プロパティ
概念
正規表現の構文

name プロパティ

エラー名を返します。

```
errorObj.name
```

引数
errorObj

必ず指定します。**Error** オブジェクトのインスタンスを指定します。

解説

name プロパティは、エラーの名前または例外種別を返します。実行時エラーが発生すると、次に示すネイティブの例外種別の 1 つが name プロパティに設定されます。

例外の種類	説明
Error	このエラーは、**Error** オブジェクト コンストラクタで作成されたユーザー定義エラーです。
ConversionError	オブジェクトを変換不可能なものに変換しようとしたときに、このエラーが発生します。
RangeError	関数に範囲外の引数を指定したときに、このエラーが発生します。たとえば、有効な正の整数でない長さの **Array** オブジェクトを作成しようとすると、このエラーが発生します。
ReferenceError	無効な参照が検出されたときに、このエラーが発生します。たとえば、あるはずの参照が **NULL** だったときに、このエラーが発生します。
RegExpError	正規表現でコンパイル エラーがあるときに、このエラーが発生します。ただし、正規表現が正常にコンパイルされた後は、このエラーは発生しません。たとえば、正規表現のパターンを宣言するときの構文が無効である場合や、フラグが **i**、**g**、または **m** 以外である場合、または同じフラグが複数個含まれる場合などに、このエラーが発生します。
SyntaxError	ソース テキストを解析して、そのソース テキストの構文が正しくないときに、このエラーが発生します。たとえば、**eval** 関数の呼び出しで無効なプログラム テキストを引数として指定したときに、このエラーが発生します。
TypeError	オペランドの実際の型が、既定の型と一致しないときに、このエラーが発生します。たとえば、関数の呼び出し対象がオブジェクトではない場合や、その呼び出しをサポートしていない場合にこのエラーが発生します。
URIError	無効な URI (Uniform Resource Indicator) が検出されたときに、このエラーが発生します。たとえば、エンコードまたはデコードされている文字列に無効な文字が見つかると、このエラーが発生します。

使用例

例外を発生させ、エラーおよびエラーの説明を表示する例を次に示します。

```
function getAge(age) {
   if(age < 0)
      throw new Error("An age cannot be negative.")
   print("Age is "+age+".");
}

// Pass the getAge an invalid argument.
try {
   getAge(-5);
} catch(e) {
```

```
    print(e.name);
    print(e.description);
}
```

このコードの出力は次のようになります。

```
Error
An age cannot be negative.
```

必要条件

Version 5.5

対象

Error オブジェクト

参照

関連項目

description プロパティ
message プロパティ (JScript)
number プロパティ

NaN プロパティ

NaN を返します。NaN は、数値が返されるはずの場合に数値ではない値が返されたことを表すための特殊な値です。

```
Number.NaN
```

引数
Number

必ず指定します。グローバルな **Number** オブジェクトを指定します。

解説

NaN プロパティの値を取得するために、あらかじめ **Number** オブジェクトを作成しておく必要はありません。

NaN は、それ自身を含むどのような値と比較しても等しくなることはありません。値が **NaN** かどうかを調べるには、**Global** オブジェクトの **isNaN** メソッドを使用します。

必要条件

Version 2

対象：

Number オブジェクト

参照
関連項目
isNaN メソッド
MAX_VALUE プロパティ
MIN_VALUE プロパティ
NEGATIVE_INFINITY プロパティ
POSITIVE_INFINITY プロパティ
toString メソッド

NaN プロパティ (Global)

式が数値ではないことを示す特別な値 (**NaN)** を返します。

```
NaN
```

解説

NaN プロパティ (not a number) は **Global** オブジェクトのメンバであり、スクリプト エンジンが初期化された時点で使用可能になります。

NaN は、それ自身を含むどのような値と比較しても等しくなることはありません。値が **NaN** かどうかを調べるには、**Global** オブジェクトの **isNaN** メソッドを使用します。

必要条件

Version 3

対象

Global オブジェクト

参照
関連項目
isNaN メソッド

NEGATIVE_INFINITY プロパティ

JScript で表現できる最も小さい負の値 (**-Number.MAX_VALUE**) よりも小さい値を返します。

```
Number.NEGATIVE_INFINITY
```

引数
Number

必ず指定します。グローバルな **Number** オブジェクトを指定します。

解説

NEGATIVE_INFINITY プロパティの値を取得するために、あらかじめ **Number** オブジェクトを作成しておく必要はありません。

JScript では、**NEGATIVE_INFINITY** 値を -Infinity として表示します。この値は、算術演算では無限大として機能します。

必要条件

Version 2

対象

Number オブジェクト

参照
関連項目
MAX_VALUE プロパティ
MIN_VALUE プロパティ
NaN プロパティ
POSITIVE_INFINITY プロパティ
toString メソッド

number プロパティ

特定のエラーと関連付けられた数値を設定したり、この数値を取得したりします。

```
object.number
```

引数
object

Error オブジェクトの任意のインスタンスを指定します。

解説

エラー番号は 32 ビット値です。上位の 16 ビット ワードは機能識別符号です。下位のワードは実際のエラー コードです。実際のエラー コードを読み取るには、**&** (ビットごとの And) 演算子を使用して、number プロパティと 16 進数の `0xFFFF` を組み合わせます。

使用例

例外を発生させ、エラー番号を表示する例を次に示します。

```
function getAge(age) {
   if(age < 0)
      throw new Error(100)
   print("Age is "+age+".");
}

// Pass the getAge an invalid argument.
try {
   getAge(-5);
} catch(e) {
// Extract the error code from the error number.
   print(e.number & 0xFFFF)
}
```

このコードの出力は次のようになります。

```
100
```

必要条件

Version 5

対象：

Error オブジェクト

参照
関連項目
description プロパティ
message プロパティ (JScript)
name プロパティ

PI プロパティ

数学定数πの値を返します。

```
Math.PI
```

引数

Math

必ず指定します。グローバルな **Math** オブジェクトを指定します。

解説

PI プロパティは、約 3.14159 の値を持つ定数です。円の円周を直径で割った値 (円周率) を表します。

必要条件

Version 1

対象

Math オブジェクト

参照

その他の技術情報

プロパティ (JScript)

POSITIVE_INFINITY プロパティ

JScript で表現できる最も大きい値 (**Number.MAX_VALUE**) よりも大きい値を返します。

```
Number.POSITIVE_INFINITY
```

引数
Number

必ず指定します。グローバルな **Number** オブジェクトを指定します。

解説

POSITIVE_INFINITY プロパティの値を取得するために、あらかじめ **Number** オブジェクトを作成しておく必要はありません。

JScript では、**POSITIVE_INFINITY** 値を Infinity として表示します。この値は、算術演算では無限大として機能します。

必要条件

Version 2

対象：

Number オブジェクト

参照
関連項目
MAX_VALUE プロパティ
MIN_VALUE プロパティ
NaN プロパティ
NEGATIVE_INFINITY プロパティ
toString メソッド

propertyIsEnumerable プロパティ

指定したプロパティがオブジェクトの一部であるかどうか、および加算できるかどうかを表すブール値を返します。

```
object.propertyIsEnumerable(propName)
```

引数

object

必ず指定します。オブジェクトのインスタンスを指定します。

propName

必ず指定します。プロパティ名の文字列値を指定します。

解説

propertyIsEnumerable プロパティは、*propName* が *object* に存在し、**For...In** ループを使用して列挙できる場合に **true** を返します。**propertyIsEnumerable** プロパティは、指定した名前のプロパティが *object* にない場合か、指定したプロパティが加算できない場合に **false** を返します。通常、定義済みのプロパティは加算可能ではなく、ユーザー定義のプロパティは必ず加算可能になります。

propertyIsEnumerable プロパティでは、プロトタイプ チェインのオブジェクトは対象外です。

使用例

propertyIsEnumerable プロパティの使用例を次に示します。

```
var a : Array = new Array("apple", "banana", "cactus");
print(a.propertyIsEnumerable(1));
```

このプログラムの出力は次のようになります。

```
true
```

必要条件

Version 5.5

対象

Object オブジェクト

参照

その他の技術情報

プロパティ (JScript)

prototype プロパティ

指定されたオブジェクトのクラスのプロトタイプへの参照を返します。

```
object.prototype
```

引数
object

必ず指定します。オブジェクトの名前を指定します。

解説

prototype プロパティを使用すると、オブジェクトのクラスが持つ機能の基本セットを知ることができます。オブジェクトの新しいインスタンスは、そのオブジェクトに割り当てられているプロトタイプの機能を "継承" します。

組み込みオブジェクトはすべて、値の取得のみ可能な **prototype** プロパティを持っています。これらのオブジェクトのプロトタイプには、このコード例のように機能を追加することはできますが、オブジェクトに別のプロトタイプを割り当てることはできません。ただし、ユーザー定義オブジェクトは、新しいプロトタイプに割り当てることができます。

このランゲージ リファレンスには、組み込みオブジェクトごとにメソッドとプロパティの一覧があり、オブジェクトのプロトタイプに含まれているメソッドとプロパティを調べることができます。

メモ :

組み込みオブジェクトの **prototype** プロパティは、高速モードで実行されている場合は変更できません。JScript の既定のモードは高速モードです。**prototype** プロパティを使用するプログラムをコマンド ラインからコンパイルするには、**/fast-** を使用して fast オプションをオフにする必要があります。ASP.NET で fast オプションをオフにするのは安全ではありません。スレッドに関する問題が発生する場合があります。

使用例

最も大きい配列要素を返す **Array** オブジェクトに任意のメソッドを追加する場合は、関数を宣言し、**Array.prototype** に追加してから使用します。

```
function array_max() {
   var i, max = this[0];
   for (i = 1; i < this.length; i++) {
      if (max < this[i])
         max = this[i];
   }
   return max;
}
Array.prototype.max = array_max;
var x = new Array(1, 2, 3, 4, 5, 6);
print(x.max());
```

/fast- オプションを指定してコンパイルすると、プログラムの出力は次のようになります。

```
6
```

必要条件

Version 2

対象

Array オブジェクト| Boolean オブジェクト| Date オブジェクト| Function オブジェクト| Number オブジェクト| Object オブジェクト| String オブジェクト

参照
関連項目
constructor プロパティ

rightContext プロパティ ($')

検索した文字列内で、最後に一致した位置から最後までの文字を返します。読み取り専用です。

```
RegExp.rightContext
```

引数

RegExp

必ず指定します。グローバルな **RegExp** オブジェクトを指定します。

解説

rightContext プロパティの初期値は空の文字列です。**rightContext** プロパティの値は、検索が成功するたびに変更されます。

メモ :

RegExp オブジェクトのプロパティは、高速モードで実行されている場合は利用できません。JScript の既定のモードは高速モードです。これらのプロパティを使用するプログラムをコマンド ラインからコンパイルするには、**/fast-** を使用して fast オプションをオフにする必要があります。ASP.NET で fast オプションをオフにするのは安全ではありません。スレッドに関する問題が発生する場合があります。

rightContext プロパティの短縮形は **$'** です。式 RegExp**["$'"]** と式 RegExp.**rightContext** は同義です。

使用例

rightContext プロパティの使用例を次に示します。

```
var s;                                //Declare variable.
var re = new RegExp("d(b+)(d)","ig"); //Regular expression pattern.
var str = "cdbBdbsbdbdz";             //String to be searched.
var arr = re.exec(str);               //Perform the search.
s = "$1 returns: " + RegExp.$1 + "\n";
s += "$2 returns: " + RegExp.$2 + "\n";
s += "$3 returns: " + RegExp.$3 + "\n";
s += "input returns : " + RegExp.input + "\n";
s += "lastMatch returns: " + RegExp.lastMatch + "\n";
s += "leftContext returns: " + RegExp.leftContext + "\n";
s += "rightContext returns: " + RegExp.rightContext + "\n";
s += "lastParen returns: " + RegExp.lastParen + "\n";
print(s);                            //Return results.
```

/fast- オプションを指定してコンパイルすると、このプログラムの出力は次のようになります。

```
$1 returns: bB
$2 returns: d
$3 returns:
input returns : cdbBdbsbdbdz
lastMatch returns: dbBd
leftContext returns: c
rightContext returns: bsbdbdz
lastParen returns: d
```

必要条件

Version 5.5

対象

RegExp オブジェクト

参照

関連項目

$1...$9 プロパティ

source プロパティ

正規表現パターンのテキストを返します。読み取り専用です。

```
rgExp.source
```

引数
rgExp

必ず指定します。**Regular Expression** オブジェクトを指定します。

解説

rgExp は、**Regular Expression** オブジェクトを格納する変数または正規表現リテラルです。

使用例

次のコードは、**source** プロパティの使用例です。

```
var src : String = "Spain";
var re : RegExp = /in/g;
var s1;
// Test string for existence of regular expression.
if (re.test(src))
   s1 = " contains ";
else
   s1 = " does not contain ";
// Get the text of the regular expression itself.
print("The string " + src + s1 + re.source + ".");
```

このプログラムの出力は次のようになります。

```
The string Spain contains in.
```

必要条件

Version 3

対象

Regular Expression オブジェクト

参照
関連項目
Regular Expression オブジェクト
概念
正規表現の構文

SQRT1_2 プロパティ

2 分の 1 (0.5) の平方根の値を返します。

```
Math.SQRT1_2
```

引数

Math

必ず指定します。グローバルな **Math** オブジェクトを指定します。

解説

SQRT1_2 プロパティは、約 0.707 の値を持つ定数です。

必要条件

Version 1

対象

Math オブジェクト

参照

関連項目

sqrt メソッド
SQRT2 プロパティ

SQRT2 プロパティ

2 の平方根の値を返します。

```
Math.SQRT2
```

引数

Math

必ず指定します。グローバルな **Math** オブジェクトを指定します。

解説

SQRT2 プロパティは、約 1.414 の値を持つ定数です。

必要条件

Version 1

対象

Math オブジェクト

参照

関連項目

sqrt メソッド
SQRT1_2 プロパティ

undefined プロパティ

undefined の値を返します。

```
undefined
```

解説

undefined プロパティは **Global** オブジェクトのメンバで、スクリプト エンジンが初期化されたときに使用できるようになります。変数を宣言し、初期化していない場合、その値は **undefined** になります。

変数を宣言していない場合、その変数と **undefined** は比較できません。ただし、その変数の型と文字列 "undefined" は比較できます。高速モードでは、宣言されていない変数は使用できません。

undefined プロパティは、変数に undefined を明示的に設定するときまたはテストするときに便利です。

使用例

undefined プロパティの使用例を次に示します。

```
var declared;                          //Declare variable.
if (declared == undefined)             //Test variable.
   print("The variable declared has not been given a value.");
```

このコードの出力は次のようになります。

```
The variable declared has not been given a value.
```

必要条件

Version 5.5

対象：

Global オブジェクト

参照
概念
undefined 値

ステートメント

ステートメントは、アクションを実行する JScript コードの一部です。変数、関数、クラス、列挙型などのユーザー定義要素を宣言するステートメントや、プログラム フローを制御するステートメントがあります。ここでは、JScript のステートメントの使用方法に関する情報へのリンクを示します。

