Temasek POLYTECHNIC | School of Informatics & IT

Programming Fundamentals using C#

Introduction

- Learn about the Fundamentals of Programming using C# programming language.
- Look at basic programming terminology, familiarize ourselves with programming and briefly review the different stages of software development.

What Does it Mean to Program?

- *To "program" means to write a sequence of instructions, or a set of modules or procedures, that allow for a certain type of computer operation. These sequences of instructions are called "computer programs" or "scripts".*

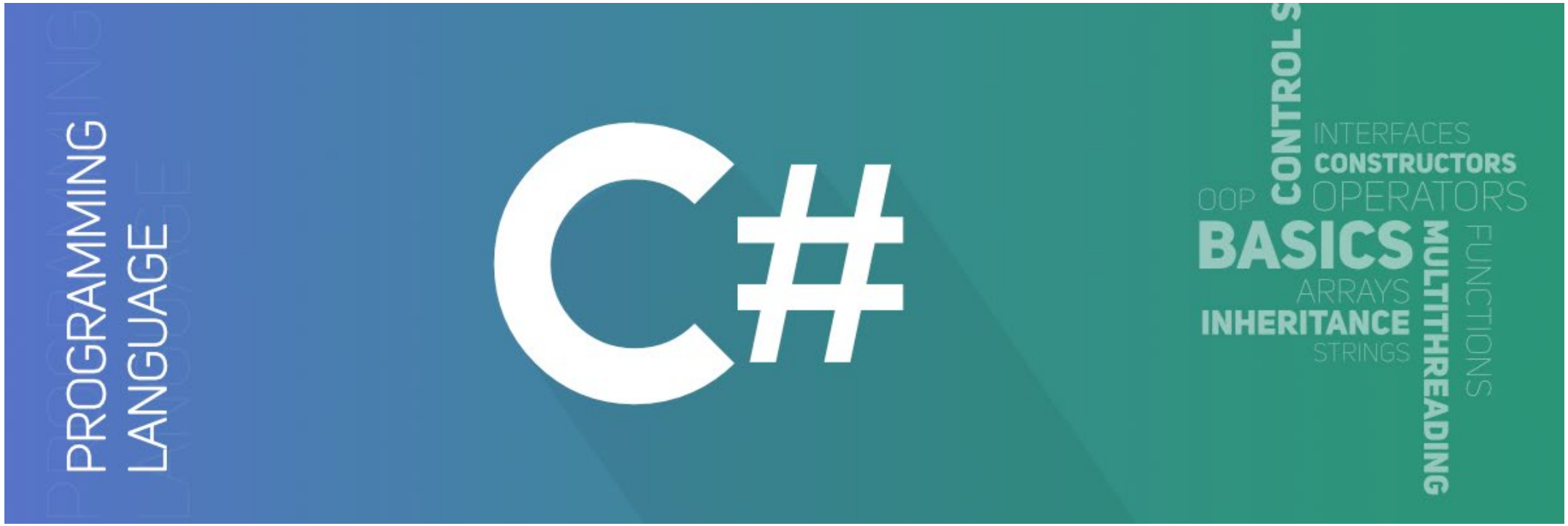
Creating Computer Games

Creating a Word Processor

Creating a Web application

And many more

The C# Language



Introduction to C#

- C# is an object oriented programming language and it supports the concepts of encapsulation, abstraction, polymorphism, etc.
- In C# all the variables, methods and application's entry point are encapsulated within the class definitions.
- C# is developed specifically for .NET Framework and it enable programmers to migrate from C/C++ and Java easily.
- C# is fully Event-driven and visual programming language.
- Microsoft provided an IDE (Integrated Development Environment) tool called Visual Studio to implement C# programs easily.

Classes

- Classes in C# can contain the following elements:
  - **Fields** – member-variables from a certain type;
  - **Properties** – these are a special type of elements, which extend the functionality of the fields by giving the ability of extra data management when extracting and recording it in the class fields.
  - **Methods** – they implement the manipulation of the data.

```
//[access modifier]           -   [class]      -
[identifier]
public class Gun
{
      // Fields, properties, methods and
events go here...
}
```

Access Modifiers

- Access modifiers specify who can use a type or a member
- Access modifiers control encapsulation
- Top-level types (those directly in a namespace) can be public or internal
- Class members can be public, private, protected, internal, or protected internal
- Struct members can be public, private or internal

Access Modifiers

If the access modifier is	Then a member defined in type T and assembly A is accessible
public	to everyone
private	within T only (the default)
protected	to T or types derived from T
internal	to types within A
protected internal	to T or types derived from T or to types within A

Fields

- A field is a member variable
- Holds data for a class or struct
- Can hold:
  - a class instance (a reference),
  - a struct instance (actual data), or
  - an array of class or struct instances (an array is actually a reference)

