

# Introduction to OOP

# Classes and Objects

- Classes describe the type of objects
- Objects are usable instances of classes

C#

```
class SampleClass  
{  
}
```

# Structures

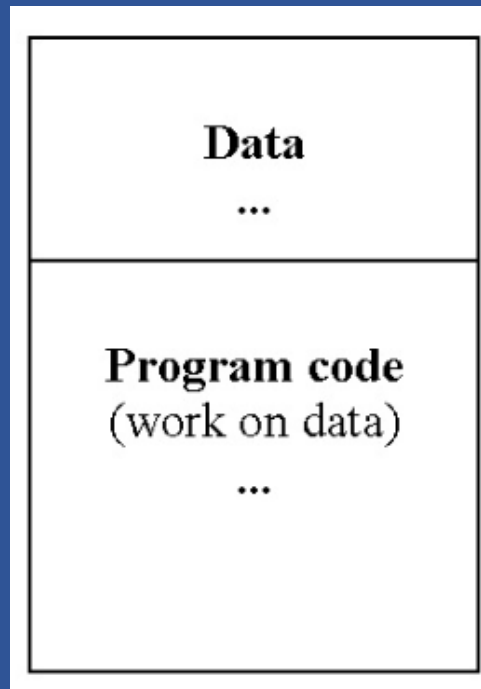
- A light version of classes
- Useful when you need to create large array of objects

C#

```
struct SampleStruct  
{  
}
```

# Class Members

- Each class can have different class members
- Properties = class data
- Methods = class behavior
- Events = communication between different classes



# Fields member

- Information that an object contains
- Fields are like variables because they can be read or set directly

C#

```
class SampleClass
{
    public string sampleField;
}
```

# Properties

- Have get and set procedures
- Standard properties: have a “backing field”

C#

```
class SampleClass
{
    private int _sample;
    public int Sample
    {
        // Return the value stored in a field.
        get { return _sample; }
        // Store the value in the field.
        set { _sample = value; }
    }
}
```

- Auto-implemented properties: create field automatically

C#

```
class SampleClass
{
    public int SampleProperty { get; set; }
}
```

# Methods

A method is an action that an object can perform.

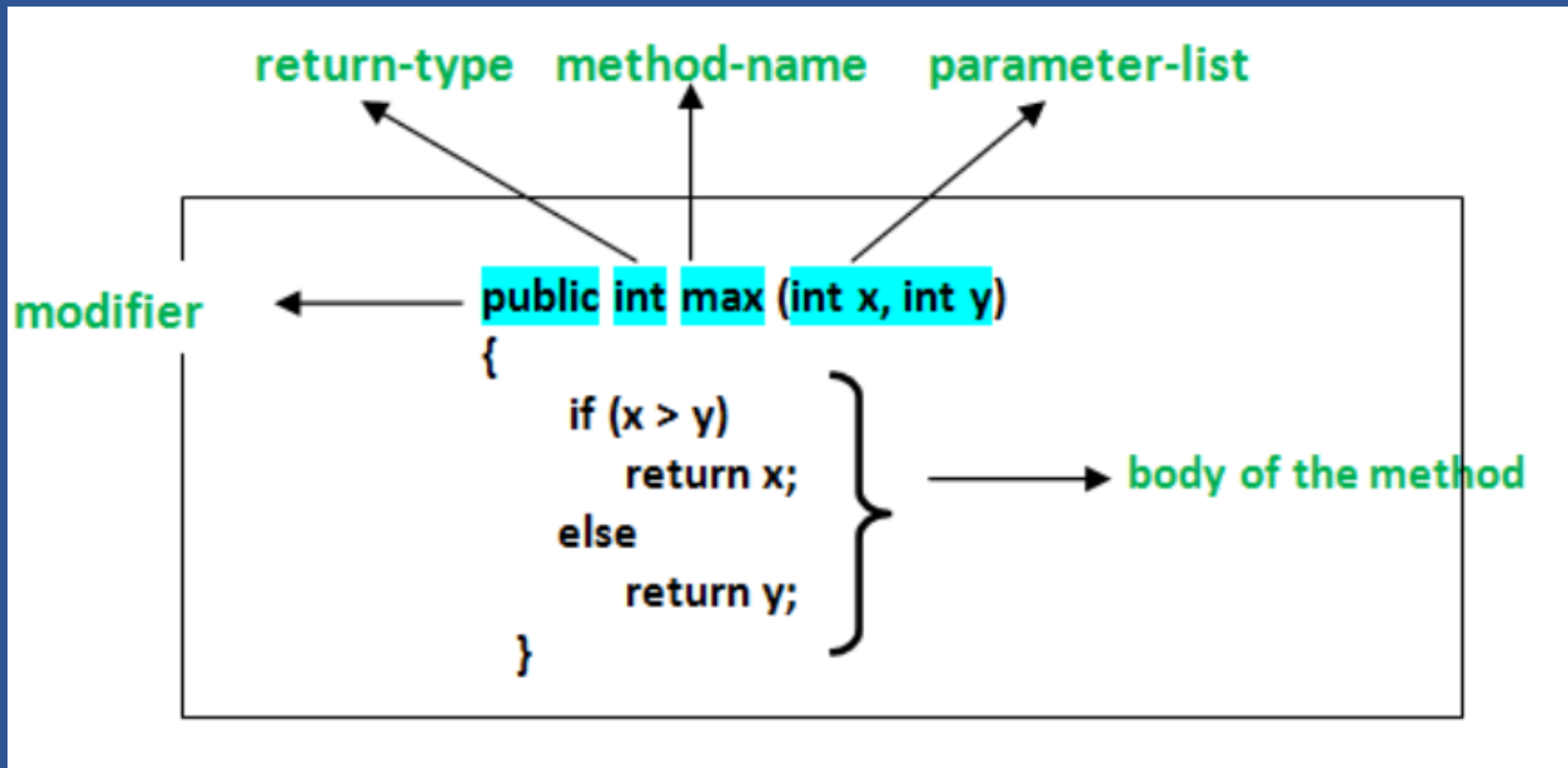
C#

```
class SampleClass
{
    public int sampleMethod(string sampleParam)
    {
        // Insert code here
    }
}
```