このセクションの内容

interface ステートメント

インターフェイスとインターフェイスのメンバを宣言します。

ラベル付きステートメント

ステートメントの識別子を指定します。

package ステートメント

クラスとインターフェイスを名前付きのコンポーネントにパッケージ化します。

print ステートメント

コマンド ラインから実行されたプログラムの情報を表示する手段を提供します。

return ステートメント

現在の関数の実行を終了し、戻り値を返します。

@set ステートメント

条件付きコンパイル ステートメントで使用する変数を作成します。

static ステートメント

クラスを初期化するコードのブロックを宣言します。

super ステートメント

現在のオブジェクトの基本クラスを参照します。

switch ステートメント

指定した式の値がラベルと一致したときに 1 つ以上のステートメントを実行する機能を提供します。

this ステートメント

現在のオブジェクトを返します。

throw ステートメント

try...catch ステートメントで処理できるエラー条件を生成します。

try...catch...finally ステートメント

JScript のエラー処理機能を実装します。

var ステートメント

変数を宣言します。

while ステートメント

指定した条件が偽 (**false**) になるまでステートメントを繰り返し実行します。

with ステートメント

ステートメントで使用する既定のオブジェクトを設定します。

関連するセクション

JScript リファレンス

JScript の言語リファレンスを構成する要素と、言語要素の適切な使用に関する背景情報を説明するトピックへのリンク一覧を示します。

.NET Framework リファレンス

.NET Framework クラス ライブラリおよびその他の基本要素の構文と構造を説明するトピックへのリンクを示します。

break ステートメント

現在のループを終了します。ラベルが指定された場合は、関連するステートメントを終了します。

```
break [label];
```

引数
label

省略可能です。終了するステートメントのラベルを指定します。

解説

通常、**break** ステートメントは、**switch** ステートメント、**while** ループ、**for** ループ、**for...in** ループ、または **do...while** ループの中で使用します。引数 *label* は、**switch** ステートメントの中で頻繁に使用しますが、他のステートメントの中でも使用できます。引数 *label* は、複合ステートメントの中でも使用できます。

break ステートメントを実行すると、プログラムは現在のループまたはステートメントを終了します。プログラムは、終了したループまたはステートメントの直後のステートメントから処理を再開します。

例 1

次のコードは、**break** ステートメントの使用例です。

```
function breakTest(breakpoint){
   var i = 0;
   while (i < 100) {
      if (i == breakpoint)
         break;
      i++;
   }
   return(i);
}
```

例 2

次のコードは、ラベル付きの **break** ステートメントの使用例です。

```
function nameInDoubleArray(name, doubleArray) {
   var i, j, inArray;
   inArray = false;
   mainloop:
   for(i=0; i<doubleArray.length; i++)
      for(j=0; j<doubleArray[i].length; j++)
         if(doubleArray[i][j] == name) {
            inArray = true;
            break mainloop;
         }
   return inArray;
}
```

必要条件

Version 1

参照
関連項目
continue ステートメント
do...while ステートメント
for ステートメント
for...in ステートメント
ラベル付きステートメント
while ステートメント

class ステートメント

クラスの名前と、クラスを構成する変数、プロパティ、およびメソッドの定義を宣言します。

```
[modifiers] class classname [extends baseclass] [implements interfaces]{
   [classmembers]
}
```

引数

modifiers

省略可能です。クラスの参照可能範囲と動作を制御する修飾子。

classname

必ず指定します。**class** の名前。標準的な変数の名前付け規則に従って名前を付けます。

extends

省略可能です。クラス *classname* がクラス *baseclass* を拡張することを示すキーワード。このキーワードが指定されていない場合、**System.Object** を拡張する標準の JScript 基本クラスが作成されます。

baseclass

省略可能です。拡張されるクラスの名前。

implements

省略可能です。クラス *classname* が 1 つ以上のインターフェイスを実装することを示すキーワード。

interfaces

省略可能です。インターフェイス名のコンマ区切りのリスト。

classmembers

省略できます。**function** ステートメントで定義されるメソッドまたはコンストラクタの宣言、**function get** および **function set** ステートメントで定義されるプロパティの宣言、**var** または **const** ステートメントで定義されるフィールドの宣言、**static** ステートメントで定義される初期化子の宣言、**enum** ステートメントで定義される列挙の宣言、または入れ子になったクラス宣言。

解説

クラスの修飾子に応じて、クラスを使用してインスタンスを作成したり、クラスを他のクラスの基本クラスにしたりできます。クラスに **abstract** 修飾子が指定されていると、他のクラスを拡張するための基本クラスとして機能します。ただし、**abstract** クラスのインスタンスは作成できません。クラスに **final** 修飾子が指定されている場合、**new** 演算子を使用してクラスのインスタンスを作成できますが、基本クラスとしては使用できません。

メソッドおよびコンストラクタはオーバーロードされる場合があります。したがって、複数のメソッド (またはコンストラクタ) が同じ名前を持つ場合もあります。オーバーロードされたクラスのメンバは、一意のシグネチャによって区別されます。シグネチャは、メンバの名前と、それぞれの仮パラメータのデータ型によって構成されます。オーバーロードにより、クラスは類似する機能ごとにメソッドをグループ化できます。

クラスは、**extends** キーワードを使用することで、既存の基本クラスの機能を継承できます。基本クラスのメソッドは、同じシグネチャを持つ新しいメソッドを宣言することでオーバーライドできます。新しいクラスのメソッドは、**super** ステートメントを使用して、オーバーライドされた基本クラスのメンバにアクセスできます。

クラスは、**implements** キーワードを使用して、1 つ以上のインターフェイスに基づいて作成できます。インターフェイスはメンバの実装を提供しないため、クラスはインターフェイスから動作を継承できません。インターフェイスは、他のクラスとやり取りするときに使用する "シグネチャ" をクラスに与えます。インターフェイスを実装するクラスが **abstract** でない場合は、インターフェイスで定義されている各メソッドに実装を提供する必要があります。

修飾子を指定すると、クラスのインスタンスは、より JScript オブジェクトらしく動作します。動的に追加されるプロパティをクラスのインスタンスで処理するには、**expando** 修飾子を使用します。この修飾子は、クラスの既定のインデックス付きプロパティを自動的に作成します。expando プロパティにアクセスするには、JScript の **Object** オブジェクトの角かっこ表記を使用する必要があります。

例 1

次の例では、さまざまなフィールドとメソッドを持つ `CPerson` クラスを作成します。それぞれの詳細は省略されています。`CPerson` クラスは、2 番

目の例の `CCustomer` クラスの基本クラスとなります。

```
// All members of CPerson are public by default.
class CPerson{
   var name : String;
   var address : String;

   // CPerson constuctor
   function CPerson(name : String){
      this.name = name;
   };

   // printMailingLabel is an instance method, as it uses the
   // name and address information of the instance.
   function printMailingLabel(){
      print(name);
      print(address);
   };

   // printBlankLabel is static as it does not require
   // any person-specific information.
   static function printBlankLabel(){
      print("-blank-");
   };
}

// Print a blank mailing label.
// Note that no CPerson object exists at this time.
CPerson.printBlankLabel();

// Create a CPerson object and add some data.
var John : CPerson = new CPerson("John Doe");
John.address = "15 Broad Street, Atlanta, GA 30315";
// Print a mailing label with John's name and address.
John.printMailingLabel();
```

このコードの出力は次のようになります。

```
-blank-
John Doe
15 Broad Street, Atlanta, GA 30315
```

例 2

`CCustomer` クラスは `CPerson` から派生します。追加されたフィールドとメソッドは、`CPerson` クラスの汎用メンバには適用できません。

```
// Create an extension to CPerson.
class CCustomer extends CPerson{
   var billingAddress : String;
   var lastOrder : String;

   // Constructor for this class.
   function CCustomer(name : String, creditLimit : double){
      super(name); // Call superclass constructor.
      this.creditLimit = creditLimit;
   };

   // Customer's credit limit. This is a private field
   // so that only member functions can change it.
   private var creditLimit : double;
   // A public property is needed to read the credit limit.
   function get CreditLimit() : double{
      return creditLimit;
   }
}
```

```
// Create a new CCustomer.
var Jane : CCustomer = new CCustomer("Jane Doe",500.);
// Do something with it.
Jane.billingAddress = Jane.address = "12 Oak Street, Buffalo, NY 14201";
Jane.lastOrder = "Windows 2000 Server";
// Print the credit limit.
print(Jane.name + "'s credit limit is " + Jane.CreditLimit);
// Call a method defined in the base class.
Jane.printMailingLabel();
```

このコードの出力は次のようになります。

```
Jane Doe's credit limit is 500
Jane Doe
12 Oak Street, Buffalo, NY 14201
```

必要条件

バージョン .NET

参照

関連項目

interface ステートメント
function ステートメント
function get ステートメント
function set ステートメント
var ステートメント
const ステートメント
static ステートメント
new 演算子
this ステートメント
super ステートメント

その他の技術情報

修飾子

@cc_on ステートメント

条件付きコンパイルの機能をアクティブにします。

```
@cc_on
```

解説

@cc_on ステートメントを記述すると、スクリプト エンジンで条件コンパイルが行われるようになります。

@cc_on ステートメントは、コメントの中に記述するようにします。この場合、条件コンパイルをサポートしていないブラウザでも、構文エラーが発生しません。次にそのコード例を示します。

```
/*@cc_on*/
// The remainder of the script.
```

@if ステートメントまたは **@set** ステートメントをコメント外に記述することでも、条件コンパイルをアクティブにできます。

必要条件

Version 3

参照

関連項目

@if...@elif...@else...@end ステートメント

@set ステートメント

概念

条件付きコンパイル変数

その他の技術情報

条件付きコンパイル

コメント ステートメント

コード内にコメントを記述するときに使用します。

```
//single-line comment
// comment

//multiline comment
/*
comment
*/

//single-line conditional comment
//@CondStatement

//multiline conditional comment
/*@
condStatement
@*/
```

解説

CondStatement は、条件付きコンパイルがアクティブな場合に使用される条件付きコンパイル コードです。構文 3 を使用する場合、"//" と "@" の間にはスペースを入力しません。

コメントは、スクリプト内に JScript パーサーに解釈されない部分を作るときに使用します。コメントを使用すると、プログラム内に動作の説明を入れることができます。

構文 1 を使用すると、パーサーはコメント記号から行末までのテキストを無視します。構文 2 を使用すると、パーサーは開始記号から終了記号までのテキストを無視します。

構文 3 または構文 4 を使用すると、条件付きコンパイルがサポートされるようになり、条件付きコンパイルの機能がサポートされていないブラウザとの互換性が維持されます。このようなブラウザは、構文 3 と構文 4 の形式のコメントを、それぞれ構文 1 と構文 2 の形式のコメントとして扱います。

使用例

次のコードは、コメント ステートメントの最も一般的な使用例です。

```
function myfunction(arg1, arg2){
   /* This is a multiline comment that
      can span as many lines as necessary. */
   var r = 0;
   // This is a single line comment.
   r = arg1 + arg2; // Sum the two arguments.
   return(r);
}
```

必要条件

Version 1

参照

関連項目

@cc_on ステートメント

@set ステートメント

概念

条件付きコンパイル変数

その他の技術情報

条件付きコンパイル

const ステートメント

定数を宣言します。

```
//Syntax for declaring a constant of global scope or function scope.
const name1 [: type1] = value1 [, ... [, nameN [: typeN] = valueN]]
//Syntax for declaring a constant field in a class.
[modifiers] const name1 [: type1] = value1 [, ... [, nameN [: typeN] = valueN]]
```

引数

modifiers

省略可能です。フィールドの参照可能範囲と動作を制御する修飾子。

name1, ..., nameN

必ず指定します。宣言する定数の名前。

type1, ..., typeN

省略可能です。宣言する定数の型。

value1, ..., valueN

定数に代入する値。

解説

定数を宣言するには、**const** ステートメントを使用します。定数は特定のデータ型に連結され、タイプ セーフを提供します。これらの定数には、宣言時に値を代入する必要があります。代入した値を後からスクリプトで変更することはできません。

クラスの定数フィールドは、グローバル定数や関数定数に似ていますが、クラスのスコープを持ち、参照可能範囲と使用方法を制御するさまざまな修飾子を指定できる点で異なっています。

メモ：

定数を参照データ型 (**Object**、**Array**、クラス インスタンス、型指定された配列など) に連結すると、定数で参照されるデータが変更されることがあります。これは、**const** ステートメントが参照型だけを定数として、参照するデータを定数としないために起こります。

使用例

次に、**const** ステートメントの使用例を示します。

```
class CSimple {
   // A static public constant field. It will always be 42.
   static public const constantValue : int = 42;
}
const index = 5;
const name : String = "Thomas Jefferson";
const answer : int = 42, oneThird : float = 1./3.;
const things : Object[] = new Object[50];
things[1] = "thing1";
// Changing data referenced by the constant is allowed.
```

必要条件

バージョン .NET

参照

関連項目

var ステートメント
function ステートメント
class ステートメント

概念

変数と定数のスコープ

型の注釈
その他の技術情報
修飾子

continue ステートメント

ループの現在の反復の実行を中止し、次の反復の実行を開始します。

```
continue [label];
```

引数
label

省略可能です。**continue** を適用するステートメントを指定します。

解説

continue ステートメントは、**while** ループ、**do...while** ループ、**for** ループ、および **for...in** ループの中だけで使用できます。**continue** ステートメントを実行すると、ループの現在の反復の実行が中止され、プログラムの実行は、ループの先頭から続行されます。このステートメントの動作は、次のようにループの種類により少しずつ異なります。

- **while** ループと **do...while** ループでは、条件が評価され、その結果が真 (true) の場合はループの実行が繰り返されます。
- **for** ループでは、最初にインクリメント式が実行され、次に条件式が真 (true) の場合はループの実行が繰り返されます。
- **for...in** ループでは、指定された変数の次のフィールドに進み、ループの実行が繰り返されます。

使用例

次のコードは、**continue** ステートメントの使用例です。

```
function skip5(){
   var s = "", i=0;
   while (i < 10) {
      i++;
      // Skip 5
      if (i==5) {
         continue;
      }
      s += i;
   }
   return(s);
}
```

必要条件

Version 1

参照
関連項目
break ステートメント
do...while ステートメント
for ステートメント
for...in ステートメント
ラベル付きステートメント
while ステートメント

debugger ステートメント

デバッガを起動します。

```
debugger
```

解説

debugger ステートメントは、インストールされているデバッガを起動します。効果は、debugger ステートメントが使用されているプログラムにブレークポイントを設定するのと似ています。

デバッガがインストールされていない場合、**debugger** ステートメントの効果はありません。

必要条件

Version 3

参照

その他の技術情報

ステートメント

JScript コードの作成、コンパイル、およびデバッグ

do...while ステートメント

ステートメント ブロックを一度実行し、その後、条件式の評価が偽 (**false**) になるまでループ実行を繰り返します。

```
do
      statement
while (expression)
```

引数
statement

必ず指定します。*expression* の評価が真 (**true**) の場合に実行するステートメントを指定します。複合ステートメントを指定することもできます。

expression

必ず指定します。真 (**true**) または偽 (**false**) のブール値に強制変換できる式を指定します。この式の評価が真 (**true**) の場合は、再びループが実行されます。偽 (**false**) の場合は、ループ処理を終了します。

解説

expression に指定した式の値は、ループの最初の繰り返し実行が終わるまでは確認されません。このため、ループは少なくとも一度は必ず実行されます。その後は、ループの繰り返し実行が正常に終了するたびに式の値が確認されます。

使用例

次のコードは、**do...while** ステートメントを使って **Drives** コレクションを繰り返し処理する例です。

```
function GetDriveList(){
   var fso, s, n, e, x;
   fso = new ActiveXObject("Scripting.FileSystemObject");
   e = new Enumerator(fso.Drives);
   s = "";
   do {
      x = e.item();
      s = s + x.DriveLetter;
      s += " - ";
      if (x.DriveType == 3)
         n = x.ShareName;
      else if (x.IsReady)
         n = x.VolumeName;
      else
         n = "[Drive not ready]";
         s +=  n + "\n";
      e.moveNext();
   }
   while (!e.atEnd());
   return(s);
}
```

