C# Language Constructs

Objectives

- On completion of this Session you will be able to :
  - Write a C# program
  - Use CTS types
  - Compare value and reference types
  - Implement Boxing & UnBoxing concept
  - Write functionality using methods
  - Use `ref`, `out` and `params` keywords
  - Declare user defined data types like enum and struct
  - Declare and implement arrays

C# Programming Language

- C++ Heritage
  - Namespaces, pointers (in unsafe code), unsigned types, etc.
- Interoperability
  - C# can talk to COM, DLLs and any of the .NET Framework languages
- Increased productivity
  - Short learning curve
- C# is a type safe Object-oriented Language.
- C# is case sensitive.

Structure of first C# program

```
using System;
. . .
// A "Hello World!" program in C#
class HelloWorld
{
    static void Main()
    {
        Console.WriteLine ("Hello, World");
    }
}
```

- Entry point is `Main()` function.
- The **using** keyword refers to resources in the .NET Framework class library.

Passing Command Line Arguments

```
using System;
. . .
class HelloWorld
{
 static void Main(string[] args)
   {
       Console.Write("parameter count = {0}",
                     args.Length);
       Console.WriteLine("Hello  {0}", args[0]);
       Console.ReadLine();
    }
}
```

Command line arguments

placeholder

Types

- A C# program is a collection of types.
  - Classes, structs, enums, interfaces, delegates
- C# provides a set of predefined types.
  - E.g. `int`, `byte`, `char`, `string`, `object`, …
- Custom types can be created.
- All data and code is defined within a type.
  - No global variables, no global functions

Types

- Types can be instantiated and used by
  - calling methods, get and set properties, etc.
- Can convert from one type to another.
- Types are organized into namespaces, files, assemblies.
- Types are arranged in a hierarchy.
- There are two categories of types:
  - value type
  - reference type

Types

- Value type
  - Directly contain data on stack
- Reference type
  - Contain reference to the actual object on managed heap.

Types

- Value types
  - Primitives `int num; float speed;`
  - Enums `enum State { Off, On }`
  - Structs `struct Point {int x,y;}`
- Reference types
  - Root `Object`
  - String `string`
  - Classes `class Line: Shape`
  - Interfaces `interface IDrawble {...}`
  - Arrays `string[] a = new string[10];`
  - Delegates `delegate void operation();`

Types

- No heap allocation, less GC pressure
- More efficient use of memory
- Less reference indirection
- Unified type system

Type Conversion

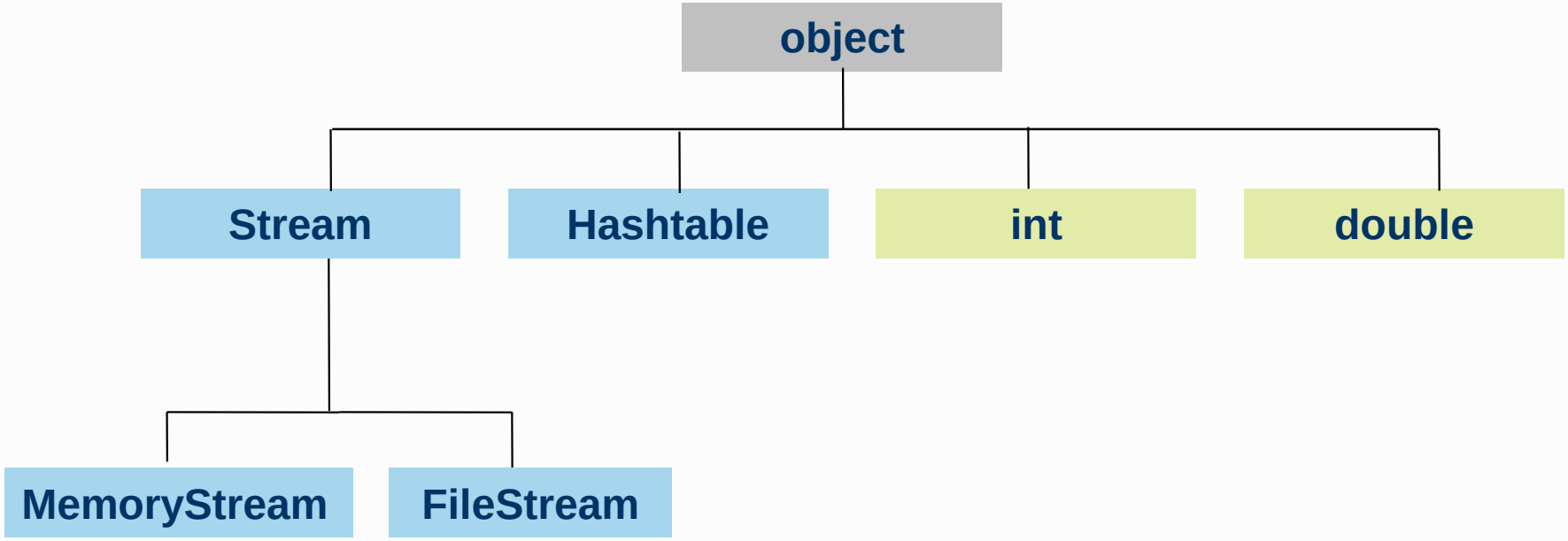
- Implicit conversions
  - No information loss
  - Occur automatically
- Explicit conversions
  - Require a cast
  - May not succeed
  - Information (precision) might be lost

```
int x = 123456;
long y = x;   // implicit
short z = (short)x;// explicit

double d = 1.2345678901234;
float f = (float)d; // explicit
long l = (long)d; // explicit
```

System.Object class

- Everything is an object.
  - All types ultimately inherit from object.



