

【ねらい】

クラス概念と、基本的な使い方について理解します。

1 クラスとは

クラスは C# で最も重要なデータ型と言えます。C# のプログラムは必ず 1 つ以上のクラスが集まって作られています。これまでに紹介してきたサンプル プログラムでも、プロジェクトを作成した時に既定で Program というクラスが自動的に作成され、そのクラスの中にコードを書いていました。この Program クラスとは別にクラスを作ることができます。

クラスは構造体と似ています。構造体のサンプルにあった Book という構造体は、クラスを使用して次のように書くこともできます。

<pre>class Book { public string Title; public string Author; public int Price; }</pre>	Book クラスの定義。 Title フィールドの定義。 Author フィールドの定義。 Price フィールドの定義。
<pre>class Program { static void Main() { Book book1 = new Book(); book1.Title = "C# 入門"; book1.Author = "田中一郎"; book1.Price = 2000; } }</pre>	Program クラスの定義。 Main メソッドの定義。 Book オブジェクトの作成と代入。 book1 の Title フィールドを設定。 book1 の Author フィールドを設定。 book1 の Price フィールドを設定。

上の例では、Program クラスの外で Book クラスを定義していますが、構造体のときのように Program クラスの中で Book クラスを定義して、Program クラスのサブクラスとすることもできます。その場合には構造体を使用したサンプルとのコードの違いは、見かけ上 struct と class というキーワードの違いしかなくなります。

構造体の方は主に、フィールドを持つだけの単純なデータを定義するために使用します。これに対してクラスは、内部にメソッドを持つ、より複雑なオブジェクトを定義するために使用します。(構造体にメソッドを定義することもできますが、メソッドを積極的に利用する場合には通常クラスを

用います。) 構造体は値型でしたが、クラスは列挙型になります。

次は、列車の運行をモデル化したプログラムの例です。

[コード例]

<pre>enum Stations { 東京, 品川, 新横浜, 静岡, 名古屋, 米原, 京都, 新大阪} enum Directions {上り, 下り} class Train { public bool Express; public Directions Direction; public Stations CurrentStation; // 次の駅に移動するメソッド public void MoveNext() { if (Direction == Directions.下り) { if (Express) { CurrentStation = CurrentStation + 2; } else { CurrentStation = CurrentStation + 1; } // 途中省略 } } } class Program { static void Main() { // Train クラスのインスタンスを生成する Train train1 = new Train(); train1.CurrentStation = Stations.東京; train1.Direction = Directions.下り; train1.Express = false; } }</pre>	<p>行 7: 停車駅の列挙型の定義。</p> <p>行 8: 進行方向を表す列挙型の定義。</p> <p>行 9:</p> <p>行 10: rain クラスの定義。</p> <p>行 11:</p> <p>行 12: 列車が急行かどうかを表すフィールド。</p> <p>行 13: 現在の進行方向を表すフィールド。</p> <p>行 14: 現在の停車駅を表すフィールド。</p> <p>行 15:</p> <p>行 16:</p> <p>行 17: 次の駅に移動するメソッドの定義。</p> <p>行 18:</p> <p>行 19: 進行方向が下りかどうか比較する。</p> <p>行 20:</p> <p>行 21: 急行かどうかを確認する。</p> <p>行 22:</p> <p>行 23: 2 つ先の駅に進む。</p> <p>行 24:</p> <p>行 25:</p> <p>行 26:</p> <p>行 27: 1 つ先の駅に進む。</p> <p>行 28:</p> <p>行 57:</p> <p>行 58:</p> <p>行 59:</p> <p>行 60:</p> <p>行 61: Program クラスの定義。</p> <p>行 62:</p> <p>行 63: Main メソッドの定義。</p> <p>行 64:</p> <p>行 65:</p> <p>行 66: Train クラスのオブジェクトの作成。</p> <p>行 67: 現在の停車駅を東京に設定。</p> <p>行 68: 進行方向を下りに設定。</p> <p>行 69: 急行を示すフィールドを false に設定。</p> <p>行 70:</p>
---	---

```

Train train2 = new Train();
train2.CurrentStation = Stations.東京;
train2.Direction = Directions.下り;
train2.Express = true;

// 列車を動かしながら現在の位置を出力する
Console.WriteLine("[列車 1][列車 2]");
Console.WriteLine("{0,3} {1,3}",
train1.CurrentStation, train2.CurrentStation);
for (int i = 1; i <= 5; i++)
{
    train1.MoveNext();
    train2.MoveNext();
    Console.WriteLine("{0,3} {1,3}",
train1.CurrentStation, train2.CurrentStation);
}
}
}

```

行 71: 別の Train クラスのオブジェクトの作成。
 行 72: 現在の停車駅を東京に設定。
 行 73: 進行方向を下りに設定。
 行 74: 急行を示すフィールドを true に設定。
 行 75:
 行 76:
 行 77: 文字列「[列車 1][列車 2]」を出力する。
 行 78: 列車 1 と列車 2 の現在の停車駅を、書式を整えた上で出力する。
 行 79: 以下の処理を 5 回繰り返す。
 行 80:
 行 81: 列車 1 を次の駅に進める。
 行 82: 列車 2 を次の駅に進める。
 行 83: 列車 1 と列車 2 の現在の停車駅を、書式を整えた上で出力する。
 行 84:
 行 87:
 行 88:

〔出力結果〕

```

[列車 1][列車 2]
東京   東京
品川   新横浜
新横浜 名古屋
静岡   京都
名古屋 新大阪
米原   京都

```

サンプル プログラム

サンプル プログラムの「Application1」プロジェクトにこのサンプルがあります。上のコードでは、部分的にソース コードを省略して掲載しています。なお、行番号はコンソール アプリケーションの方のソース ファイルの行番号に合わせてあります。

サンプルの 10 行目で定義されている Train クラスは、プログラムの中で列車を表すために使用されています。現実の列車を模倣するために、Express、Direction、CurrentStation の 3 つのフィールド値を内部に持っています。Express は列車が急行かどうかを表す bool 型のフィールドです。Direction は現在の進行方向を表すためのフィールドで、Directions 列挙型で定義されている「上り」または「下り」の値を持ちます。また、CurrentStation は列車の現在の停車駅を値に持ちます。さらに Train クラスには、列車を次の駅に動かすための MoveNext というメソッドがあり、列車の現在の各フィールド値に基づいて次の駅を計算した結果を CurrentStation に設定し直します。61 行目から始まる Program クラスの Main メソッドの中で Train クラスのオブジェクトを 2 つ、train1 と

train2 という名前で生成しています。イメージ的にはプログラムの中に 2 台の列車があるイメージになります。現在の停車駅と進行方向はそれぞれ同じですが、train1 は各駅停車に、train2 は急行になるように Express フィールドの設定を変えています。最後に 79 行目からの for ループで、MoveNext を 5 回繰返して、列車を 1 つずつ先に進めながら、現在の列車の位置を順に表示しています。同じ Train クラスの 2 つのオブジェクトに対して同じ MoveNext メソッドを呼び出していますが、停車駅は train1 と train2 で異なって表示されます。

クラスではこのようにデータとデータの操作をまとめて定義しておくことができます。このようにデータとデータに関連する操作を 1 か所で定義しておくこと、後で停車駅を変えたり、列車に付随するデータである「列車名」や「乗客の数」などの項目を増やしたくなった時に、Train クラスの中のフィールドやメソッドを修正すれば良いことがわかり、修正がしやすくなるメリットがあります。

2 クラスの定義

クラスの定義は class キーワードを使用して次のように定義します。

```
修飾子 class クラス名
{
    // フィールド、メソッドなどのメンバの定義
}
```

修飾子の例としては public、private などのアクセスレベルを指定するアクセス修飾子があります。Program クラスと同列に、他のクラスの外で宣言されたクラスは、既定で public となり、他のクラスから利用することができます。一方、あるクラスの内部でサブクラスとして定義されたクラスは既定で private となり、そのクラスの外の別のクラスからは利用することはできません。利用できるようにするには public 修飾子を付けてクラスを宣言します。

クラスの中には、いままで出てきたフィールドやメソッドなどのメンバに加えて、まだ説明していない他のメンバも定義することができます。これらのメンバ要素については別途説明します。

3 インスタンスの生成

クラスのオブジェクトを代入する変数の定義の仕方は、他の型の変数の場合と変わりません。下の例では Train 型の変数 train1 を宣言しています。

```
Train train1;
```

配列や構造体と同じように、クラスを使用する前には、new 演算子を使用してオブジェクト (実体) を作成する必要があります⁽¹⁾。クラス型のオブジェクトは他のオブジェクトと区別して**インスタンス**と呼ばれることもあります。次の例では Train クラスのインスタンスを new 演算子を使用して生成し、train1 という名前の Train 型の変数に、そのインスタンスへの参照を代入しています。

```
Train train1 = new Train();
```

後述するコンストラクタがクラスで定義されている場合には、次の例のように初期化する値を伴ってインスタンスを生成することができる場合もあります。

```
Train train1 = new Train(false, Directions.下り, Stations.東京);
```

1 詳細説明

クラスの中にはインスタンスを生成せずにフィールドやメソッドをそのまま使用するタイプのメンバ (静的メンバと言います) を持つクラスもあります。たとえば、Console クラスの WriteLine メソッドは、new Console() のようにして Console クラスのインスタンスを生成したりせずに、Console.WriteLine("abc"); のようにクラス名だけで使用します。これは WriteLine メソッドが静的なメンバとして定義された静的メソッドだからです。静的メンバの詳細については別途説明します。

4 クラスの利用

クラスの中で public キーワードを付けて定義されているメンバは、他のクラスからドット演算子を使用して利用することができます。

```
Stations cs = train1.CurrentStation;  
train1.MoveNext();
```

train1 の CurrentStation フィールドの参照。
train1 の MoveNext メソッドの呼び出し。

クラス (のインスタンス) を引数として受け取るメソッドや、戻り値のデータ型がクラスであるメソッドを作成することもできます。

クラスは参照型であるため、同じクラスの変数同士で代入演算を行った場合には、参照のみがコピーされ、インスタンス自体はコピーされません。同じインスタンスを参照する結果となります。