必要条件

Version 3

参照
関連項目
break ステートメント
continue ステートメント
for ステートメント
for...in ステートメント
while ステートメント
ラベル付きステートメント

enum ステートメント

列挙型の名前と、列挙型のメンバの名前を宣言します。

```
[modifiers] enum enumName [ : typeAnnotation]{
   enumValue1 [ = initializer1]
   [,enumValue2 [ = initializer2]
   [, ... [,enumValueN [ = initializerN ] ]]]
}
```

引数

modifiers

省略可能です。列挙型の参照可能範囲と動作を制御する修飾子。

enumName

必ず指定します。列挙型の名前。

typeAnnotation

省略可能です。列挙型の基になるデータ型。整数型である必要があります。既定値は **int** です。

enumValue1, enumValue2, ..., enumValueN

省略可能です。列挙型のメンバ。

initializer1, initializer2, ..., initializerN

省略可能です。列挙型メンバの既定の数値をオーバーライドする定数式。

解説

enum 宣言は、プログラムに新しい列挙型を導入します。**enum** 宣言は、クラスを宣言できる場所にだけ指定できます。つまり、グローバル スコープ、パッケージ スコープ、またはクラス スコープでは宣言できますが、関数またはメソッド内部では宣言できません。

列挙型の基になる型は、整数型 (**int**、**short**、**long**、**byte**、**uint**、**ushort**、**ulong**、または **sbyte**) として宣言できます。列挙メンバは、基になるデータ型に対して、相互に暗黙的に強制変換されます。これにより、**enum** 型の変数に数値データを直接代入できます。既定では、列挙型の基になるデータ型は **int** です。

列挙型メンバには、それぞれ名前と省略可能な初期化子があります。初期化子は、指定した列挙型と同じ型、またはその型に変換可能な、コンパイル時の定数式である必要があります。列挙型の最初のメンバの値が指定されている場合、その値は 0 または初期化子の値です。以降のメンバの値が指定されている場合、その値は、直前のメンバより 1 大きい値か、初期化子の値になります。

enum 値にアクセスする方法は、静的クラス メンバにアクセスする方法に似ています。メンバの名前は、`Color.Red` のように列挙型の名前で修飾する必要があります。**enum** 型の変数に値を代入する場合は、完全限定名 (`Color.Red` など)、文字列形式の名前 (`"Red"` など)、または数値のいずれかを使用します。

コンパイル時に **enum** に既知の文字列が代入される場合、コンパイラは必要な変換を実行します。たとえば、`"Red"` は `Color.Red` に置き換えられます。文字列がコンパイル時に既知でない場合、変換は実行時に行われます。文字列が列挙型の有効なメンバでない場合、変換に失敗することがあります。変換に時間がかかり、ランタイム エラーが発生する場合があるため、**enum** を可変長の文字列に代入することは避けてください。

列挙型の変数は、宣言された値の範囲を超える値を保持できます。この機能を利用して、次の例に示すように、ビット フラグとして使用されるメンバを組み合わせることができます。**enum** 変数を文字列に変換すると、文字列形式のメンバ名となります。

例 1

次の例は、列挙型の動作を示しています。`CarType` という名前の簡単な列挙型を宣言します。列挙型には、`Honda`、`Toyota`、および `Nissan` というメンバがあります。

```
     enum CarType {
   Honda,    // Value of zero, since it is first.
   Toyota,   // Value of 1, the successor of zero.
   Nissan    // Value of 2.
}
```

```
// Declare a variable of type CarType, and give it the value Honda.
var myCar : CarType = CarType.Honda;
print(int(myCar) + ": " + myCar);

myCar = "Nissan"; // Change the value to "Nissan".
print(int(myCar) + ": " + myCar);

myCar = 1; // 1 is the value of the Toyota member.
print(int(myCar) + ": " + myCar);
```

このコードの出力は次のようになります。

```
0: Honda
2: Nissan
1: Toyota
```

例 2

次の例は、列挙型を使用してビット フラグを保持する方法を示しています。また、**enum** 変数には、メンバ リストに明示的に示されていない値を保持できることも示しています。この例では、`FormatFlags` という列挙型が定義され、`Format` 関数の動作を変更するために使用されています。

```
// Explicitly set the type to byte, as there are only a few flags.
enum FormatFlags : byte {
   // Can't use the default values, since we need explicit bits
   ToUpperCase = 1,   // Should not combine ToUpper and ToLower.
   ToLowerCase = 2,
   TrimLeft    = 4,   // Trim leading spaces.
   TrimRight   = 8,   // Trim trailing spaces.
   UriEncode   = 16   // Encode string as a URI.
}

function Format(s : String, flags : FormatFlags) : String {
   var ret : String = s;
   if(flags & FormatFlags.ToUpperCase) ret = ret.toUpperCase();
   if(flags & FormatFlags.ToLowerCase) ret = ret.toLowerCase();
   if(flags & FormatFlags.TrimLeft)    ret = ret.replace(/^\s+/g, "");
   if(flags & FormatFlags.TrimRight)   ret = ret.replace(/\s+$/g, "");
   if(flags & FormatFlags.UriEncode)   ret = encodeURI(ret);
   return ret;
}

// Combine two enumeration values and store in a FormatFlags variable.
var trim : FormatFlags = FormatFlags.TrimLeft | FormatFlags.TrimRight;
// Combine two enumeration values and store in a byte variable.
var lowerURI : byte = FormatFlags.UriEncode | FormatFlags.ToLowerCase;

var str : String = "  hello, WORLD  ";

print(trim + ": " + Format(str, trim));
print(FormatFlags.ToUpperCase + ": " + Format(str, FormatFlags.ToUpperCase));
print(lowerURI + ": " + Format(str, lowerURI));
```

このコードの出力は次のようになります。

```
12: hello, WORLD
ToUpperCase:   HELLO, WORLD
18: %20%20hello,%20world%20%20
```

必要条件

バージョン .NET

参照

概念

型変換

型の注釈

その他の技術情報

修飾子

for ステートメント

指定された条件が真 (true) の間、複数のステートメントから構成されるブロックを繰り返して実行します。

```
for (initialization; test; increment)
...statement
```

引数

initialization

必ず指定します。任意の式を指定します。この式は、ループを実行する直前に一度だけ実行されます。

test

必ず指定します。ブール式を指定します。*test* が真 (**true**) の場合は、*statement* が実行されます。*test* が偽 (**false**) になると、ループの実行は終了します。

increment

必ず指定します。任意の式を指定します。インクリメント式は、ループ内のブロックの実行が終わるたびに実行されます。

statement

省略可能です。*test* の評価が真 (**true**) の場合に実行するステートメントを指定します。複合ステートメントを指定することもできます。

解説

特定の回数だけ処理を繰り返す場合は、**for** ループを使います。

使用例

次のコードは、**for** ループの使用例です。

```
/* i is set to 0 at start, and is incremented by 1 at the end
   of each iteration. Loop terminates when i is not less
   than 10 before a loop iteration. */
var myarray = new Array();
for (var i = 0; i < 10; i++) {
   myarray[i] = i;
}
```

必要条件

Version 1

参照

関連項目

for...in ステートメント
while ステートメント

for...in ステートメント

オブジェクトの各プロパティ、または配列やコレクションの各要素に対して、1 つ以上のステートメントを実行します。

```
for ( [var] variable in {object | array | collection})
      statement
```

引数

variable

必ず指定します。*object* のプロパティ名、*array* のインデックス、または *collection* の要素を格納するために使用する変数を指定します。

object

反復処理の対象となる JScript オブジェクトを指定します。

array

反復処理の対象となる配列を指定します。JScript **Array** オブジェクトまたは .NET Framework 配列を指定できます。

collection

反復処理の対象となるコレクションを指定します。.NET Framework の **IEnumerable** インターフェイスまたは **IEnumerator** インターフェイスを実装する任意のクラスを指定できます。

statement

省略可能です。*object* または *collection* のプロパティ、または *array* の各メンバに対して実行するステートメントを指定します。複合ステートメントを指定することもできます。

解説

ループの各反復処理の実行前に、*object* の次のプロパティ名、*array* の次のインデックス、または *collection* の次の要素が *variable* に代入されます。ループ内のステートメントで *variable* を使用して、*object* のプロパティまたは *array* の要素を参照できます。

オブジェクトに対して反復処理を行う場合は、*variable* に代入されるオブジェクトのメンバ名の順番を決定したり制御したりすることはできません。**for...in** ステートメントは、.NET Framework オブジェクトなどの JScript 以外のオブジェクトのメンバに対してはループを実行できません。

配列の反復処理は、要素の順序で実行されます。最小のインデックスから開始し、最大のインデックスで終了します。JScript の **Array** オブジェクトは疎配列であるため、**for...in** ステートメントは、配列の定義済み要素だけにアクセスします。JScript の **Array** オブジェクトには expando プロパティが含まれることがあります。その場合は、配列インデックスがプロパティ名として *variable* に代入されます。配列が .NET Framework の多次元配列の場合は、最初の次元だけが列挙されます。

コレクションに対する反復操作の場合は、コレクション内の順序どおりに要素が *variable* に代入されます。

例 1

次のコードは、**for ... in** ステートメントで、連想処理可能な配列として使用する例です。

```
function ForInDemo1() {
   var ret = "";

   // Initialize the object with properties and values.
   var obj : Object = {"a" : "Athens" ,
                       "b" : "Belgrade",
                       "c" : "Cairo"};

   // Iterate over the properties.
   for (var key in obj)
      // Loop and assign 'a', 'b', and 'c' to key.
      ret += key + ":\t" + obj[key] + "\n";

   return(ret);
} // ForInDemo1
```

この関数は、以下を含む文字列を返します。

```
a:      Athens
b:      Belgrade
c:      Cairo
```

例 2

expando プロパティを含む JScript の **Array** オブジェクトと共に **for ... in** ステートメントを使用する例を次に示します。

```
function ForInDemo2() {
   var ret = "";

   // Initialize the array.
   var arr : Array = new Array("zero","one","two");
   // Add a few expando properties to the array.
   arr["orange"] = "fruit";
   arr["carrot"] = "vegetable";

   // Iterate over the properties and elements.
   for (var key in arr)
      // Loop and assign 0, 1, 2, 'orange', and 'carrot' to key.
      ret += key + ":\t" + arr[key] + "\n";

   return(ret);
} // ForInDemo2
```

この関数は、以下を含む文字列を返します。

```
0:      zero
1:      one
2:      two
orange: fruit
carrot: vegetable
```

例 3

コレクションと共に **for ... in** ステートメントを使用する例を次に示します。この例の **System.String** オブジェクトの **GetEnumerator** メソッドは、文字列内の文字のコレクションを提供します。

```
function ForInDemo3() {
   var ret = "";

   // Initialize collection.
   var str : System.String = "Test.";
   var chars : System.CharEnumerator = str.GetEnumerator();

   // Iterate over the collection elements.
   var i : int = 0;
   for (var elem in chars) {
      // Loop and assign 'T', 'e', 's', 't', and '.' to elem.
      ret += i + ":\t" + elem + "\n";
      i++;
   }

   return(ret);
} // ForInDemo3
```

この関数は、以下を含む文字列を返します。

```
0:      T
1:      e
2:      s
3:      t
4:      .
```

必要条件

Version 5

メモ：
コレクションに対してループを実行するには、Version .NET が必要です。

参照

関連項目

for ステートメント

while ステートメント

String.GetEnumerator Method

その他の技術情報

JScript の配列

function ステートメント

新しい関数を宣言します。このステートメントは、次の場合に使用できます。

```
// in the global scope
function functionname([parmlist]) [: type] {
   [body]
}

// declares a method in a class
[attributes] [modifiers] function functionname([parmlist]) [: type] {
   [body]
}

// declares a method in an interface
[attributes] [modifiers] function functionname([parmlist]) [: type]
```

引数

attributes

省略可能です。メソッドの参照可能範囲と動作を制御する属性を指定します。

modifiers

省略可能です。メソッドの参照可能範囲と動作を制御する修飾子を指定します。

functionname

必ず指定します。関数またはメソッドの名前を指定します。

paramlist

省略可能です。関数またはメソッドの、コンマ区切りのパラメータ リストを指定します。各パラメータには、型指定が含まれる場合があります。最後のパラメータは *parameterarray* となる場合があります。この配列は、3 つのピリオド (**...**)、パラメータ配列名、および型指定された配列の型の注釈で表されます。

type

省略可能です。メソッドの戻り値の型を指定します。

body

省略可能です。関数またはメソッドの動作を定義する 1 つ以上のステートメントを指定します。

解説

function ステートメントは、関数を宣言するときに使用します。*body* に記述したコードは、スクリプトの他の場所でこの関数が呼び出されるまでは実行されません。**return** ステートメントは、関数から値を返すときに使用されます。**return** ステートメントは使用しなくてもかまいません。関数の終わりに達すると、プログラムは関数から戻ります。

メソッドはグローバル関数に似ていますが、スコープは定義されている **class** または **interface** となり、参照可能範囲や動作を制御するさまざまな修飾子を指定できます。**interface** 内では、メソッドは本体を持ちません。**class** 内のメソッドには、本体を指定する必要があります。ただし、**class** 内のメソッドが **abstract** であるか、**class** が **abstract** である場合、メソッドは本体を持ちません。

型の注釈を使用して、関数またはメソッドが返すデータ型を宣言できます。戻り値の型として **void** が指定されている場合は、関数内部のどの **return** ステートメントからも値が返されないことがあります。**void** 以外の戻り値の型を指定すると、関数内のすべての **return** ステートメントは、指定した戻り値の型に変換可能な値を返す必要があります。戻り値の型を指定して、**return** ステートメントに値を指定しなかった場合、または **return** ステートメントが実行されないまま関数の終わりに到達した場合は、**undefined** 値が返されます。コンストラクタ関数には戻り値の型を指定できません。**new** 演算子が、作成されるオブジェクトを自動的に返します。

関数に対して戻り値の型を明示的に指定しない場合、戻り値の型は **Object** または **void** になります。**return** ステートメントがない場合、または関数本体で **return** ステートメントに値が指定されていない場合、戻り値の型は **void** になります。

関数の最後のパラメータとして、パラメータ配列を使用できます。必須のパラメータより後で関数に渡される引数は、パラメータ配列に入れられます。パラメータの型の注釈は省略できますが、型指定された配列である必要があります。任意の型のパラメータを受け取るために、型指定された配列として **Object[]** を使用します。引数の数が変化する関数を呼び出すときに、予期される型の明示的な配列がパラメータのリストの代わりに使用されます。

関数を呼び出すときは、必須の引数とかっこを記述してください。かっこを付けずに関数を呼び出すと、関数の結果ではなく、関数のテキストが返されます。

例 1

次のコードは、**function** ステートメントの構文 1 の使用例です。

```
interface IForm {
   // This is using function in Syntax 3.
   function blank() : String;
}

class CForm implements IForm {
   // This is using function in Syntax 2.
   function blank() : String {
      return("This is blank.");
   }
}

// This is using function in Syntax 1.
function addSquares(x : double, y : double) : double {
   return(x*x + y*y);
}

// Now call the function.
var z : double = addSquares(3.,4.);
print(z);

// Call the method.
var derivedForm : CForm = new CForm;
print(derivedForm.blank());

// Call the inherited method.
var baseForm : IForm = derivedForm;
print(baseForm.blank());
```

このプログラムによる出力は次のとおりです。

```
25
This is blank.
This is blank.
```

例 2

この例では、関数 `printFacts` は入力として **String** を受け取ります。また、パラメータ配列を使用して可変数の **Objects** を受け取ります。

```
        function printFacts(name : String, ... info : Object[]) {
   print("Name: " + name);
   print("Number of extra information: " + info.length);
   for (var factNum in info) {
      print(factNum + ": " + info[factNum]);
   }
}

// Pass several arguments to the function.
printFacts("HAL 9000", "Urbana, Illinois", new Date(1997,0,12));
// Here the array is intrepeted as containing arguments for the function.
printFacts("monolith", [1, 4, 9]);
// Here the array is just one of the arguments.
printFacts("monolith", [1, 4, 9], "dimensions");
printFacts("monolith", "dimensions are", [1, 4, 9]);
```