Readonly Fields

- Similar to a const, but is initialized at run-time in its declaration or in a constructor
  - Once initialized, it cannot be modified
- Differs from a constant
  - Initialized at run-time (vs. compile-time)
    - Don't have to re-compile clients
  - Can be static or per-instance

```
public class MyClass
{
        public static readonly double d1 = Math.Sin(Math.PI);
        public readonly string s1;
        public MyClass(string s) { s1 = s; }
}
```

Constants

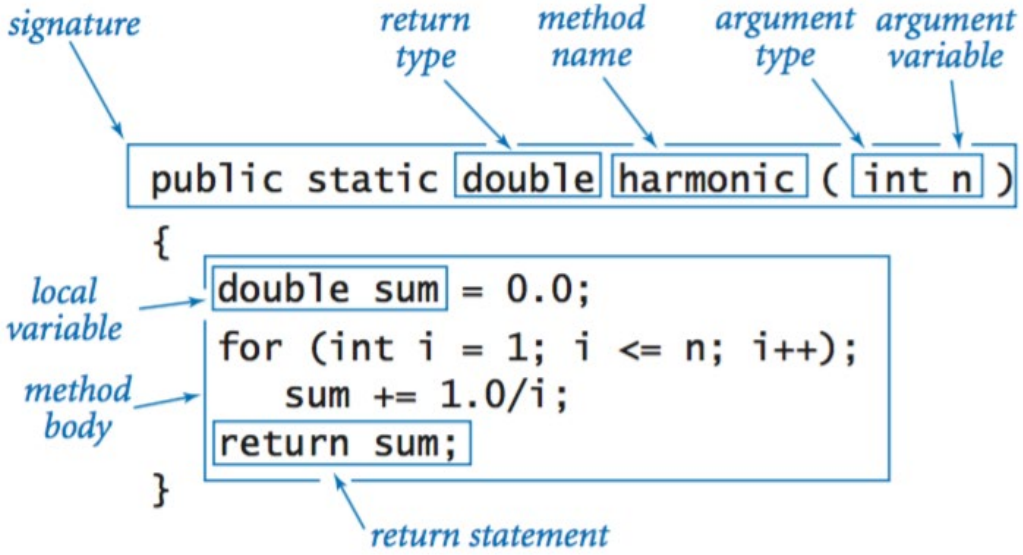
- A constant is a data member that is evaluated at compile-time and is implicitly static (per type)
  - e.g. Math.PI

```
public class MyClass
{
  public const string version = “1.0.0.1”;
  public const string s1 = “abc” + “def”;
  public const int i3 = 1 + 2;
  public const double PI_I3 = i3 * Math.PI;
  public const double s = Math.Sin(Math.PI);  //ERROR
  ...
}
```

Functions / Methods

- A **function** is a way of packaging code that does something and then returns a value.
- In C# functions cannot exist by themselves and must be part of the class.

```
class class_name
    {
        ...
        ...
        <Access_Specifier> <Return_Type> Method_Name(<Parameters>)
        {
            // Statements to Execute
        }
        ...
        ...
    }
```



Non Virtual Methods

- Methods may be virtual or non-virtual (default)
- Non-virtual methods are not polymorphic
  - They cannot be overridden
- Non-virtual methods cannot be abstract

```
class Foo
{
   public void DoSomething(int i)
   {
     ...
   }
}
```

```
Foo f = new Foo();
f.DoSomething();
```

Virtual Methods

- Defined in a base class
- Can be overridden in derived classes
  - Derived classes provide their own specialized implementation
- May contain a default implementation
  - Use abstract method if no default implementation
- A form of polymorphism
- Properties, indexers and events can also be virtual

Virtual Methods

```
class Shape
{
    public virtual void Draw() { ... }
}
class Box : Shape
{
    public override void Draw() { ... }
}
class Sphere : Shape
{
    public override void Draw() { ... }
}
```

```
void HandleShape(Shape s)
{
    s.Draw();
   ...
}
```

```
HandleShape(new Box());
HandleShape(new Sphere());
HandleShape(new Shape());
```

Abstract Methods

- An abstract method is virtual and has no implementation
- Must belong to an abstract class
- Intended to be implemented in a derived class

```
abstract class Shape {
  public abstract void Draw();
}
class Box : Shape {
  public override void Draw() { ... }
}
class Sphere : Shape {
  public override void Draw() { ... }
}
```

```
void HandleShape(Shape s) {
  s.Draw();
  ...
}
```

```
HandleShape(new Box());
HandleShape(new Sphere());
HandleShape(new Shape()); // Error!
```