Object class

- Base class for all classes.
- Object methods:
  - public bool Equals(object)
  - protected void Finalize()
  - public int GetHashCode()
  - public System.Type GetType()
  - protected object MemberwiseClone()
  - public void Object()
  - public string ToString()

Polymorphism

- The ability to perform an operation on an object without knowing the precise type of the object

```
void Display(object o) {
Console.WriteLine(o.ToString());
}
```

```
Display(42);
Display(".NET");
Display(4.365434334m);
Display(new Point(56,25));
```

Boxing

- Polymorphic behavior for reference types
  - How does an `int` (value type) get converted into an object (reference type)?

- Solution: Boxing!
  - Only value types get boxed.
  - Reference types do not get boxed.

Unboxing

- Opposite operation of boxing
  - Copies the value out of the box
    - Copies from reference type to value type
- Requires an explicit conversion
    - May not succeed (like all explicit conversions)
    - Essentially a “down cast”

Benefits of boxing

- Enables polymorphism across all types
- Collection classes work with all types
- Eliminates need for wrapper classes
- Replaces OLE Automation's variant

- Many examples in .NET Framework.

```
Hashtable t = new Hashtable();
t.Add(0, "zero");
t.Add(1, "one");
t.Add(2, "two");
```

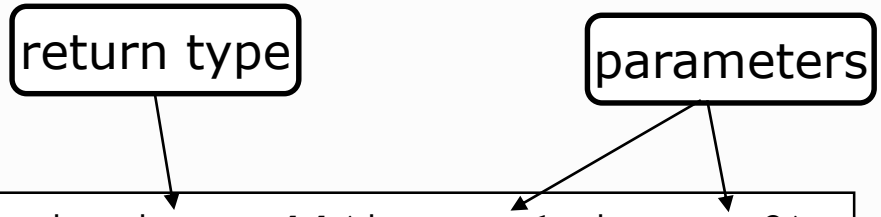
```
string s = string.Format(
  "Your total was {0} on {1}",
  total, date);
```

Disadvantage of boxing

- Disadvantage of boxing
  - Performance cost
- The need for boxing will decrease when the CLR supports generics (similar to C++ templates).

Writing Methods

- Allow modular programming.




```
static int Add(int no1,int no2)
{
   return no1+ no2;
}
```

```
static int ShowMenu()
{
Console.WriteLine("1.Add"+\n+
   "2.Subtract");
int choice =
   Console.ReadLine();
return choice;
}
```

```
public static void Main()
{
//code . . .
int choice = ShowMenu();
//code . . .
int result=Add(3,5);
}
```

Calling methods

`ref` and `out` parameters

```
void static Swap(ref int n1, ref int n2)
{ int temp =  n1; n1 = n2;
    n2 = temp;
}
void static Calculate(float radius,out float area, out float
   circum)
{
   area = 3.14f * radius * radius;
   circum = 2 * 3.14f * radius;
}
```

```
public static  void Main()
{ int x = 10,y =20;
 Swap( ref x, ref y);
 float area,circum;
 Calculate(5, out area, out circum);
}
```

variables need to be initialized to be passed as ref

params

- Defines a method that can accept a variable number of arguments.

```
static void ViewNames(params string[] names)
    {

    Console.WriteLine("Names: {0}, {1},{2}",
                          names[0],names[1],names[2]);


    }
```

```
public static  void Main()
{
   ViewNames("Cheryl", "Joe", "Matt");
}
```

enum

- Is a user defined data type which consists of a set of named integer constants.

```
enum WeekDays {Mon,Tue,Wed,Thu,Fri,Sat}
```

  - Each member starts from 0 by default and is incremented by 1 for each next member.
- Using Enumeration Types

```
WeekDays day = WeekDays.Mon;

Console.WriteLine("{0}", day); //Displays Mon
```

struct

- Is a value type that is typically used to encapsulate small groups of related variables.

```
public struct Point
{ public int x;
 public int y;
}
```

Array

- Declare

```
int[] marks;
```

- Allocate

```
int[] marks = new int[9];
```

- Initialize

```
int[] marks = new int[] {1,2,3,5,7,11,13,17,19};
int[] marks = {1,2,3,5,7,11,13,17,19};
```

- Access and assign

```
marks2[i] = marks[i];
```

- Enumerate

```
foreach (int i in marks) Console.WriteLine(i);
```

Iteration - `foreach`

- `foreach` loop is used to iterate through the items in the collection.
- The elements of collection cannot be modified.

Example

```
int iArr[]= new int[]{1,2,3,4}
foreach(int i in iArr)
{
   Console.WriteLine(i);
}
```

Arrays

- Multidimensional arrays
  - Rectangular

```
int[,] mtrx = new int[2,3];
```

    - Can initialize declaratively.

```
int[,] mtrx =  new int[2,3] {
                    {1,2,3}, {4,5,6} };
```

  - Jagged
    - An array of arrays

```
int[ ][ ] mtrxj = new int[2][];
```

    - Must be initialize procedurally.

Quick Recap…

- C# is type safe Object-oriented programming language.
- `Main()` method is the entry point of a C# program.
- Types are of either value type or reference type.
- Boxing and Unboxing reduces performance of an application.
- `enum` is internally treated as integer constant.
- `struct` is a value type used to define small group of related variables.