このプログラムを実行すると、次の出力が表示されます。

```
Name: HAL 9000
Number of extra information: 2
0: Urbana, Illinois
1: Sun Jan 12 00:00:00 PST 1997
Name: monolith
Number of extra information: 3
0: 1
1: 4
2: 9
Name: monolith
Number of extra information: 2
0: 1,4,9
1: dimensions
Name: monolith
Number of extra information: 2
0: dimensions are
1: 1,4,9
```

必要条件

バージョン 1 (構文 1)バージョン .NET (構文 2 および構文 3)

参照

関連項目

new 演算子
class ステートメント
interface ステートメント
return ステートメント

概念

変数と定数のスコープ
型の注釈
型指定された配列

その他の技術情報

修飾子

function get ステートメント

クラスまたはインターフェイスで、新しいプロパティのアクセサを宣言します。**function get** は、**function set** と共に使用されることが多く、プロパティへの読み取り/書き込みアクセスを許可します。

```
// Syntax for the get accessor for a property in a class.
 [modifiers] function get propertyname() [: type] {
   [body]
}

// Syntax for the get accessor for a property in an interface.
[modifiers] function get propertyname() [: type]
```

引数

modifiers

省略可能です。プロパティの参照可能範囲と動作を制御する修飾子。

propertyname

必ず指定します。作成するプロパティの名前。クラス内で一意の名前にする必要があります。ただし、読み取り/書き込みを行うプロパティを識別するために、同じ *propertyname* を **get** アクセサと **set** アクセサ両方に対して使用することはできます。

type

省略可能です。**get** アクセサが返す型。定義する場合は、**set** アクセサのパラメータ型と一致させる必要があります。

body

省略可能です。**get** アクセサの動作を定義する 1 つ以上のステートメント。

解説

オブジェクトのプロパティは、フィールドの場合とほとんど同じ方法でアクセスされます。ただし、プロパティでは、オブジェクトに格納されている値やオブジェクトから返される値を、より詳細に制御できます。クラスで定義される **get** および **set** プロパティ アクセサの組み合わせによって、プロパティは、読み取り専用、書き込み専用、または読み書き可能のいずれかになります。多くの場合、プロパティは、**private** フィールドまたは **protected** フィールドに適切な値だけを格納するために使用されます。読み取り専用のプロパティに値を代入したり、書き込み専用のプロパティから値を読み取ることはできません。

get アクセサは、戻り値の型を指定する必要があり、引数を持ちません。**get** アクセサは、**set** アクセサとペアで使用することもできます。set アクセサは引数を 1 つ受け取り、戻り値の型を持ちません。プロパティに両方のアクセサが使用されている場合は、**get** アクセサの戻り値の型と、**set** アクセサの引数の型が一致している必要があります。

プロパティは、**get** アクセサと **set** アクセサのいずれか、または両方を持ちます。プロパティの **get** アクセサ (**get** アクセサがない場合は **set** アクセサ) だけが、プロパティ全体に適用されるカスタム属性を持ちます。**get** および **set** アクセサはどちらも、各アクセサに適用される修飾子とカスタム属性を持つことができます。プロパティ アクセサはオーバーロードされませんが、隠ぺいまたはオーバーライドされます。

プロパティは **interface** の定義に指定できますが、インターフェイスに実装は与えられません。

使用例

いくつかのプロパティ宣言の例を次に示します。`Age` プロパティは、読み書き可能として定義されています。読み取り専用の `FavoriteColor` プロパティも定義されています。

```
class CPerson {
   // These variables are not accessible from outside the class.
   private var privateAge : int;
   private var privateFavoriteColor : String;

   // Set the initial favorite color with the constructor.
   function CPerson(inputFavoriteColor : String) {
      privateAge = 0;
      privateFavoriteColor = inputFavoriteColor;
   }

   // Define an accessor to get the age.
```

```
    function get Age() : int {
       return privateAge;
    }
    // Define an accessor to set the age, since ages change.
    function set Age(inputAge : int) {
       privateAge = inputAge;
    }

    // Define an accessor to get the favorite color.
    function get FavoriteColor() : String {
       return privateFavoriteColor;
    }
    // No accessor to set the favorite color, making it read only.
    // This assumes that favorite colors never change.
}

var chris: CPerson = new CPerson("red");

// Set Chris age.
chris.Age = 27;
// Read chris age.
print("Chris is " + chris.Age + " years old.");

// FavoriteColor can be read from, but not written to.
print("Favorite color is " + chris.FavoriteColor + ".");
```

このプログラムを実行すると、次のように表示されます。

```
Chrisis 27 years old.
Favorite color is red.
```

必要条件

バージョン .NET

参照

関連項目

class ステートメント
interface ステートメント
function ステートメント
function set ステートメント

概念

型の注釈

その他の技術情報

修飾子

function set ステートメント

クラスまたはインターフェイスで、新しいプロパティのアクセサを宣言します。**function set** は **function get** と共に使用されることが多く、プロパティへの読み取り/書き込みアクセスを許可します。

```
// Syntax for the set accessor of a property in a class.
 [modifiers] function set propertyname(parameter [: type]) {
   [body]
}

// Syntax for the set accessor of a property in an interface.
[modifiers] function set propertyname(parameter [: type])
```

引数

modifiers

省略可能です。プロパティの参照可能範囲と動作を制御する修飾子。

propertyname

必ず指定します。作成するプロパティの名前。クラス内で一意の名前にする必要があります。ただし、読み取り/書き込みを行うプロパティを識別するために、同じ *propertyname* を **get** アクセサと **set** アクセサ両方に対して使用することはできます。

parameter

必ず指定します。**set** アクセサが受け取る仮パラメータ。

type

省略可能です。**set** アクセサのパラメータの型。定義する場合は、**get** アクセサの戻り値の型と一致させる必要があります。

body

省略可能です。**set** アクセサの動作を定義する 1 つ以上のステートメント。

解説

オブジェクトのプロパティは、フィールドの場合とほとんど同じ方法でアクセスされます。ただし、プロパティでは、オブジェクトに格納されている値やオブジェクトから返される値を、より詳細に制御できます。クラスで定義される **get** および **set** プロパティ アクセサの組み合わせによって、プロパティは、読み取り専用、書き込み専用、または読み書き可能のいずれかになります。多くの場合、プロパティは、**private** フィールドまたは **protected** フィールドに適切な値だけを格納するために使用されます。読み取り専用のプロパティに値を代入したり、書き込み専用のプロパティから値を読み取ることはできません。

set アクセサの引数は 1 つだけで、戻り値の型は指定できません。**set** アクセサは、**get** アクセサとペアで使用することもできます。get アクセサは引数を持たず、戻り値の型を指定する必要があります。プロパティに両方のアクセサが使用されている場合は、**get** アクセサの戻り値の型と、**set** アクセサの引数の型が一致している必要があります。

プロパティは、**get** アクセサと **set** アクセサのいずれか、または両方を持ちます。プロパティの **get** アクセサ (**get** アクセサがない場合は **set** アクセサ) だけが、プロパティ全体に適用されるカスタム属性を持ちます。**get** および **set** アクセサはどちらも、各アクセサに適用される修飾子とカスタム属性を持つことができます。プロパティ アクセサはオーバーロードされませんが、隠ぺいまたはオーバーライドされます。

プロパティは **interface** の定義に指定できますが、インターフェイスに実装は与えられません。

使用例

いくつかのプロパティ宣言の例を次に示します。`Age` プロパティは、読み書き可能として定義されています。読み取り専用の `FavoriteColor` プロパティも定義されています。

```
class CPerson {
   // These variables are not accessible from outside the class.
   private var privateAge : int;
   private var privateFavoriteColor : String;

   // Set the initial favorite color with the constructor.
   function CPerson(inputFavoriteColor : String) {
      privateAge = 0;
```

```
        privateFavoriteColor = inputFavoriteColor;
    }

    // Define an accessor to get the age.
    function get Age() : int {
        return privateAge;
    }
    // Define an accessor to set the age, since ages change.
    function set Age(inputAge : int) {
        privateAge = inputAge;
    }

    // Define an accessor to get the favorite color.
    function get FavoriteColor() : String {
        return privateFavoriteColor;
    }
    // No accessor to set the favorite color, making it read only.
    // This assumes that favorite colors never change.
}

var chris : CPerson = new CPerson("red");

// Set Chris's age.
chris.Age = 27;
// Read Chris's age.
print("Chris is " + chris.Age + " years old.");

// FavoriteColor can be read from, but not written to.
print("Favorite color is " + chris.FavoriteColor + ".");
```

このプログラムを実行すると、次のように表示されます。

```
Chris is 27 years old.
Favorite color is red.
```

必要条件

バージョン .NET

参照

関連項目

class ステートメント
interface ステートメント
function ステートメント
function get ステートメント

概念

型の注釈

その他の技術情報

修飾子

@if...@elif...@else...@end ステートメント

条件式の値を評価し、条件に応じて適切なステートメントを実行します。

```
@if (
   condition1
)
   text1
[@elif (
   condition2
)
   text2]
[@else
   text3]
@end
```

引数
condition1, condition2

必ず指定します。ブール式に強制変換できる式を指定します。

text1

省略可能です。*condition1* に指定した条件が真 (**true**) の場合に解析するテキストを指定します。

text2

省略可能です。*condition1* に指定した条件が偽 (**false**) で *condition2* に指定した条件が真 (true) の場合に解析するテキストを指定します。

text3

省略可能です。*condition1* も *condition2* も偽 (**false**) の場合に解析するテキストを指定します。

解説

@if ステートメントの句は、**@if** ステートメントと同じ行に記述する必要があります。複数の **@elif** 句を使用することもできます。ただし、**@elif** 句はすべて **@else** 句より前に記述する必要があります。

@if ステートメントは、テキスト出力に使用するテキストを複数の候補の中から選択する場合などによく使用されます。

使用例

次のコードは、**@if...@else...@end** ステートメントの使用例です。

```
      @if (@_win32)
   print("Operating system is 32-bit.");
@else
   print("Operating system is not 32-bit.");
@end
```

必要条件

Version 3

参照
関連項目
@cc_on ステートメント
@set ステートメント
概念
条件付きコンパイル変数
その他の技術情報
条件付きコンパイル

if...else ステートメント

条件式の値を評価し、条件に応じて適切なステートメントを実行します。

```
if (condition)
      statement1
[else
      statement2]
```

引数

condition

必ず指定します。ブール式を指定します。この式が NULL または undefined の場合は、偽 (**false**) となります。

statement1

必ず指定します。*condition* が真 (**true**) の場合に実行するステートメントを指定します。複合ステートメントを指定することもできます。

statement2

省略可能です。*condition* が偽 (**false**) の場合に実行するステートメントを指定します。複合ステートメントを指定することもできます。

解説

statement1 および *statement2* に対し、必ず中かっこ ({ }) で囲む習慣を付けておくと、コードが読みやすくなり、不注意によるエラーも減らすことができます。

使用例

たとえば、次のコード例のような場合、最初の **if** ステートメントに対して **else** を使用しているつもりでいても、実際には **else** は 2 つ目の **if** ステートメントに使用されます。

```
          if (x == 5)
   if (y == 6)
      z = 17;
else
   z = 20;
```

次のコード例のように書き換えると、あいまいさを排除できます。

```
          if (x == 5)
   {
   if (y == 6)
      z = 17;
   }
else
   z = 20;
```

また、*statement1* へステートメントを単純に追加する場合でも、次のコード例のように、ステートメントだけ追加して中かっこを記述し忘れると、エラーが発生します。

```
          if (x == 5)
   z = 7;
   q = 42;
else
   z = 19;
```

この場合は、**if** ステートメントと **else** ステートメントの間に複数のステートメントが記述されているため、構文エラーになります。**if** と **else** の間のステートメントを必ず中かっこで囲む必要があります。

必要条件

Version 1

参照

関連項目

条件 (三項) 演算子 (?:)

import ステートメント

現在のスクリプトまたは外部ライブラリのどちらかに含まれる名前空間へのアクセスを有効にします。

```
import namespace
```

引数
namespace

必ず指定します。インポートする名前空間の名前。

解説

import ステートメントは、*namespace* に指定された名前で、グローバル オブジェクトのプロパティを作成します。また、インポートされる名前空間に対応するオブジェクトを含むように、そのプロパティを初期化します。**import** ステートメントを使用して作成されたプロパティは、代入、削除、および列挙できません。すべての **import** ステートメントは、スクリプトの開始時に実行されます。

import ステートメントは、名前空間をスクリプトで利用できるようにします。名前空間は、スクリプトで **package** ステートメントを使用して定義するか、外部アセンブリによって提供されます。名前空間がスクリプト内に見つからないときは、指定したアセンブリ ディレクトリ内で、この名前空間と同じ名前のアセンブリが検索されます。ただし、プログラムがコンパイル中でなく、/autoref オプションがオフでない場合に限ります。たとえば、名前空間 `Acme.Widget.Sprocket` をインポートし、名前空間が現在のスクリプトで定義されていない場合、JScript は次のアセンブリで名前空間を検索します。

- `Acme.Widget.Sprocket.dll`
- `Acme.Widget.dll`
- `Acme.dll`

インクルードするアセンブリの名前は、明示的に指定できます。/autoref オプションがオフの場合、または名前空間の名前がアセンブリ名に一致しない場合は、明示的に指定する必要があります。コマンド ライン コンパイラは、/reference オプションを使用してアセンブリ名を指定します。ASP.NET では、同じ処理に **@ Import** および **@ Assembly** ディレクティブが使用されます。たとえば、アセンブリ mydll.dll を明示的にインクルードするには、コマンド ラインで次のように入力します。

```
jsc /reference:mydll.dll myprogram.js
```

ASP.NET ページからアセンブリをインクルードするには、次のように入力します。

```
<%@ Import namespace = "mydll" %>
<%@ Assembly name = "mydll" %>
```

コードでクラスが参照されている場合、コンパイラは、最初にローカル スコープでクラスを検索します。一致するクラスが見つからなかった場合、コンパイラはそれぞれの名前空間でクラスを検索します。名前空間の検索は、インポートされた順に行われ、一致する項目が見つかった時点で終了します。クラスの完全限定名を使用すると、クラスの派生元の名前空間を正確に指定できます。

JScript は、入れ子になった名前空間を自動的にインポートしません。各名前空間は、完全限定名前空間を使用してインポートする必要があります。たとえば、`Outer` という名前空間のクラス、および `Outer.Inner` という入れ子になった名前空間のクラスにアクセスするには、両方の名前空間をインポートする必要があります。

使用例

次の例では、3 つの簡単なパッケージを定義し、スクリプトに名前空間をインポートします。一般的に、各パッケージは個別のアセンブリに存在し、パッケージ内容を保守および配布できるようになっています。

```
// Create a simple package containing a class with a single field (Hello).
package Deutschland {
   class Greeting {
      static var Hello : String = "Guten tag!";
   }
};
// Create another simple package containing two classes.
// The class Greeting has the field Hello.
```

```
// The class Units has the field distance.
package France {
   public class Greeting {
      static var Hello : String = "Bonjour!";
   }
   public class Units {
      static var distance : String = "meter";
   }
};
// Use another package for more specific information.
package France.Paris {
   public class Landmark {
      static var Tower : String = "Eiffel Tower";
   }
};

// Declare a local class that shadows the imported classes.
class Greeting {
   static var Hello : String = "Greetings!";
}

// Import the Deutschland, France, and France.Paris packages.
import Deutschland;
import France;
import France.Paris;

// Access the package members with fully qualified names.
print(Greeting.Hello);
print(France.Greeting.Hello);
print(Deutschland.Greeting.Hello);
print(France.Paris.Landmark.Tower);
// The Units class is not shadowed, so it can be accessed with or without a fully qualified
name.
print(Units.distance);
print(France.Units.distance);
```

