

## Q2.1. Compare Java and C#'s class access modifiers

### Top-level class access modifiers

Java	C#	Description
private	private	This modifier is applied to nested class. Meaning: the private class can only accessed (by the members of the) class in which it is defined
public	public	It is a class that is accessible to any other classes
No modifiers (package*)	internal	It is a class that is accessible only to classes in the same Package/Assembly

By default, Java classes are “package”, while C# classes are “internal”

# Class Access Modifiers (cont'd)

- C# class definition

```
[Modifier] class identifier [:base class] {class body}
```

**sealed** the class cannot serve as a base class for another class ← **final**

**abstract** an instance of the class cannot be created ← **abstract**

**static** the class that cannot serve as a base class for another class (i.e. it is sealed) as well as whose instance cannot be created, and can contain only static members ← **static**

**Java:**

## Member access modifiers

Java	C#	Description
private	private	Accessible by member functions of the same class
n/a	protected	Accessible by member functions of the class and its derived classes
No modifiers (package*)	internal	Accessible by member functions in the same package/assembly
protected	protected internal	Accessible by member functions of the class and derived classes, and by classes in the same package/assembly
public	public	Accessible by any functions

If access modifier is not set explicitly, member access is implicitly package access level for Java, and private for C#

Ques 2.2. Re-write the following C# code in Java:

```
class Class1 {  
  
    static void Main(string[] args)    {  
        int a=3; int b= 5;  
        Swap (ref a, ref b);  
        Console.Write ("Results: {0}, {1}", a, b);  
    }  
}
```

```
static public void Swap(ref int A, ref int B) {  
    int temp;  
    temp = A; A = B; B = temp;  
}
```

```
}
```

When parameters are passed by value, swap would not work.

swap(A, B);

```
public static void swap(int x, int y)
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

~~Would not "swap"~~

Primitives are "passed by value" and  
Objects are "~~passed by reference~~" ..

Correct Version:

Possible Answer:

```
public class Test {
```

```
    class MyInteger {  
        int val;  
    }  
}
```



Object references are  
"passed by value"  
swap(A, B);

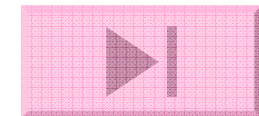
A, B keep pointing to original  
objects they were pointing to ...

```
public void swap (MyInteger X, MyInteger Y) {  
    MyInteger Temp;  
    Temp = X;  
    X = Y;  
    Y = Temp;  
}
```

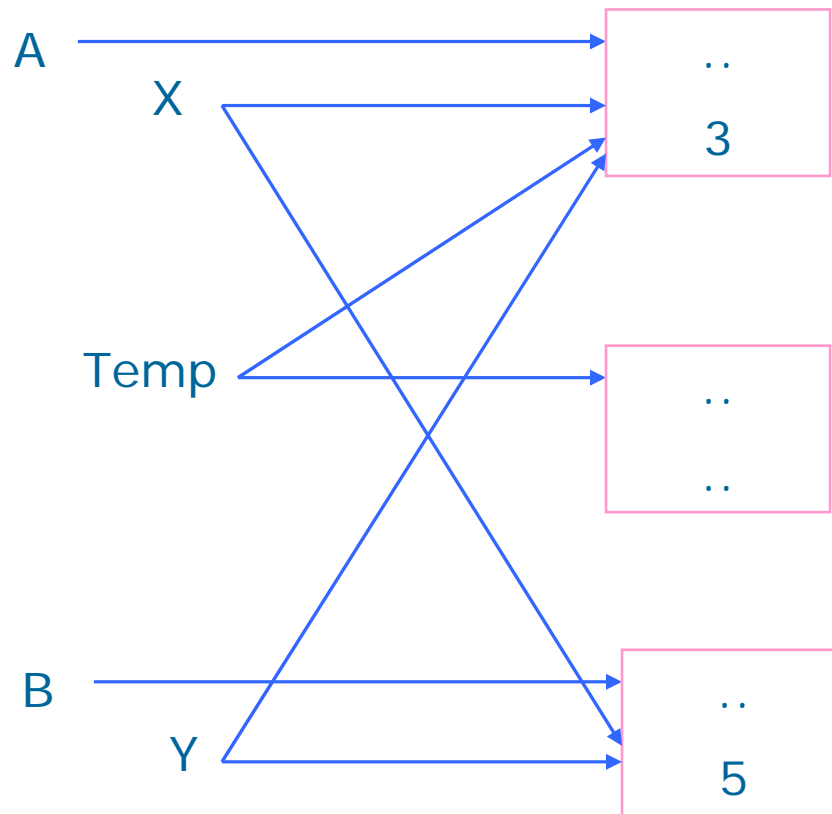
Even this would not "swap"

More details at the website

<http://javadude.com/articles/passbyvalue.htm>



# Working of "swap" with "object references are passed by value"

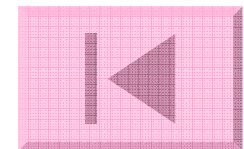


- 
- 

**swap(A, B);**

A, B keep pointing to original objects they were pointing to ...

- 
- 



## One Correct Answer:

```
public class Test {
```

```
    class MyInteger {  
        public int val; ←
```

We can access the Value for the objects of MyInteger

```
    public MyInteger (int i) {  
        val = i;  
    }  
}
```

```
public void swap (MyInteger X, MyInteger Y) {
```

```
    int t = X.val;  
    X.val = Y.val; ←  
    Y.val = t;
```

We can change the value itself !!!

```
}
```

## Other Correct Answer:

```
public class Test {  
  
    class MyInteger {  
        public int val;  
        Public int GetVal () {  
            return val;  
        }  
        Public void SetVal (int i) {  
            val = i;  
        }  
    }  
  
    public void swap (MyInteger X, MyInteger Y) {  
        int t = X.GetVal();  
        X.SetVal(Y.GetVal());  
        Y.SetVal(t);  
    }  
}
```

## Third Correct Answer : “more correct”!

```
public class Test {
```

```
    class MyInteger {
```

```
        private int val;
```

```
        public int GetVal () {  
            return val;  
        }
```

```
        public void SetVal (int i) {  
            val = i;  
        }  
    }
```

```
    public void swap (MyInteger X, MyInteger Y) {
```

```
        int t = X.GetVal();  
        X.SetVal(Y.GetVal());  
        Y.SetVal(t);
```

```
    }
```

Continued from last part ... “Remaining Part” : “part which calls swap”

```
public void test () {  
    MyInteger A = new MyInteger (3);  
    MyInteger B = new MyInteger (5);  
    System.out.println("A is "+A.val);  
    swap (A,B);  
    System.out.println("A is "+A.val);  
}  
  
