

# Object Oriented Programming with Visual Basic .NET

## Visual Basic as a true Object Oriented language

- Previous versions (6.0 and earlier) were object-based, e.g. allowed Encapsulation within classes and interface-based programming to take advantage of COM.
- Visual Basic .NET now has inheritance (#1 most requested feature), polymorphism, parameterized constructors, Structured Exception Handling.
- Visual Basic is now a first class citizen and no longer the 'poor relation' as compared to C++, C#, etc. Very powerful yet maintaining the ease of use through its famous syntax.

## Agenda

- Core Tenets of Object-Orientation
- OO features/keywords of Visual Basic .NET
- Abstraction
- Encapsulation
- Inheritance
- Abstract Classes
- Interfaces
- Polymorphism

## Core Tenets of Object Orientation

- **A**bstraction
- **P**olymorphism
- **I**nheritance
- **E**ncapsulation
- This equates to: **A PIE**



## OO keywords in Visual Basic 6.0

- Class
- Interface
- Implements
- Private
- Public
- Friend

## OO keywords in Visual Basic .NET

- Class
- Interface
- Implements
- Private
- Public
- Friend
- Inherits
- MustInherit
- NotInheritable
- Protected
- Overrides
- MustOverride
- Overridable
- NotOverridable

## Abstraction

- Abstraction manages the complexities of a problem by allowing you to identify a set of objects
- Decomposing a programming problem into a set of classes
- More of a software architecture/design phase activity.

## Encapsulation

- Hides the internal implementation of an abstraction within the particular object.
- Prevents usage in undesired ways, and lets you modify such items later without risk of compatibility problems.
- Allows you to contain and control access to a group of associated items.
- Class data should be modified or retrieved only via **Property** procedures or methods.

## Object Lifecycle Management

- Constructors – replace VB 6.0 initialize event. Can accept parameters.
  - E.g. *Public Sub New (ByVal salary As Decimal)*
  - Create new objects with *New* keyword.
- Finalize event replaces VB 6.0 *Terminate()*, but non-deterministic. Implement *IDisposable* interface instead and call *obj.Dispose()* to simulate a *delete*.
- No true deterministic ~destructors as in C++ due to garbage collection mechanism.

## Implementation Inheritance

- Visual Basic .NET provides for true implementation inheritance whereby you can reuse the implementation of a class.
- By inheriting, a class receives the visible methods, properties, and events of the base class.
- We all inherit implicitly from *Object* whenever we make a new class, e.g. the *toString()* method.

## When to use Implementation Inheritance

- Inheritance hierarchy represents 'is-a' relationships and not 'has-a'.
- The base class code is reusable.
- You want to make global changes to derived classes.
- Don't go too deep!

## Class Hierarchy Design: Considerations for Extensibility

- Define classes at each level of an inheritance hierarchy to be as general as possible. Derived classes inherit, and can reuse and extend methods from base classes.
- Be generous in defining data types to avoid difficult changes later on. For example, you might consider using a variable of type **Long** even though your current data may only require an **Integer** variable.
- Only expose items that you are sure derived classes need. **Private** fields and methods reduce naming conflicts and prevent other users from using items that you may need to change down the road.
- Members that are only needed by derived classes should be marked as protected. This reduces the number of classes that are dependent on these members to just derived classes making it easier to update those members during development of the class.

## Polymorphism

- Polymorphism provides for multiple implementations of the same method. For example, different objects can have a **Save()** method, each of which perform different processing.
- Late binding – overridden method calls depend on an object's runtime type.
- Two types:
  - Inheritance Polymorphism
  - Interface Polymorphism

## Abstract Classes

- Recognized by the *MustInherit* keyword.
- Cannot be instantiated – so, essentially can only be used as a base class.
- Different to *Interfaces* because we can have *implementation* in the class.
- Usually partially implemented.
- Useful for encapsulating common functionality.
- Classes may inherit from only one abstract class.
- Useful when creating components because they allow you specify an invariant level of functionality in some methods, but leave the implementation of other methods until a specific implementation of that class is needed.

## Abstract classes vs. Interfaces

- Abstract classes may provide members that have already been implemented. Therefore, you can ensure a certain amount of identical functionality with an abstract class, but cannot with an interface.
- If you anticipate creating multiple versions of your component, create an abstract class. Abstract classes provide a simple and easy way to version your components. By updating the base class, all inheriting classes are automatically updated with the change. Interfaces, on the other hand, cannot be changed once created. If a new version of an interface is required, you must create a whole new interface.
- If the functionality you are creating will be useful across a wide range of disparate objects, use an interface. Abstract classes should be used primarily for objects that are closely related, whereas interfaces are best suited for providing common functionality to unrelated classes.
- If you are designing small, concise bits of functionality, use interfaces. If you are designing large functional units, use an abstract class.
- If you want to provide common, implemented functionality among all implementations of your component, use an abstract class. Abstract classes allow you to partially implement your class, whereas interfaces contain no implementation for any members.

## Interfaces

- Allow you to separate the definition of objects from their implementation.
- Allow systems of software components to evolve without breaking existing code.
- Classes may implement many interfaces

## When to use Interfaces


- Better suited to situations when applications require many possibly unrelated object types to provide certain functionality, e.g. *IDisposable*.
- More flexible than Implementation Inheritance because a single implementation that can implement multiple interfaces.
- Better when you do not need to inherit implementation from a base class.
- Structures cannot inherit from classes, but they can implement interfaces.

## Interface design considerations

- An interface once defined and accepted must remain invariant, to protect applications written to use it. Do not change interfaces once you have published them.
- When an interface needs enhancement, a new interface should be created. This interface might be named by appending a "2" onto the original name, to show its relationship to the existing interface.
- Group a few closely related functions in an interface. Too many functions make the interface unwieldy; dividing the parts of a feature too finely results in extra overhead and diminished ease of use.
- The ability to evolve the system by adding interfaces allows you to gain the advantages object-oriented programming was intended to provide.



demo



Thank you for listening! 😊

**Microsoft®**