このスクリプトの出力は次のようになります。

```
Greetings!
Bonjour!
Guten tag!
Eiffel Tower
meter
meter
```

必要条件

バージョン .NET

参照

関連項目

package ステートメント
/autoref
/lib
@ Assembly
@ Import

interface ステートメント

インターフェイスの名前と、インターフェイスを構成するプロパティおよびイベントを宣言します。

```
[modifiers] interface interfacename [implements baseinterfaces] {
   [interfacemembers]
}
```

引数

modifiers

省略可能です。プロパティの参照可能範囲と動作を制御する修飾子。

interfacename

必ず指定します。**interface** の名前。標準的な変数の名前付け規則に従って名前を付けます。

implements

省略可能です。名前付きのインターフェイスが、定義済みのインターフェイスを実装するか、定義済みのインターフェイスにメンバを追加するかどうかを示すキーワード。このキーワードが指定されていない場合、標準の JScript 基本インターフェイスが作成されます。

baseinterfaces

省略可能です。*interfacename* で実装されるインターフェイス名のコンマ区切りのリスト。

interfacemembers

省略できます。**function** ステートメントで定義されるメソッドの宣言、または **function get** ステートメントと **function set** ステートメントで定義されるプロパティの宣言。

解説

JScript の **interface** 宣言の構文は、**class** 宣言の構文と似ています。インターフェイスは、クラスと同様にすべてのメンバが **abstract** であり、関数本体を持たないプロパティの宣言とメソッドの宣言だけを含むことができます。**interface** ステートメントは、フィールドの宣言、初期化子の宣言、または入れ子になったクラスの宣言を含むことができません。**interface** は、**implements** キーワードを使用して 1 つ以上の **interface** を実装できます。

class ステートメントは 1 つの基本クラスしか拡張できませんが、複数の **interface** を実装することはできます。**class** で複数の **interface** を実装することにより、C++ などの他のオブジェクト指向言語よりも簡単に多重継承を作成できます。

使用例

次のコードは、1 つの実装がどのようにして複数の interface で継承されるかを示しています。

```
          interface IFormA {
   function displayName();
}

// Interface IFormB shares a member name with IFormA.
interface IFormB {
   function displayName();
}

// Class CForm implements both interfaces, but only one implementation of
// the method displayName is given, so it is shared by both interfaces and
// the class itself.
class CForm implements IFormA, IFormB {
   function displayName() {
      print("This the form name.");
   }
}

// Three variables with different data types, all referencing the same class.
var c : CForm = new CForm();
```

```
var a : IFormA = c;
var b : IFormB = c;

// These do exactly the same thing.
a.displayName();
b.displayName();
c.displayName();
```

このプログラムの出力は次のようになります。

```
This the form name.
This the form name.
This the form name.
```

必要条件

バージョン .NET

参照

関連項目

class ステートメント
function ステートメント
function get ステートメント
function set ステートメント

その他の技術情報

修飾子

ラベル付きステートメント

ステートメントの識別子を指定します。

```
label :
   [statements]
```

引数
label

必ず指定します。ラベル付きステートメントを参照するときに使用する、固有の識別子を指定します。

statements

省略可能です。*label* に関連する 1 つ以上のステートメントを指定します。

解説

ラベルは、**break** と **continue** が適用されるステートメントを指定するために、**break** ステートメントと **continue** ステートメントによって使用されます。

使用例

次のコードは、**continue** ステートメントと共にラベル付きステートメントを使用する例です。配列に値を代入する際にラベル付きステートメントを利用して、各行の 3 列目の要素にだけ値を代入しません。

```
function labelDemo() {
   var a = new Array();
   var i, j, s = "", s1 = "";
   Outer:
   for (i = 0; i < 5; i++) {
      Inner:
      for (j = 0; j < 5; j++) {
         if (j == 2)
            continue Inner;
         else
            a[i,j] = j + 1;
      }
   }
   for (i = 0;i < 5; i++) {
      s = ""
      for (j = 0; j < 5; j++) {
         s += a[i,j];
      }
      s1 += s + "\n";
   }
   return(s1)
}
```

必要条件

Version 3

参照
関連項目
break ステートメント
continue ステートメント

package ステートメント

JScript パッケージを作成して、名前付きのコンポーネントを扱いやすいようにパッケージ化します。

```
package pname {
   [[modifiers1] pmember1]
   ...
   [[modifiersN] pmemberN]
}
```

引数

pname

必ず指定します。作成するパッケージの名前。

modifiers1, ..., modifiersN

省略可能です。*pmember* の参照可能範囲と動作を制御する修飾子。

pmember1, ..., pmemberN

省略可能です。クラス、インターフェイス、または列挙型の定義。

解説

パッケージ内では、クラス、インターフェイス、および列挙型だけが許可されています。パッケージ メンバには、可視性修飾子が指定されて、アクセスが制御されている場合があります。特に、**internal** 修飾子は、メンバを現在のパッケージ内だけで参照できるように指定します。

パッケージがインポートされると、パッケージ メンバに名前で直接アクセスできるようになります。ただし、インポートするスコープに参照可能な同じ名前を持つ宣言が他にある場合は、名前だけではアクセスできません。この場合は、パッケージ名を使用してメンバを修飾する必要があります。

JScript は、入れ子になったパッケージの宣言をサポートしていません。パッケージの内部には、クラス、インターフェイス、および列挙型の宣言だけを指定できます。パッケージ名に '.' 文字を含めることによって、他のパッケージに入れ子になることを示すことができます。たとえば、`Outer` というパッケージと `Outer.Inner` というパッケージには、相互に特別な関係は必要ありません。どちらもグローバル スコープのパッケージです。ただし、`Outer.Inner` は `Outer` に入れ子になると見なされることを暗黙的に示しています。

使用例

次の例では、3 つの簡単なパッケージを定義し、スクリプトに名前空間をインポートします。一般的に、各パッケージは個別のアセンブリに存在し、パッケージ内容を保守および配布できるようになっています。

```
// Create a simple package containing a class with a single field (Hello).
package Deutschland {
   class Greeting {
      static var Hello : String = "Guten tag!";
   }
};
// Create another simple package containing two classes.
// The class Greeting has the field Hello.
// The class Units has the field distance.
package France {
   public class Greeting {
      static var Hello : String = "Bonjour!";
   }
   public class Units {
      static var distance : String = "meter";
   }
};
// Use another package for more specific information.
package France.Paris {
   public class Landmark {
      static var Tower : String = "Eiffel Tower";
   }
};
```

```
// Declare a local class that shadows the imported classes.
class Greeting {
   static var Hello : String = "Greetings!";
}

// Import the Deutschland, France, and France.Paris packages.
import Deutschland;
import France;
import France.Paris;

// Access the package members with fully qualified names.
print(Greeting.Hello);
print(France.Greeting.Hello);
print(Deutschland.Greeting.Hello);
print(France.Paris.Landmark.Tower);
// The Units class is not shadowed, so it can be accessed with or without a fully qualified
name.
print(Units.distance);
print(France.Units.distance);
```

このスクリプトの出力は次のようになります。

```
Greetings!
Bonjour!
Guten tag!
Eiffel Tower
meter
meter
```

必要条件

バージョン .NET

参照

関連項目

import ステートメント

internal 修飾子

その他の技術情報

修飾子

print ステートメント

コンソールに文字列とそれに続く改行文字を送信します。

```
function print(str : String)
```

パラメータ
str

省略可能です。コンソールに送信する文字列。

解説

print ステートメントを使用すると、JScript コマンド ライン コンパイラ (`jsc.exe`) でコンパイルされた JScript プログラムからのデータを表示できます。**print** ステートメントは、文字列をパラメータとして受け取り、コンソールに送信して、文字列とそれに続く改行文字を表示します。

print ステートメントに渡す文字列にエスケープ シーケンスを使用すると、出力に書式を設定できます。エスケープ シーケンスは、先頭に円記号 (\) が指定されている、アルファベットや数字で構成される文字の組み合わせです。エスケープ シーケンスを使用して、キャリッジ リターンやタブ移動などの動作を指定できます。エスケープ シーケンスの詳細については、**String** オブジェクトのトピックを参照してください。コンソール出力の書式を詳細に制御する必要がある場合は、**System.Console.WriteLine** メソッドを使用できます。

JScript コマンド ライン コンパイラ (jsc.exe) では、**print** ステートメントは既定で有効です。ASP.NET では、**print** ステートメントは無効です。このステートメントをコマンド ライン コンパイラで無効にする場合は、/print- オプションを使用します。

出力先のコンソールがない (たとえば、Windows GUI アプリケーション) 場合、**print** ステートメントは、何も表示せずに失敗します。

print ステートメントからの出力は、コマンド ラインからファイルにリダイレクトできます。プログラムの出力がリダイレクトされると予想される場合は、出力される各行の終わりに \r エスケープ文字を使用してください。これにより、出力が正しい書式でファイルにリダイレクトされます。ただし、コンソールに表示される出力には影響しません。

使用例

print ステートメントの使用例を次に示します。

```
var name : String = "Fred";
var age : int = 42;
// Use the \t (tab) and \n (newline) escape sequences to format the output.
print("Name: \t" + name + "\nAge: \t" + age);
```

このスクリプトの出力は次のようになります。

```
Name:   Fred
Age:    42
```

参照
関連項目
/print
String オブジェクト
Console Class
概念
コマンド ライン プログラムからの表示

return ステートメント

現在の関数の実行を終了し、戻り値を返します。

```
return[(][expression][)]
```

引数
expression

省略可能です。関数から返す値を指定します。省略した場合、関数は戻り値を返しません。

解説

return ステートメントを使用すると、関数の実行を中止し、*expression* の値を返すことができます。*expression* を省略した場合および関数内で **return** ステートメントが実行されなかった場合は、関数を呼び出した**式**が代入されます。

return ステートメントを実行すると、関数本体に他のステートメントが残っている場合でも関数は終了します。ただし、**try** ブロック内にある **return** ステートメントを実行した場合で、対応する **finally** ブロックが記述されている場合は、**finally** ブロックのコードを実行してから関数が終了します。

return ステートメントを実行せずに、関数本体の最後に到達して関数が終了した場合は、戻り値は **undefined** 値になります (この場合、関数の戻り値を式の一部として使用できません)。

メモ :

finally ブロックのコードは、**try** または **catch** ブロックの **return** ステートメントが見つかり、その **return** ステートメントが実行される前に実行されます。この場合、**finally** ブロックに **return** ステートメントがあると、**try** または **catch** ブロックの **return** ステートメントよりも先に実行され、戻り値が異なる場合があります。このような状況を避けるために、**finally** ブロックには **return** ステートメントを使用しないでください。

使用例

次のコードは、**return** ステートメントの使用例です。

```
function myfunction(arg1, arg2){
   var r;
   r = arg1 * arg2;
   return(r);
}
```

必要条件

Version 1

参照
関連項目
function ステートメント
try...catch...finally ステートメント

@set ステートメント

条件付きコンパイル ステートメントで使用する変数を作成します。

```
@set @varname = term
```

引数
varname

必ず指定します。JScript で有効な変数名を指定します。必ず先頭に "@" という文字を記述します。

term

必ず指定します。0 個以上の単項演算子に続けて、定数、条件コンパイル変数、またはかっこで囲んだ式を指定します。

解説

条件コンパイルでは、数値変数とブール変数がサポートされています。文字列はサポートされていません。通常、**@set** ステートメントで作成した変数は、条件コンパイルの中で使用しますが、Jscript コードのどの場所でも使用できます。

変数宣言のコード例を次に示します。

```
        @set @myvar1 = 12
@set @myvar2 = (@myvar1 * 20)
@set @myvar3 = @_jscript_version
```

かっこで囲んだ式の中で使用できる演算子は次のとおりです。

- `! ~`
- `* / %`
- `+ -`
- `<< >> >>>`
- `< <= > >=`
- `== != === !==`
- `& ^ |`
- `&& | |`

まだ定義していない変数を使用すると、その値は **NaN** になります。**@if** ステートメントを次のコード例のように使用すると、値が **NaN** かどうかを確認できます。

```
@if (@newVar != @newVar)
   // ...
```

NaN は、自身と比較しても等しいと評価されない唯一の値で、コードによって確認できます。

必要条件

Version 3

参照
関連項目
@cc_on ステートメント
@if...@elif...@else...@end ステートメント
概念
条件付きコンパイル変数
その他の技術情報

条件付きコンパイル

static ステートメント

クラス宣言の内部で新しいクラス初期化子を宣言します。

```
static identifier {
   [body]
}
```

引数
identifier

必ず指定します。初期化子ブロックを含むクラスの名前。

body

省略可能です。初期化子ブロックを構成するコード。

解説

static 初期化子は、使用する前の **class** オブジェクト (オブジェクト インスタンスではありません) を初期化します。この初期化は一度だけ実行され、**static** 修飾子を持つクラスのフィールドを初期化するために使用できます。

static フィールド宣言を使用すると、複数の **static** 初期化子ブロックを、クラスのさまざまな場所に記述できます。クラスを初期化するときには、すべての **static** ブロックと **static** フィールド初期化子が、クラス本体に記述されている順に実行されます。この初期化は、**static** フィールドが最初に参照される前に実行されます。

static 修飾子と **static** ステートメントを混同しないでください。**static** 修飾子は、メンバがクラスのインスタンスではなく、クラス自身に属していることを示します。

使用例

static 初期化子を使用して、一度だけ実行する必要のある計算を実行する、簡単な **class** 宣言の例を次に示します。この例では、階乗の表が一度だけ計算されています。階乗が必要になった場合は、表から読み取ります。プログラム中で大きな階乗が何度も必要になる場合は、繰り返し階乗を計算するよりも、この手法を使用する方が高速です。

static 修飾子は、階乗のメソッドに使用されています。

```
class CMath {
   // Dimension an array to store factorial values.
   // The static modifier is used in the next two lines.
   static const maxFactorial : int = 5;
   static const factorialArray : int[] = new int[maxFactorial];

   static CMath {
      // Initialize the array of factorial values.
      // Use factorialArray[x] = (x+1)!
      factorialArray[0] = 1;
      for(var i : int = 1; i< maxFactorial; i++) {
         factorialArray[i] = factorialArray[i-1] * (i+1);
      }
      // Show when the initializer is run.
      print("Initialized factorialArray.");
   }

   static function factorial(x : int) : int {
      // Should have code to check that x is in range.
      return factorialArray[x-1];
   }
};

print("Table of factorials:");

for(var x : int = 1; x <= CMath.maxFactorial; x++) {
   print( x + "! = " + CMath.factorial(x) );
}
```

このコードの出力は次のようになります。

```
Table of factorials:
Initialized factorialArray.
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

必要条件

バージョン .NET

参照

関連項目

class ステートメント

static 修飾子

super ステートメント

現在のオブジェクトの基本オブジェクトを参照します。このステートメントは、次の 2 つのコンテキストで使用できます。

```
// Syntax 1: Calls the base-class constructor with arguments.
super(arguments)

// Syntax 2: Accesses a member of the base class.
super.member
```

引数
arguments

構文 1 の省略可能な引数。基本クラスのコンストラクタを指定するコンマ区切りの引数のリストです。

member

構文 2 の必ず指定する引数。アクセスする基本クラスのメンバです。

解説

super キーワードは、一般的に、次の 2 つの状況のどちらかで使用します。1 つ以上の引数を指定して、明示的に基本クラスのコンストラクタを呼び出します。または、現在のクラスでオーバーライドされている基本クラスのメンバにアクセスします。

例 1

次の例では、**super** は基本クラスのコンストラクタを参照しています。