Method Argument Passing

- By default, data is passed by value
- A copy of the data is created and passed to the method
- For value types, variables cannot be modified by a method call
- For reference types, the instance can be modified by a method call, but the variable itself cannot be modified by a method call

Method Argument Passing

- The ref modifier causes arguments to be passed by reference
- Allows a method call to modify a variable
- Have to use ref modifier in method definition and the code that calls it
- Variable has to have a value before call

```
void RefFunction(ref int p) {
  p++;
}
```

```
int x = 10;
RefFunction(ref x);
// x is now 11
```

Method Argument Passing

- The out modifier causes arguments to be passed out by reference
- Allows a method call to initialize a variable
- Have to use out modifier in method definition and the code that calls it
- Argument has to have a value before returning

```
void OutFunction(out int p) {
  p = 22;
}
```

```
int x;
OutFunction(out x);
// x is now 22
```

Overloaded Methods

- A type may overload methods, i.e. provide multiple methods with the same name
- Each must have a unique signature
- Signature is based upon arguments only, the return value is ignored

One of these is incorrect

```
void Print(int i);
void Print(string s);
void Print(char c);
void Print(float f);
int Print(float f);
```

// Error: duplicate signature

Classes and Objects

```
public class Car : Vehicle
{
   public enum Make { GM, Honda, BMW }
   Make make;
   string vid;
   Point location;

   Car(Make m, string vid; Point loc)
   {
      this.make = m;
      this.vid = vid;
      this.location = loc;
   }
   public void Drive()
   {
      Debug.Log("vroom");
   }
}
```

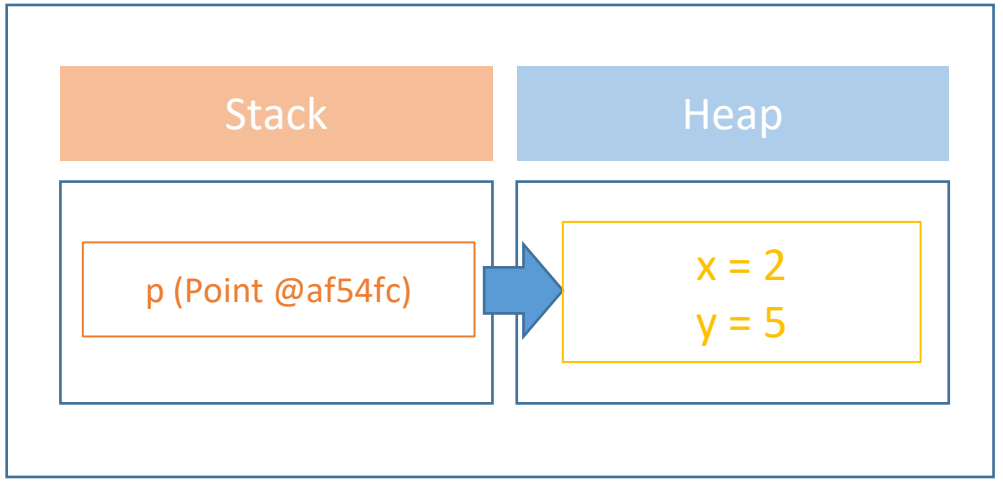
Class

Object

```
Car c =
  new Car(Car.Make.BMW,
        "JF3559QT98",
        new Point(3,7));
c.Drive();
```

Classes and Objects

```
public struct Point
{
   int x, y;
   public Point(int x, int y)
   {
      this.x = x;
      this.y = y;
   }
   public int X
   {
      get { return x; }
      set { x = value; }
   }
   public int Y
   {
      get { return y; }
      set { y = value; }
   }
}
```




```
Point p = new Point(2,5);
p.X += 100;
int px = p.X;     // px = 102
```

Static vs. Instance Members

- By default, members are per instance
  - Each instance gets its own fields
  - Methods apply to a specific instance
- Static members are per type
  - Static methods can't access instance data
  - No this variable in static methods
- Don't abuse static members
  - They are essentially object-oriented
    global data and global functions

Example of Static Method and Variable

```
public class Sequence
{
   // Static field, holding the current sequence value
   private static int currentValue = 0;
   // Intentionally deny instantiation of this class
   private Sequence()
   {
   }
   // Static method for taking the next sequence value
   public static int NextValue()
   {
      currentValue++;
      return currentValue;
   }
}
```

Abstract Classes

- An abstract class is one that cannot be instantiated
- Intended to be used as a base class
- May contain abstract and non-abstract function members
- Similar to an interface
- Cannot be sealed

```
abstract class Shape
{
    public abstract void Draw();
}
```

Sealed Classes

- A sealed class is one that cannot be used as a base class
- Sealed classes can’t be abstract
- All structs are implicitly sealed
- Why seal a class?
  - To prevent unintended derivation
  - Code optimization
    - Virtual function calls can be resolved at compile-time