# Method Signature

All parts but **Name** and **Return type**



# Method overloading

A class can have several implementations, or overloads, of the same method that differ in the number of parameters or parameter types.

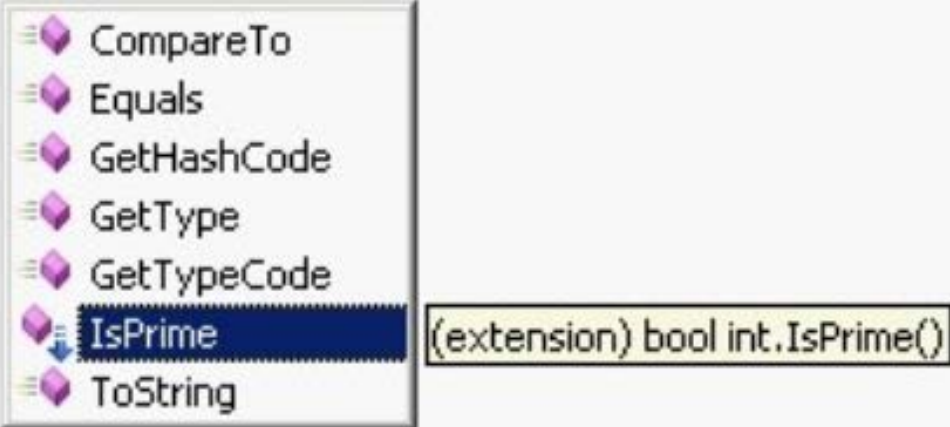
C#

```
public int sampleMethod(string sampleParam) {};  
public int sampleMethod(int sampleParam) {}
```

# Extension Method

add methods to an existing class outside the actual definition of the class.

```
class Program
{
    static void Main(string[] args)
    {
        int anyNumber = 123456;
        anyNumber.
    }
}
```



# Method Constructors

- Methods that are executed automatically when an object of a given type is created.
- Always runs before any other code in a class

```
public class SampleClass
{
    public SampleClass()
    {
        // Add code here
    }
}
```

# Nested Classes

A class defined within another class is called nested. By default, the nested class is private.

```
class Container
{
    class Nested
    {
        // Add code here.
    }
}
```

# Create object of nested class

Use the name of the container class followed by the dot and then followed by the name of the nested class.

```
Container.Nested nestedInstance = new Container.Nested()
```

# Access Modifiers

Modifier	Definition
<code>public</code>	The type or member can be accessed by any other code in the same assembly or another assembly that references it.
<code>private</code>	The type or member can only be accessed by code in the same class.
<code>protected</code>	The type or member can only be accessed by code in the same class or in a derived class.
<code>internal</code>	The type or member can be accessed by any code in the same assembly, but not from another assembly.
<code>protected internal</code>	The type or member can be accessed by any code in the same assembly, or by any derived class in another assembly.
<code>private protected</code>	The type or member can be accessed by code in the same class or in a derived class within the base class assembly.