```
class baseClass {
   function baseClass() {
      print("Base class constructor with no parameters.");
   }
   function baseClass(i : int) {
      print("Base class constructor. i is "+i);
   }
}
class derivedClass extends baseClass {
   function derivedClass() {
      // The super constructor with no arguments is implicitly called here.
      print("This is the derived class constructor.");
   }
   function derivedClass(i : int) {
      super(i);
      print("This is the derived class constructor.");
   }
}

new derivedClass;
new derivedClass(42);
```

このプログラムを実行すると、次の出力が表示されます。

```
Base class constructor with no parameters.
This is the derived class constructor.
Base class constructor. i is 42
This is the derived class constructor.
```

例 2

次の例では、**super** は基本クラスのオーバーライドされたメンバにアクセスします。

```
class baseClass {
   function test() {
```

```
      print("This is the base class test.");
   }
}
class derivedClass extends baseClass {
   function test() {
      print("This is the derived class test.");
      super.test(); // Call the base class test.
   }
}

var obj : derivedClass = new derivedClass;
obj.test();
```

このプログラムを実行すると、次の出力が表示されます。

```
This is the derived class test.
This is the base class test.
```

必要条件

バージョン .NET

参照

関連項目

new 演算子

this ステートメント

switch ステートメント

指定した式の値がラベルと一致したときに 1 つ以上のステートメントを実行する機能を提供します。

```
switch (expression) {
      case label1 :
      [statementlist1]
      [break;]
 [ ...
 [ case labelN :
      [statementlistN]
      [break;] ] ]
 [ default :
      [statementlistDefault]]
}
```

引数
expression

必ず指定します。評価される式を指定します。

label1, ..., labelN

必ず指定します。*expression* に指定した式と一致するかどうかを調べられる識別子を指定します。*label* === *expression* が成立すると、コロンの直後のステートメント リストの実行が開始され、**break** ステートメント (省略可能です) が見つかった箇所か、**switch** ステートメントの最後まで実行が続行されます。

statementlist1, ..., statementlistN, statementlistDefault

省略可能です。実行する 1 つ以上のステートメントを指定します。

解説

どのラベルの値も式の値と一致しなかった場合に実行するステートメントを指定するには、**default** 句を使用します。default 句は、**switch** コード ブロック内のどこでも記述できます。

label で指定するブロックの数に制限はありません。式の値がどのラベルの値にも一致せず、**default** 句を指定していなかった場合は、ステートメントは何も実行されません。

switch ステートメントでの実行の流れは次のようになります。

- *expression* に指定した式が評価され、この式に一致するラベルが見つかるまで、順序どおりにラベルが評価されます。
- 式と一致するラベルがある場合は、その直後に記述されたステートメント リストが実行されます。

  実行は、**break** ステートメントが実行されるか、**switch** ステートメントが終了するまで続行されます。つまり、**break** ステートメントを記述しない場合は、複数の *label* ブロックが実行されます。
- 式と一致するラベルが 1 つもない場合は、**default** 句へ進みます。**default** 句がない場合は、最後の処理へ進みます。
- **switch** コード ブロックの次のステートメントから処理を続行します。

使用例

オブジェクトの型を調べる ASP.NET の例を次に示します。この例では 1 種類の型しか使用していませんが、他のオブジェクト型に対する関数がどのように機能するかは簡単にわかります。

```
<%@ language="jscript" %>
<%
var d = new Number();
function MyObjectType(obj : Object) : String {
   switch (obj.constructor){
      case Date:
         return "Object is a Date.";
         break;
      case Number:
```

```
            return "Object is a Number.";
            break;
        case String:
            return "Object is a String.";
            break;
        default:
            return "Object is unknown.";
    }
}
Response.Write(MyObjectType(d));
%>
```

必要条件

Version 3

参照

関連項目

break ステートメント

if...else ステートメント

this ステートメント

現在のオブジェクトを返します。

```
this.property
```

引数
property

必ず指定します。現在のオブジェクトのプロパティの識別子を指定します。

解説

通常、キーワード **this** は、現在のオブジェクトを参照するために、オブジェクト コンストラクタに使用します。

使用例

次のコードは、**this** ステートメントを使って新しく作成された Car オブジェクトを示している例です。このコードでは、3 つのプロパティに値を代入しています。

```
function Car(color, make, model){
   this.color = color;
   this.make = make;
   this.model = model;
}
```

クライアント版 JScript では、どのオブジェクトのコンテキストでもない場合、**this** ステートメントは現在のウィンドウを表す **window** オブジェクトを示します。

必要条件

Version 1

参照
関連項目
new 演算子

throw ステートメント

try...catch...finally ステートメントで処理できるエラー条件を生成します。

```
throw [exception]
```

引数
exception

省略可能です。任意の式を指定します。

解説

throw ステートメントは、**catch** ブロック内にある場合にだけ、引数なしで使用できます。その場合、**throw** ステートメントは、外側の **catch** ステートメントでキャッチされたエラーを再スローします。引数が指定されている場合、**throw** ステートメントは *exception* の値をスローします。

使用例

渡された値に基づいてエラーをスローし、そのエラーを **try...catch...finally** ステートメントの階層で処理する例を次に示します。

```
function TryCatchDemo(x){
   try {
      try {
      if (x == 0)                            // Evalute argument.
         throw "x equals zero";              // Throw an error.
      else
         throw "x does not equal zero";      // Throw a different error.
      }
      catch(e) {                             // Handle "x=0" errors here.
         if (e == "x equals zero")           // Check for a handled error.
            return(e + " handled locally."); // Return error message.
         else                                // Can't handle error here.
            throw e;                         // Rethrow the error for next
      }                                      // error handler.
   }
   catch(e) {                                // Handle other errors here.
      return(e + " error handled higher up."); // Return error message.
   }
}
print(TryCatchDemo(0)+ "\n");
print(TryCatchDemo(1));
```

必要条件

Version 5

参照
関連項目
try...catch...finally ステートメント
Error オブジェクト

try...catch...finally ステートメント

JScript のエラー処理機能を実装します。

```
try {
   [tryStatements]
} catch(exception) {
   [catchStatements]
} finally {
   [finallyStatements]}
```

引数

tryStatements

省略可能です。エラーが発生する可能性のあるステートメントを指定します。

exception

必ず指定します。任意の変数名を指定します。*exception* の初期値は、スローされたエラーの値です。

catchStatements

省略可能です。*tryStatement* で発生しているエラーを処理するためのステートメントを指定します。

finallyStatements

省略可能です。他のエラー処理がすべて発生すると無条件に実行されるステートメントを指定します。

解説

try...catch...finally ステートメントでは、コード内の所定のブロックで発生する可能性のあるエラーの一部またはすべてに対して、コードを実行しながらエラー処理を実行できます。プログラマによって対処されていないエラーが発生した場合は、エラー処理が存在しなかったかのように、通常のエラー メッセージが表示されます。

引数 *tryStatements* にはエラーが発生する可能性のあるコードを指定し、引数 *catchStatements* には発生したエラーを処理するコードを指定します。*tryStatements* でエラーが発生した場合、*catchStatements* にプログラムの処理が渡されます。*exception* の初期値は、*tryStatements* で発生したエラーの値です。エラーが発生しないと、*catchStatements* は実行されません。

tryStatements で発生したエラーが、対応する *catchStatements* で処理できない場合は、そのエラーをより上位のエラー ハンドラに **throw** ステートメントで通知するかまたは再度スローしてください。

tryStatements のすべてのステートメントが実行され、*catchStatements* でいずれかのエラー処理が発生すると、*finallyStatements* のステートメントが無条件に実行されます。

finallyStatements 内部のコードは、**return** ステートメントが **try** ブロック内や **catch** ブロック内にある場合、または **catch** ブロックからエラーがスローされた場合にも実行されます。*finallyStatments* は、必ず実行されます。

使用例

JScript の例外処理の例を次に示します。

```
        try {
    print("Outer try running...");
    try {
       print("Nested try running...");
       throw "an error";
    } catch(e) {
       print("Nested catch caught " + e);
       throw e + " re-thrown";
    } finally {
       print("Nested finally is running...");
    }
} catch(e) {
   print("Outer catch caught " + e);
} finally {
```

```
    print("Outer finally running");
}
```

これによって次の文字列が出力されます。

```
Outer try running..
Nested try running...
Nested catch caught an error
Nested finally is running...
Outer catch caught an error re-thrown
Outer finally running
```

必要条件

Version 5

参照

関連項目

throw ステートメント

Error オブジェクト

var ステートメント

変数を宣言します。

```
// Syntax for declaring a variable of global scope or function scope.
var name1 [: type1] [= value1] [, ... [, nameN [: typeN] [= valueN] ]]

// Syntax for declaring a variable field within a class.
 [attributes] [modifiers] var name1 [: type1] [= value1] [, ... [, nameN [: typeN] [= value
N].]]
```

引数
attributes

省略可能です。フィールドの参照可能範囲と動作を制御する属性を指定します。

modifiers

省略可能です。フィールドの参照可能範囲と動作を制御する修飾子を指定します。

name1, ..., nameN

必ず指定します。宣言する変数の名前を指定します。

type1, ..., typeN

省略可能です。宣言する変数の型を指定します。

value1, ..., valueN

省略可能です。変数に代入する初期値を指定します。

解説

変数を宣言するには、var ステートメントを使用します。変数は特定のデータ型にバインドされ、タイプ セーフを提供します。これらの変数には、宣言時に値を代入できます。代入した値はスクリプト中で変更できます。明示的に初期化していない変数には、既定値の **undefined** (必要に応じて、変数の型に変換されます) が代入されます。

クラスの変数フィールドは、グローバル変数や関数変数に似ていますが、クラスのスコープを持ち、参照可能範囲と使用方法を制御するさまざまな属性を指定できます。

使用例

次のコードは、**var** ステートメントの使用例です。

```
class Simple {
   // A field declaration of the private Object myField.
   private var myField : Object;
   // Define sharedField to be a static, public field.
   // Only one copy exists, and is shared by all instances of the class.
   static public var sharedField : int = 42;
}
var index;
var name : String = "Thomas Jefferson";
var answer : int = 42, counter, numpages = 10;
var simpleInst : Simple = new Simple;
```

必要条件

Version 1

参照
関連項目
const ステートメント
function ステートメント
new 演算子

概念

変数と定数のスコープ

型の注釈

その他の技術情報

修飾子

while ステートメント

指定した条件が偽 (**false**) になるまでステートメントを繰り返し実行します。

```
while (expression)
      statement
```

引数
expression

必ず指定します。ループの各反復処理の前に評価するブール式を指定します。この式の評価が真 (**true**) の場合は、ループが実行されます。偽 (**false**) の場合は、ループ処理を終了します。

statement

必ず指定します。*expression* の評価が真 (**true**) の場合に実行するステートメントを指定します。複合ステートメントを指定することもできます。

解説

while ステートメントでは、ループが初めて実行される前に *expression* が調べられます。この時点で *expression* の評価が偽 (**false**) の場合は、ループは一度も実行されません。

使用例

次のコードは、**while** ステートメントの使用例です。

```
function BreakTest(breakpoint){
   var i = 0;
   while (i < 100) {
      if (i == breakpoint)
         break;
      i++;
   }
   return(i);
}
```

必要条件

Version 1

参照
関連項目
break ステートメント
continue ステートメント
do...while ステートメント
for ステートメント
for...in ステートメント

with ステートメント

ステートメントで使用する既定のオブジェクトを設定します。

```
with (object)
      statement
```

引数
object

必ず指定します。新しい既定のオブジェクトを指定します。

statement

必ず指定します。*object* を既定のオブジェクトとして使用するステートメントを指定します。複合ステートメントを指定することもできます。

解説

一般に、**with** ステートメントは特定の場面で、記述するコードの量を少なくするために使用します。

使用例

たとえば、次のコードは **Math** を繰り返し使用する例です。

```
var x, y;
x = Math.cos(3 * Math.PI) + Math.sin(Math.LN10);
y = Math.tan(14 * Math.E);
```

with ステートメントを使用すると、次のように簡単で読みやすいコードになります。

```
var x, y;
with (Math){
   x = cos(3 * PI) + sin (LN10);
   y = tan(14 * E);
}
```

必要条件

Version 1

参照
関連項目
this ステートメント

JScript コンパイラ オプション

JScript コンパイラは、実行可能 (.exe) ファイルとダイナミック リンク ライブラリ (.dll) を生成します。

各コンパイラ オプションには、それぞれ *-option* と */option* という 2 つの形式があります。ここでは */option* 形式だけを使用します。

このセクションの内容

JScript コンパイラ オプション一覧 (アルファベット順)

コンパイラ オプションをアルファベット順で一覧に示します。

JScript コンパイラ オプション一覧 (カテゴリ別)

コンパイラ オプションを出力ファイル、.NET Framework アセンブリ、デバッグ/エラー チェック、プリプロセッサ、リソース、およびその他のカテゴリ別に示します。

関連するセクション

コマンド ラインからのビルド

コマンド ラインから JScript アプリケーションをビルドする場合の、構文や結果などの詳細を説明します。

JScript コードの作成、コンパイル、およびデバッグ

Visual Studio 統合開発環境 (IDE: Integrated Development Environment) を使用して、JScript コードを作成および編集する方法について説明します。

JScript コンパイラ オプション一覧 (アルファベット順)

次のコンパイラ オプションは、アルファベット順に並んでいます。

コンパイラ オプション

オプション	目的
@ (応答ファイルの指定)	応答ファイルを指定します。
/autoref	インポートされた名前空間と同じ名前、または変数宣言時の型の注釈と同じ名前のアセンブリを自動的に参照します。
/codepage	コンパイルですべてのソース コード ファイルに使用するコード ページを指定します。
/debug	デバッグ情報を生成します。
/define	プリプロセッサ シンボルを定義します。
/fast	速度が最適化された出力ファイルを生成します。このファイルでは、以前のリリースの言語機能の一部がサポートされません。
/help、/?	標準出力にコンパイラ オプションの一覧を表示します。
/lcid	コンパイラ メッセージのコード ページを指定します。
/lib	/reference によって参照されるアセンブリの場所を指定します。
/linkresource	マネージ リソースへのリンクを作成します。
/nologo	コンパイラの開始メッセージが表示されないようにします。
/nostdlib	標準ライブラリ (mscorlib.dll) をインポートしません。
/out	出力ファイル名を指定します。
/platform (JScript)	プラットフォームの種類を指定します。
/print	print ステートメントを使用できるかどうかを指定します。
/reference	アセンブリを含むファイルからメタデータをインポートします。
/resource	マネージ リソースをアセンブリに埋め込みます。
/target	次の 3 つのオプションのいずれかを使用して、出力ファイルの形式を指定します。 /target:exe/target:library/target:winexe
/utf8output	UTF-8 エンコーディングを使用してコンパイラ出力を表示します。
/versionsafe	すべてのオーバーライドが明示的かどうかを確認します。
/warn	警告レベルを設定します。
/warnaserror	警告をエラーとして扱います。

/win32res	Win32 リソースを出力ファイルに挿入します。

参照

概念

JScript コンパイラ オプション一覧 (カテゴリ別)

コマンド ラインからのビルド

その他の技術情報

JScript コンパイラ オプション

JScript コンパイラ オプション一覧 (カテゴリ別)

次のコンパイラ オプションは、カテゴリ別に並んでいます。

コンパイラ オプション

出力ファイル

オプション	目的
/out	出力ファイル名を指定します。
/target	次の 3 つのオプションのいずれかを使用して、出力ファイルの形式を指定します。/target:exe/target:library/target:winexe

.NET Framework アセンブリ

オプション	目的
/autoref	インポートされた名前空間と同じ名前、または変数宣言時の型の注釈と同じ名前のアセンブリを自動的に参照します。
/lib	/reference によって参照されるアセンブリの場所を指定します。
/nostdlib	標準ライブラリ (mscorlib.dll) をインポートしません。
/reference	アセンブリを含むファイルからメタデータをインポートします。

デバッグ/エラーのチェック

オプション	目的
/debug	デバッグ情報を生成します。
/lcid	コンパイラ メッセージのコード ページを指定します。
/versionsafe	すべてのオーバーライドが明示的かどうかを確認します。
/warn	警告レベルを設定します。
/warnaserror	警告をエラーとして扱います。

プリプロセッサ

オプション	目的
/define	プリプロセッサ シンボルを定義します。

リソース

オプション	目的
/linkresource	マネージ リソースへのリンクを作成します。
/resource	マネージ リソースをアセンブリに埋め込みます。
/win32res	Win32 リソースを出力ファイルに挿入します。

その他

オプション	目的
@ (応答ファイルの指定)	応答ファイルを指定します。
/codepage	コンパイルですべてのソース コード ファイルに使用するコード ページを指定します。
/fast	速度が最適化された出力ファイルを生成します。このファイルでは、以前のリリースの言語機能の一部がサポートされません。

/help、/?	標準出力にコンパイラ オプションの一覧を表示します。
/nologo	コンパイラの開始メッセージが表示されないようにします。
/platform (JScript)	プラットフォームの種類を指定します。
/print	print ステートメントを定義するかどうかを指定します。
/utf8output	UTF-8 エンコーディングを使用してコンパイラ出力を表示します。

参照

概念

JScript コンパイラ オプション一覧 (アルファベット順)

コマンド ラインからのビルド

その他の技術情報

JScript コンパイラ オプション

コマンド ラインからのビルド

コンパイラは、コマンド ライン上でその実行可能ファイル (jsc.exe) の名前を入力することによって起動できます。詳細については、「コマンド ラインでの JScript コードのコンパイル」を参照してください。

コマンド ラインの例

- File.js をコンパイルして File.exe を作成するには、次のコードを使用します。