This Keyword

- The this keyword is a predefined variable available in non-static function members
- Used to access data and function members unambiguously

```
class Person
{
  string name;
  public Person(string name)
  {
     this.name = name;
  }
  public void Introduce(Person p)
  {
     if (p != this)
        Debug.Log("Hi, I'm " + name);
  }
}
```

Base Keyword

- The base keyword is used to access class members that are hidden by similarly named members of the current class

```
class Shape {
  int x, y;
  public override string ToString() {
    return "x=" + x + ",y=" + y;
  }
}
class Circle : Shape {
  int r;
  public override string ToString() {
    return base.ToString() + ",r=" + r;
  }
}
```

Properties

- A property is a virtual field
- Looks like a field, but is implemented with code

```
public class Button: Control
{
  private string caption;
  public string Caption
  {
    get { return caption; }
    set { caption = value;
          Repaint(); }
  }
}
```

Can be
read-only, write-only,
or read/write

```
Button b = new Button();
b.Caption = "OK";
String s = b.Caption;
```

Constructors

- Instance constructors are special methods that are called when a class or struct is instantiated
- Performs custom initialization
- Can be overloaded
- If a class doesn't define any constructors, an implicit parameterless constructor is created
- Cannot create a parameterless constructor for a struct
  - All fields initialized to zero/null

Constructor Initializers

- One constructor can call another with a constructor initializer
- Can call this(…) or base(…)
- Default constructor initializer is base()

```
class B {
  private int h;
  public B() { }
  public B(int h) { this.h = h; }
}
class D : B {
  private int i;
  public D() : this(24) { }
  public D(int i) { this.i = i; }
  public D(int h, int i) : base(h) { this.i = i; }
}
```

Static Constructors

- A static constructor lets you create initialization code that is called once for the class
- Guaranteed to be executed before the first instance of a class or struct is created and before any static member of the class or struct is accessed
- No other guarantees on execution order
- Only one static constructor per type
- Must be parameterless

Static Constructors

```
class SimpleClass
{
    // Static variable that must be initialized at run time.
    static readonly long baseline;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass()
    {
        baseline = DateTime.Now.Ticks;
    }
}
```

Destructors

- A destructor is a method that is called before an instance is garbage collected
- Used to clean up any resources held by the instance, do bookkeeping, etc.
- Only classes, not structs can have destructors

```
class Foo
{
   ~Foo()
   {
      Debug.Log(“Destroyed {0}”, this);
   }
}
```

Destructors

- Unlike C++, C# destructors are non-deterministic
- They are not guaranteed to be called at a specific time
- They are guaranteed to be called before shutdown
- Use the using statement and the IDisposable interface to achieve deterministic finalization

Operator Overloading

- User-defined operators
- Must be a static method

```
class Car
{
  string vid;
  public static bool operator ==(Car x, Car y)
  {
    return x.vid == y.vid;
  }
}
```

Operator Overloading

- Overloadable unary operators

+	-	!	~
true	false	++	--

- Overloadable binary operators

+	-	*	/	!	~
%	&	|	^	==	!=
<<	>>	<	>	<=	>=

Operator Overloading

- No overloading for member access, method invocation, assignment operators, nor these operators: sizeof, new, is, as, typeof, checked, unchecked, &&, ||, and ?:
- The && and || operators are automatically evaluated from & and |
- Overloading a binary operator (e.g. *) implicitly overloads the corresponding assignment operator (e.g. *=)

Operator Overloading

```
public class Vector
{
  int x, y;
  public Vector(x, y) { this.x = x; this.y = y; }
  public static Vector operator +(Vector a, Vector b)
  {
    return Vector(a.x + b.x, a.y + b.y);
  }
  ...
}
```

is Operator

- The is operator is used to dynamically test if the run-time type of an object is compatible with a given type

```
static void DoSomething(object o)
{
    if (o is Car)
        ((Car)o).Drive();
}
```

- Don’t abuse the is operator: it is preferable to design an appropriate type hierarchy with polymorphic methods

as Operator

- The as operator tries to convert a variable to a specified type; if no such conversion is possible the result is null

```
static void DoSomething(object o)
{
    Car c = o as Car;
    if (c != null) c.Drive();
}
```

- More efficient than using is operator: test and convert in one operation
- Same design warning as with the is operator

typeof Operator

- The typeof operator returns the System.Type object for a specified type
- Can then use reflection to dynamically obtain information about the type

```
Debug.Log(typeof(int).FullName);
Debug.Log(typeof(System.Int).Name);
Debug.Log(typeof(float).Module);
Debug.Log(typeof(double).IsPublic);
Debug.Log(typeof(Car).MemberType);
```