# Instantiating Classes

To create an object, you need to instantiate a class, or create a class instance.

```
SampleClass sampleObject = new SampleClass();
```



# Using class members

After instantiating a class, you can assign values to the instance's properties and fields and invoke class methods.

```
// Set a property value.  
sampleObject.sampleProperty = "Sample String";  
// Call a method.  
sampleObject.sampleMethod();
```

# Object initializers

To assign values to properties during the class instantiation process, use object initializers.

```
// Set a property value.  
SampleClass sampleObject = new SampleClass  
    { FirstProperty = "A", SecondProperty = "B" };
```

# Static Members

A static member of the class is a property, procedure, or field that is shared by all instances of a class.

```
static class SampleClass
{
    public static string SampleString = "Sample String";
}
```

# Accessing Static Member

To access the static member, use the name of the class without creating an object of this class:

```
Console.WriteLine(SampleClass.SampleString);
```

# Anonymous Types

- Enable you to create objects without writing a class.
- The following example shows an anonymous type that is initialized with two properties named **Amount** and **Message**.

```
var v = new { Amount = 108, Message = "Hello" };

// Rest the mouse pointer over v.Amount and v.Message in the following
// statement to verify that their inferred types are int and string.
Console.WriteLine(v.Amount + v.Message);
```

- Anonymous types typically are used in LINQ queries.

# Inheritance

- Create a new class that reuses, extends, and modifies the behavior that is defined in another class.
- The class whose members are inherited is called the **base class**, and the class that inherits those members is called the **derived class**.

To inherit from a base class:

```
class DerivedClass:BaseClass {}
```

# Sealed Class

By default all classes can be inherited. However, you can specify whether a class must not be used as a base class, or create a class that can be used as a base class only.

To specify that a class cannot be used as a base class:

```
public sealed class A { }
```

# Abstract Class

To specify that a class can be used as a base class only and cannot be instantiated:

```
public abstract class B { }
```



# Overriding Members

If you want to change the behavior of the inherited member, you need to override it. That is, you can define a new implementation of the method, property or event in the derived class.

```
16      class foo
17      {
18          public int i = 1;
19          public virtual void changeI()
20          {
21              i++;
22          }
23      }
24      class bar : foo
25      {
26          override public void changeI()
27          {
28              i--;
29          }
30      }
31  }
```

- By default, a derived class inherits all members from its base class.
- The following modifiers are used to control how properties and methods are overridden:

C# Modifier	Definition
<code>virtual</code>	Allows a class member to be overridden in a derived class.
<code>override</code>	Overrides a virtual (overridable) member defined in the base class.
<code>abstract</code>	Requires that a class member to be overridden in the derived class.
<code>new</code> Modifier	Hides a member inherited from a base class

# Interfaces

Interfaces, like classes, define a set of properties, methods, and events. but do not provide implementation.

Feature	Interface	Abstract class
Multiple inheritance	A class may inherit several interfaces.	A class may inherit only one abstract class.
Default implementation	An interface cannot provide any code, just the signature.	An abstract class can provide complete, default code and/or just the details that have to be overridden.
Access Modifiers	An interface cannot have access modifiers for the subs, functions, properties etc everything is assumed as public	An abstract class can contain access modifiers for the subs, functions, properties

# Generics

Is a concept of type parameters, which make it possible to design classes and methods that defer the specification of one or more types.

To define a generic class:

```
public class SampleGeneric<T>
{
    public T Field;
}
```

To create an instance of a generic class:

```
SampleGeneric<string> sampleObject = new SampleGeneric<string>();
sampleObject.Field = "Sample string";
```

# Delegates

Delegates are used to pass methods as arguments to other methods.

To create a delegate:

```
public delegate void SampleDelegate(string str);
```

To create a reference to a method that matches the signature specified by the delegate:

```
class SampleClass
{
    // Method that matches the SampleDelegate signature.
    public static void sampleMethod(string message)
    {
        // Add code here.
    }
    // Method that instantiates the delegate.
    void SampleDelegate()
    {
        SampleDelegate sd = sampleMethod;
        sd("Sample string");
    }
}
```