```
jsc File.js
```

- File.js をコンパイルして File.dll を作成するには、次のコードを使用します。

```
jsc /target:library File.js
```

- File.js をコンパイルして My.exe を作成するには、次のコードを使用します。

```
jsc /out:My.exe File.js
```

- `test.js` をコンパイルして .dll を作成するには、次のコードを使用します。

```
jsc /target:library test.js
```

参照

処理手順

方法 : コマンド ラインで JScript コードをコンパイルする

その他の技術情報

JScript コンパイラ オプション

@ (応答ファイルの指定)

応答ファイルを指定します。

```
@response_file
```

引数
response_file

コンパイラ オプションやコンパイルするソース コード ファイルの一覧を含むファイル。

解説

@ オプションを使用すると、コンパイラ オプションおよびコンパイルするソース コード ファイルを含むファイルを指定できます。これらのコンパイラ オプションおよびソース コード ファイルは、コマンド ラインで指定された場合と同様に、コンパイラによって処理されます。

コンパイル時に複数の応答ファイルを指定するには、複数の応答ファイル オプションを指定します。次に例を示します。

```
@file1.rsp @file2.rsp
```

応答ファイルでは、複数のコンパイラ オプションとソース コード ファイルを 1 行に記述できます。1 つのコンパイラ オプションは 1 行に指定する必要があり、複数行にわたって指定できません。

応答ファイルには、シャープ記号 (#) で始まるコメントを記述できます。

応答ファイルでのコンパイラ オプションの指定は、コマンド ラインでのコンパイラ オプションの指定とまったく同じです。詳細については、「コマンド ラインからのビルド」を参照してください。

コマンド ラインでコマンド オプションを指定した場合と同様に、コンパイラはこれらのオプションを出現順に処理します。したがって、1 つの応答ファイル内に含まれるオプションが、他の応答ファイル内のオプションや、コマンド ライン オプションと対応しない場合もあります。このような場合は、エラーになります。

応答ファイルを入れ子にすることはできません。応答ファイル内には、*@response_file* を配置できません。そのような場合は、JScript コンパイラがエラーを報告します。

使用例

サンプルの応答ファイルの一部を次に示します。

```
# build the first output file
/target:exe /out:MyExe.exe source1.js source2.js
```

参照
その他の技術情報
JScript コンパイラ オプション

/autoref

インポートされた名前空間と同じ名前、または変数宣言時の型の注釈と同じ名前のアセンブリを自動的に参照します。

```
/autoref[+ | -]
```

引数
+| -

/nostdlib+ を指定しない場合は、既定でオンになっています。/autoref+ または /autoref を指定すると、インポートされた名前空間と完全限定名に基づいて、アセンブリを自動的に参照します。

解説

/autoref オプションを指定した場合は、/reference にアセンブリを渡さなくても、コンパイラはアセンブリを参照します。import を使用して名前空間をインポートする場合、またはコード内で完全限定名を使用する場合、JScript コンパイラは型を含むアセンブリを検索します。JScript コンパイラがアセンブリを検索する方法については、/lib の説明を参照してください。

アセンブリの名前が作成中のプログラムの出力ファイルと同じ名前の場合、コンパイラはアセンブリを参照しません。

使用例

/autoref+ が有効な場合、次のプログラムはコンパイルされて動作します。変数を宣言するときに指定された型の注釈によって、コンパイラは System.dll を参照します。

```
var s: System.Collections.Specialized.StringCollection =
                    new System.Collections.Specialized.StringCollection();
print(s);
```

/autoref+ が有効な場合、次のプログラムはコンパイルされて動作します。**import** ステートメントによって、コンパイラは System.dll を参照します。

```
import System;
var s = new System.Collections.Specialized.StringCollection();
print(s);
```

上記の 2 つの例では、型の注釈または **import** ステートメントに基づいてコンパイラがアセンブリ名を検索する方法も示されています。StringCollection を含む System.Collections.Specialized.dll という名前のアセンブリが見つからなかった場合、コンパイラは System.Collections.dll を検索します。このファイルも見つからなかった場合は、System.dll が検索され、このファイルに StringCollection が見つかります。

参照
関連項目
import ステートメント
/reference
その他の技術情報
JScript コンパイラ オプション

/codepage

コンパイルですべてのソース コード ファイルに使用するコード ページを指定します。

```
/codepage:id
```

引数
id

コンパイル対象のすべてのソース コード ファイルに対するコード ページの ID。

解説

コンピュータ上の既定のコード ページを使用するようにファイルの作成時に指定していないソース コード ファイルを 1 つ以上コンパイルする場合は、/codepage オプションを使用して、どのコード ページを使用するかを指定できます。/codepage は、コンパイル対象のすべてのソース コード ファイルに適用されます。

コンピュータ上で有効になっているコード ページを使用してソース コード ファイルが作成されている場合、またはソース コード ファイルが UNICODE や UTF-8 で作成されている場合には、/codepage を使用する必要はありません。

参照
その他の技術情報
JScript コンパイラ オプション

/debug

デバッグ情報を生成します。

```
/debug[+ | -]
```

引数

+| -

/debug+ または /debug を指定すると、コンパイラによってデバッグ情報が生成され、出力 .pdb ファイルにその情報が出力されます。/debug を指定しないと、既定で /debug- が有効になります。この場合、デバッグ情報は生成されず、デバッグ情報を含む出力ファイルも作成されません。

解説

アプリケーションのデバッグ パフォーマンスを構成する方法については、「イメージのデバッグの簡略化」を参照してください。

使用例

app.exe のデバッグ情報を app.pdb ファイルに出力する例を次に示します。

```
jsc /debug /out:app.pdb test.js
```

参照

その他の技術情報

JScript コンパイラ オプション

/define

プリプロセッサ シンボルを定義します。

```
/define:name1[=value1][,name2[=value1]]
```

引数

name1, *name2*

定義する 1 つ以上の記号の名前。

value1, *value2*

記号の値。ブール値または数値を指定できます。

解説

/define オプションは、名前をプログラム内で記号として定義します。

記号の名前をコンマで区切ると、/define を使用して複数の記号を定義できます。次に例を示します。

```
/define:DEBUG,trace=true,max_Num=100
```

詳細については、「条件付きコンパイル」を参照してください。

/d は /define の省略形です。

使用例

/define:xx を指定してコンパイルする例を次に示します。

```
print("testing")
/*@cc_on @*/
/*@if (@xx)
print("xx defined")
@else @*/
print("xx not defined")
/*@end @*/
```

参照

その他の技術情報

JScript コンパイラ オプション

/fast

高速なプログラムの実行を有効にします。

```
/fast[+ | -]
```

引数
+| -

/fast の既定値はオンです。/fast または /fast+ を指定すると、コンパイラは、速度について最適化された出力ファイルを生成します。ただし、このオプションを使用すると、以前のバージョンの言語機能のいくつかがサポートされません。一方、/fast- を指定すると、言語に関して下位互換性が得られますが、コンパイラが出力するファイルは速度について最適化されていません。

解説

/fast を有効にすると、次のようになります。

- すべての変数を宣言する必要があります。
- 関数は定数になり、関数への代入や関数の再定義はできなくなります。
- 組み込みオブジェクトの定義済みプロパティは DontEnum、DontDelete、ReadOnly になります。
- Global オブジェクト (グローバル スコープでもある) を除いて、組み込みオブジェクトのプロパティを拡張できません。
- 関数呼び出しの中では **arguments** 変数を使用できません。
- 読み取り専用の変数、フィールド、またはメソッドに代入すると、エラーが発生します。

メモ :

/fast- コンパイル モードは、JScript のレガシ コードからスタンドアロン実行可能ファイルを作成する場合に役立ちます。新しい実行可能ファイルまたはライブラリを開発する場合は、/fast+ コンパイル モードを使用してください。これにより、パフォーマンスが向上し、他のアセンブリとの互換性も向上します。

セキュリティに関するメモ :

/fast- コンパイル モードでは、/fast+ モードでは使用できない、以前のバージョンの言語機能を使用できます。これらの機能を誤って使用すると、プログラムのセキュリティが低下することがあります。詳細については、「JScript のセキュリティに関する考慮事項」を参照してください。

使用例

言語に関する完全な下位互換性を犠牲にして、速度が最適化された出力ファイルを作成する例を次に示します。

```
jsc test.js
```

参照
概念
JScript のセキュリティに関する考慮事項
その他の技術情報
JScript コンパイラ オプション

/help、/?

コンパイラのコマンド ライン ヘルプを表示します。

```
/help

-or-
/?
```

解説

このオプションを指定すると、コンパイラは、コンパイラ オプションの一覧と、各オプションの簡単な説明を表示します。

参照

その他の技術情報

JScript コンパイラ オプション

/lcid

コンパイラ メッセージのコード ページを指定します。

```
/lcid:id
```

引数
id

コンパイラからメッセージを出力するときに使用するコード ページの ID。

参照
その他の技術情報
JScript コンパイラ オプション

/lib

アセンブリの参照場所を指定します。

```
/lib:dir1[, dir2]
```

引数

dir1

参照先アセンブリが現在の作業ディレクトリ (コンパイラを起動したディレクトリ) または共通言語ランタイムのシステム ディレクトリにない場合に、コンパイラが探すディレクトリ。

dir2

アセンブリ参照を検索する 1 つ以上の追加ディレクトリ。追加のディレクトリ名はコンマまたはセミコロンで区切ります。

解説

/lib オプションでは、/reference オプションを通じて参照されるアセンブリの場所を指定します。

コンパイラは、完全には修飾されていないアセンブリ参照を次の順序で検索します。

1. 現在の作業ディレクトリ。これは、コンパイラが起動されるディレクトリです。
2. 共通言語ランタイムのシステム ディレクトリ。
3. /lib で指定したディレクトリ。
4. LIB 環境変数で指定したディレクトリ。

アセンブリ参照を指定するには、/reference を使用します。

/lib は追加して指定できます。繰り返して指定すると前の値に追加されます。

使用例

t2.js をコンパイルして .exe を作成する例を次に示します。コンパイラは、作業ディレクトリと C ドライブのルート ディレクトリでアセンブリ参照を探します。

```
jsc /lib:c:\ /reference:t2.dll t2.js
```

参照

その他の技術情報

JScript コンパイラ オプション

/linkresource

マネージ リソースへのリンクを作成します。

```
/linkresource:filename[,name[,public|private]]
-or-
/linkres:filename[,name[,public|private]]
```

引数
filename

アセンブリにリンクするリソース ファイルです。

name[,public|private] (省略可能)

リソースの論理名。リソースを読み込むときに使用します。既定値はファイル名です。オプションとして、ファイルがアセンブリ マニフェスト内でパブリックかプライベートかを指定できます。たとえば、/linkres:filename.res,myname.res,public のように指定します。既定では、*filename* はアセンブリ内でパブリックです。

解説

/linkresource オプションでは、リソース ファイルを出力ファイルに埋め込みません。リソース ファイルを出力ファイルに埋め込むには、/resource オプションを使用します。

filename がリソース ファイル ジェネレータ (Resgen.exe) や開発環境などで作成された .NET Framework リソース ファイルである場合は、System.Resources 名前空間のメンバによってアクセスできます。詳細については、System.Resources.ResourceManager の説明を参照してください。それ以外のすべてのリソースに対しては、System.Reflection.Assembly クラスの GetManifestResource* メソッドを使用して、実行時にリソースにアクセスします。

filename には任意のファイル形式を指定できます。たとえば、ネイティブ DLL をアセンブリの一部として含め、そのネイティブ DLL をグローバル アセンブリ キャッシュにインストールして、アセンブリ内のマネージ コードからアクセスできるようにすることもできます。

/linkres は /linkresource の省略形です。

使用例

`in.js` をコンパイルし、リソース ファイル `rf.resource` にリンクさせる例を次に示します。

```
jsc /linkresource:rf.resource in.js
```

参照
その他の技術情報
JScript コンパイラ オプション

/nologo

開始メッセージが表示されないようにします。

```
/nologo
```

解説

/nologo オプションは、コンパイラの起動時に開始メッセージが表示されないようにします。

参照

その他の技術情報

JScript コンパイラ オプション

/nostdlib

標準ライブラリをインポートしません。

```
/nostdlib[+ | -]
```

引数

+| -

/nostdlib または /nostdlib+ オプションを指定すると、コンパイラは mscorlib.dll をインポートしません。独自の System 名前空間やオブジェクトを定義または作成する場合は、このオプションを使用します。/nostdlib を指定しない場合は、プログラムに mscorlib.dll がインポートされます (/nostdlib- を指定した場合と同じです)。

解説

/nostdlib+ を指定すると、/autoref- も指定されます。

使用例

System.String という名前 (または mscorlib 内のその他の名前) のコンポーネントがある場合、コンポーネントを見つけるための唯一の方法は、次のようなコードを使用して mscorlib の前にライブラリを検索することです。

```
/nostdlib /r:your_library,mscorlib
```

一般に、アプリケーション内で System という名前空間は定義しません。

参照

その他の技術情報

JScript コンパイラ オプション

/out

出力ファイル名を設定します。

```
/out:filename
```

引数
filename

コンパイラで作成される出力ファイルの名前。

解説

/out オプションは、出力ファイルの名前を指定します。/out オプションの後には、1 つ以上のソース コード ファイルを指定できます。

出力ファイルの名前を指定しない場合は、次のように処理されます。

- .exe の名前は、出力ファイルのビルドに使用された最初のソース コード ファイルの名前から付けられます。
- .dll の名前は、出力ファイルのビルドに使用された最初のソース コード ファイルの名前から付けられます。

コマンド ラインでは、コンパイルに対して複数の出力ファイルを指定できます。/out オプションの後に指定したすべてのソース コード ファイルがコンパイルされ、/out オプションに指定した出力ファイルが生成されます。

作成するファイルについて、フルネームと拡張子を指定します。拡張子は、.exe または .dll にする必要があります。/t:exe プロジェクトに対して .dll 拡張子を指定することもできます。

使用例

`t2.js` をコンパイルして出力ファイル `t2.exe` を作成し、`t3.js` をビルドして出力ファイル `t3.exe` を作成する例を次に示します。

```
jsc t2.js /out:t3.exe t3.js
```

参照
その他の技術情報
JScript コンパイラ オプション

/platform (JScript)

アセンブリをどのバージョンの共通言語ランタイム (CLR: Common Language Runtime) で実行するかを指定します。

```
/platform:[string]
```

引数
string

x86、Itanium、x64、または anycpu (既定値)。

- x86 を指定すると、アセンブリは 32 ビットの x86 互換共通言語ランタイムで実行されるようにコンパイルされます。
- Itanium を指定すると、アセンブリは Itanium プロセッサ搭載コンピュータ上の 64 ビット共通言語ランタイムで実行されるようにコンパイルされます。
- x64 を指定すると、アセンブリは x64 または EM64T 命令セットをサポートするコンピュータ上の 64 ビット共通言語ランタイムで実行されるようにコンパイルされます。
- anycpu (既定値) を指定すると、アセンブリは任意のプラットフォームで実行されるようにコンパイルされます。

解説

64 ビットの Windows オペレーティング システムでは次のようになります。

- **/platform:x86** を指定してコンパイルしたアセンブリは、WOW64 環境の 32 ビット CLR 上で実行されます。
- **/platform:anycpu** を指定してコンパイルした実行可能ファイルは、64 ビット CLR 上で実行されます。
- **/platform:anycpu** を指定してコンパイルした DLL は、読み込み先のプロセスと同じ CLR 上で実行されます。

64 ビットの Windows オペレーティング システム上で実行されるアプリケーションの開発の詳細については、「64 ビット アプリケーション」を参照してください。

使用例

次の例は、**/platform** オプションを使用して、Itanium 搭載の 64 ビット Windows オペレーティング システム上の 64 ビット CLR でのみ実行されるアプリケーションをコンパイルする方法を示しています。

```
jsc /platform:Itanium myItanium.js
```

参照
その他の技術情報
JScript コンパイラ オプション

/print

print コマンドを有効にします。

```
/print[+ | -]
```

引数
+| -

既定では、/print または /print+ を指定すると、コンパイラで print ステートメントを使用できるようになります。print ステートメントの例を次に示します。

```
print("hello world");
```

/print- を指定すると、print コマンドが無効になります。

解説

コンソールのない環境に .dll を読み込む場合は、/print- を使用します。

Microsoft.JScript.ScriptStream.Out を TextWriter オブジェクトのインスタンスに設定することにより、print コマンドの出力を別の場所に送信できます。

使用例

コンパイラが print ステートメントを定義しないようにする例を次に示します。

```
jsc /print- test.js
```

参照
その他の技術情報
JScript コンパイラ オプション

/reference

メタデータをインポートします。

```
/reference:file[;file2]
```

引数
file, file2

アセンブリ マニフェストを含む 1 つ以上のファイル。複数のファイルをインポートする場合は、コンマまたはセミコロンでファイル名を区切ります。

解説

/reference オプションは、指定のファイル内のパブリック型情報をコンパイル中のプロジェクトで使用できるようにします。

参照するファイルは、アセンブリである必要があります。たとえば、参照するファイルは、Visual C#、JScript、または Visual Basic の /target:library コンパイラ オプション、または Visual C++ の /clr /LD コンパイラ オプションを使用して作成されている必要があります。

/reference の入力としてモジュールは指定できません。

別のアセンブリ (Assembly B) を参照するアセンブリ (Assembly A) を参照するときに、アセンブリ B も参照する必要があるのは、次の場合です。

- アセンブリ A で使用する型がアセンブリ B の型を継承しているか、アセンブリ B のインターフェイスを実装している場合。
- アセンブリ B の戻り値の型やパラメータの型を持つ、フィールド、プロパティ、イベント、またはメソッドを呼び出す場合。

/lib を使用して、1 つ以上のアセンブリ参照があるディレクトリを指定します。

モジュールではなくアセンブリ内の型をコンパイラで認識するには、型の解決を強制する必要があります。型の解決を実行するには、たとえば、型のインスタンスを定義します。アセンブリの型名を解決する方法は他にもあります。たとえば、アセンブリの型を継承すると、コンパイラで型名が認識されます。

/r は /reference の省略形です。

メモ：

JScript コンパイラである jsc.exe は、コンパイラと同じバージョンまたはそれ以前のバージョンを使用して作成されたアセンブリを参照できます。しかし、JScript コンパイラでは、より新しいバージョンのコンパイラで作成されたアセンブリを参照するときにコンパイル エラーが発生することがあります。たとえば、JScript .NET 2003 コンパイラは JScript .NET 2002 コンパイラで作成されたアセンブリを参照できますが、JScript .NET 2002 コンパイラの場合は、JScript .NET 2003 で作成されたアセンブリを参照するときにエラーが発生する可能性があります。

使用例

ソース ファイル `input.js` をコンパイルし、`metad1.dll` と `metad2.dll` のメタデータをインポートして `out.exe` を生成する例を次に示します。

```
jsc /reference:metad1.dll;metad2.dll /out:out.exe input.js
```

参照
その他の技術情報
JScript コンパイラ オプション

/resource

マネージ リソースをアセンブリに埋め込みます。

```
/resource:filename[,name[,public|private]]
-or-
/res:filename[,name[,public|private]]
```

引数
filename

出力ファイルに埋め込むリソース ファイル。

name[,public|private] (省略可能)

リソースの論理名。リソースを読み込むときに使用します。既定値はファイル名です。オプションとして、ファイルがアセンブリ マニフェスト内でパブリックかプライベートかを指定できます。たとえば、/res:filename.res,myname.res,public のように指定します。既定では、*filename* はアセンブリ内でパブリックです。

解説

/resource オプションを使用すると、リソースがアセンブリにリンクされ、リソース ファイルは出力ファイルに組み込まれません。

filename がリソース ファイル ジェネレータ (Resgen.exe) や開発環境などで作成された .NET Framework リソース ファイルである場合は、System.Resources 名前空間のメンバによってアクセスできます。詳細については、System.Resources.ResourceManager の説明を参照してください。それ以外のすべてのリソースに対しては、System.Reflection.Assembly クラスの GetManifestResource* メソッドを使用して、実行時にリソースにアクセスします。

/res は /resource の省略形です。

使用例

`in.js` をコンパイルし、リソース ファイル `rf.resource` をアタッチする例を次に示します。

```
jsc /res:rf.resource in.js
```

参照
その他の技術情報
JScript コンパイラ オプション

/target

出力ファイル形式を指定します。

/target コンパイラ オプションは、次のいずれかの形式で指定できます。

/target:exe

コンソール アプリケーション (.exe ファイル) を作成します。

/target:library

コード ライブラリ (.dll) を作成します。

/target:winexe

Windows プログラムを作成します。

解説

/target を使用すると、.NET Framework のアセンブリ マニフェストが出力ファイルに組み込まれます。

アセンブリを作成する場合は、CLSCompliantAttribute クラス属性を使用して、コードのすべてまたは一部が CLS 準拠であることを示すことができます。

```
import System;
[assembly:System.CLSCompliant(true)]   // specify assembly compliance

System.CLSCompliant(true) class TestClass   // specify compliance for element
{
   var i: int;
}
```

参照

その他の技術情報

JScript コンパイラ オプション

/target:exe

コンソール アプリケーションを作成します。

```
/target:exe
```

解説

/target:exe オプションは、実行可能な (EXE) コンソール アプリケーションをコンパイラで作成します。/target:exe オプションは、既定で有効になっています。実行可能ファイルは、.exe という拡張子で作成されます。

/out オプションで指定しない限り、出力ファイル名は、各出力ファイルに対するコンパイル内の最初のソース コード ファイルと同じになります。

Windows プログラムの実行形式を作成するには、/target:winexe を使用します。

コマンド ラインで指定すると、/out オプションまたは /target:library オプションを次に指定するまでのすべてのファイルが、.exe の作成に使用されます。/target:exe オプションは、前の /out オプションまたは /target:library オプション以降のすべてのファイルについて有効になります。

/t は /target の省略形です。

使用例

`in.js` をコンパイルし、`in.exe` を作成するコマンド ラインの例を次に示します。

```
jsc /target:exe in.js
jsc in.js
```

参照

関連項目

/target

その他の技術情報

JScript コンパイラ オプション

/target:library

コード ライブラリを作成します。

```
/target:library
```

解説

/target:library オプションは、コンパイラで、実行可能ファイル (EXE) ではなくダイナミック リンク ライブラリ (DLL: Dynamic Link Library) を作成します。作成される DLL の拡張子は .dll です。

/out オプションで特に指定しない限り、出力ファイル名は最初の入力ファイルと同じになります。

コマンド ラインで指定すると、/out オプションまたは /target:exe オプションを次に指定するまでのすべてのソース ファイルが、.dll の作成に使用されます。

/t は /target の省略形です。

メモ :

JScript コンパイラである jsc.exe は、コンパイラと同じバージョンまたはそれ以前のバージョンを使用して作成されたアセンブリを参照できます。しかし、JScript コンパイラでは、より新しいバージョンのコンパイラで作成されたアセンブリを参照するときにコンパイル エラーが発生することがあります。たとえば、JScript .NET 2003 コンパイラは JScript .NET 2002 コンパイラで作成されたアセンブリを参照できますが、JScript .NET 2002 コンパイラの場合は、JScript .NET 2003 で作成されたアセンブリを参照するときにエラーが発生する可能性があります。

使用例

`in.js` をコンパイルし、`in.dll` を作成する例を次に示します。

```
jsc /target:library in.js
```

参照

関連項目

/target

その他の技術情報

JScript コンパイラ オプション

/target:winexe

Windows プログラムを作成します。

```
/target:winexe
```

解説

/target:winexe オプションは、実行可能な (EXE) Windows プログラムをコンパイラで作成します。実行可能ファイルは、.exe という拡張子で作成されます。Windows プログラムは、.NET Framework ライブラリからのユーザー インターフェイスを提供するプログラムです。

コンソール アプリケーションを作成するには、/target:exe を使用します。

/out オプションで指定しない限り、出力ファイル名は、出力ファイルに対するコンパイル内の最初のソース コード ファイルと同じになります。

コマンド ラインで指定すると、/out オプションまたは /target オプションが次に指定されるまで、すべてのファイルが Windows プログラムの作成に使用されます。

/t は /target の省略形です。

使用例

`in.cs` をコンパイルし、Windows プログラムを生成する例を次に示します。

```
jsc /target:winexe in.js
```

参照

関連項目

/target

その他の技術情報

JScript コンパイラ オプション

/utf8output

UTF-8 エンコーディングを使用してコンパイラ出力を表示します。

```
/utf8output[+ | -]
```

引数

+| -

既定では、/utf8output- を指定すると、出力が直接コンソール上に表示されます。/utf8output または /utf8output+ を指定すると、コンパイラの出力がファイルにリダイレクトされます。

解説

国際対応の構成によっては、コンパイル出力がコンソールに正しく表示されないことがあります。このような構成では、**/utf8output** を使用してコンパイラ出力をファイルにリダイレクトします。

このオプションの既定値は /utf8output- です。

/utf8output の指定は、/utf8output+ の指定と同じです。

使用例

`in.js` をコンパイルし、コンパイラの出力を UTF-8 エンコードで表示する例を次に示します。

```
jsc /utf8output in.js
```

参照

その他の技術情報

JScript コンパイラ オプション

/versionsafe

暗黙のオーバーライドを検出します。

```
/versionsafe[+ | -]
```

引数

+| -

既定では /versionsafe- が有効であり、コンパイラは、暗黙のメソッド オーバーライドを検出してもエラーを生成しません。/versionsafe+ (/versionsafe も同じ) を指定すると、コンパイラは暗黙のメソッド オーバーライドに対してエラーを生成します。

解説

メソッドのオーバーライド ステータスを明示的に示すには、hide または override キーワードを使用します。たとえば、/versionsafe を指定して次のコードをコンパイルすると、エラーが発生します。

```
class c
{
function f()
{
}
}
class d extends c
{
function f()
{
}
}
```

使用例

`in.js` をコンパイルし、暗黙のオーバーライドを検出した場合にコンパイルでエラーを生成するようにする例を次に示します。

```
jsc /versionsafe in.js
```

参照

その他の技術情報

JScript コンパイラ オプション

/warn

警告レベルを指定します。

```
/warn:option
```

引数
option

ビルド中に表示させる警告レベルの最小値を指定します。次の 0 ～ 4 の値を指定できます。

警告レベル	説明
0	すべての警告メッセージの出力をオフにします。エラーだけを表示します。
1	エラーおよび重大な警告メッセージを表示します。
2	すべてのエラーおよびレベル 1 の警告に加えて、それより重大度が低いいくつかの警告を表示します。これらの警告には、クラス メンバが非表示になっていることに関するものが含まれます。
3	エラーおよびレベル 1 とレベル 2 の警告に加えて、それより重大度が低いいくつかの警告を表示します。これらの警告には、常に true または false に評価される式に関するものが含まれます。
4	すべてのエラーおよびレベル 1 ～ 3 の警告に加えて、情報を提供するだけの警告を表示します。これは、コマンド ラインにおける既定の警告レベルです。

解説

/warn オプションは、コンパイラで表示する警告レベルを指定します。

指定した警告レベルまでの警告をすべてエラーとして扱うには、/warnaserror を使用します。それより高いレベルの警告は無視されます。

コンパイラは、エラーを常に表示します。

/w は /warn の省略形です。

使用例

`in.js` をコンパイルし、コンパイラでレベル 1 の警告だけを表示する例を次に示します。

```
jsc /warn:1 in.js
```

参照
その他の技術情報
JScript コンパイラ オプション

/warnaserror

警告をエラーとして扱います。

```
/warnaserror[+ | -]
```

引数
+| -

/warnaserror+ オプションは、すべての警告をエラーとして扱います。

解説

通常は警告として通知されるメッセージが、エラーとして通知されるようになります。出力ファイルは作成されません。エラーや警告をできるだけ多く通知するために、ビルドは継続されます。

既定では、/warnaserror- が有効になっていて、警告が通知されても出力ファイルは生成されます。/warnaserror (warnaserror+ も同じ) を指定すると、警告がエラーとして扱われます。

コンパイラで表示する警告のレベルを指定するには、/warn を使用します。

使用例

in.js をコンパイルし、コンパイラで警告を表示しないようにする例を次に示します。

```
jsc /warnaserror in.js
```

参照
その他の技術情報
JScript コンパイラ オプション

/win32res

Win32 リソースを出力ファイルに挿入します。

```
/win32res:filename
```

引数
filename

出力ファイルに加えるリソース ファイル。

解説

Win32 リソース ファイルは、リソース コンパイラを使用して作成できます。

Win32 リソースには、Windows エクスプローラでアプリケーションを識別するときに役立つ、バージョンやビットマップ (アイコン) の情報を含めることができます。/win32res を指定しない場合、コンパイラでは、アセンブリのバージョンに基づくバージョン情報が生成されます。

.NET Framework のリソース ファイルの参照については、「/linkresource」、アタッチについては「/resource」を参照してください。

使用例

`in.js` をコンパイルし、Win32 リソース ファイル `rf.res` をアタッチして `in.exe` を生成する例を次に示します。

```
jsc /win32res:rf.res in.js
```

参照
その他の技術情報
JScript コンパイラ オプション