public static void main(String[] args) {  
    Test t = new Test ();  
    t.test ();  
}  
}
```

- Ques 2.3. 1) Identify the **boxing** and **unboxing** operations.  
2) What is printed in the console upon **execution** of testFun()?  
3) Repeat if Point is a **class**.

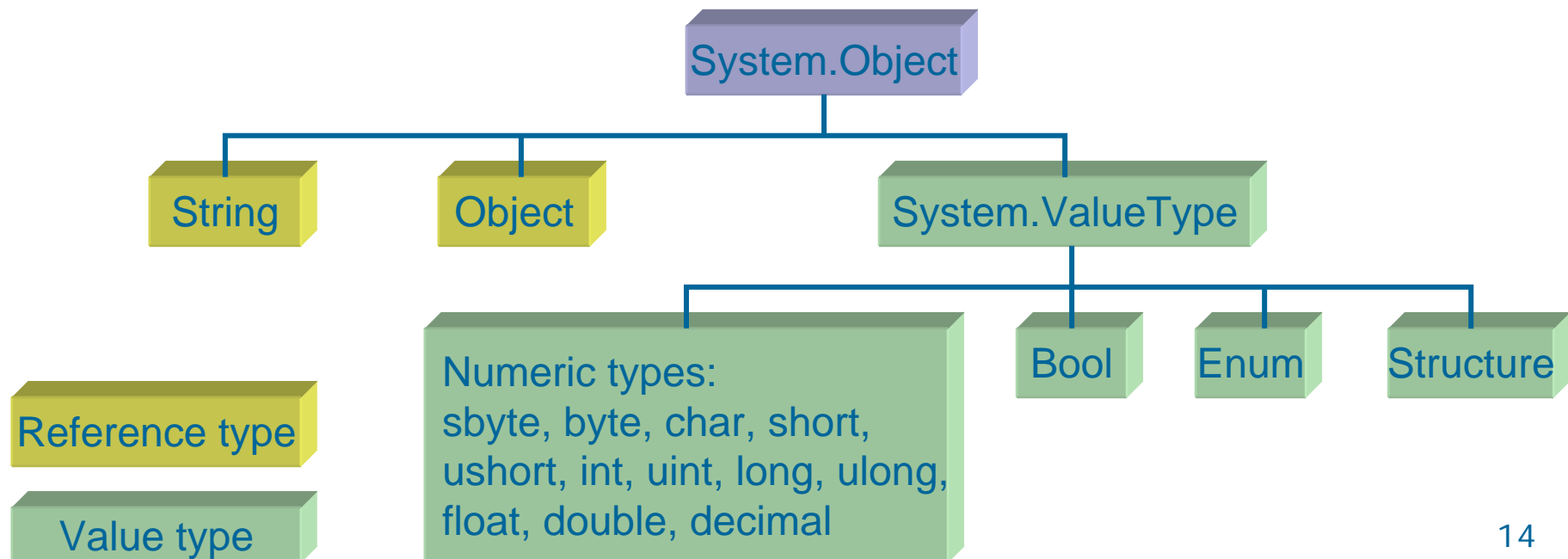


C#

```
struct Point {  
    public int x; public int y;  
    public Point(int x, int y) {  
        this.x = x; this.y = y;  
    }  
}  
  
public static object myFun(object o) { ...}  
  
public void testFun() {  
    int x=1;  
    float y = (float)(myFun(x));  
  
    Point p = new Point(10, 10); //Point is a struct  
    object box = p;  
    p.x = 20;  
    Console.WriteLine(((Point)box).x);  
}
```

# Value Types and Reference Types

- A variable of a *value type* contains data of that type
- A variable of *reference type* contains the address of the location in memory where the data are stored
- Value types are passed to methods *by value* while reference types are effectively passed *by reference*



# Boxing and Unboxing

- **Boxing** and **unboxing** enable value types to be converted to and from the type **object**
- **Boxing** is an implicit conversion of a *value type* to the type *object*
  - Boxing a value of a value type allocates an object instance and copies the value into the new object
- **Unboxing** is an explicit conversion from the type *object* to a *value type*. An unboxing operation consists of:
  - Checking the object instance to make sure it is a boxed value of the given value type
  - Copying the value from the instance into the value-type variable

# 1) Identify the boxing and unboxing operations

```
public static object myFun (object o) { ... }
```

```
public void testFun() {
```

```
    int x=1;
```

```
    float y = (float)(myFun(x));
```

Unboxing

boxing

```
    Point p = new Point(10, 10);
```

boxing

```
    object box = p;
```

```
    p.x = 20;
```

```
    Console.Write(((Point)box).x);
```

Unboxing

## 2) What is printed in the console upon execution of testFun()?

- Original code will generate an **InvalidCastException**
- Amend:

```
public void testFun() {
```

```
    int x=1;
```

```
    int y = (int)(myFun(x));
```

```
    Point p = new Point(10, 10);
```

```
    object box = p;
```

```
    p.x = 20;
```

```
    Console.WriteLine(((Point)box).x);
```

```
}
```

### 3) What is printed in the console upon execution of testFun()?

- value type : allocated on main memory (system stack)
- object type : allocated on heap

- Boxing a value of value type like *int* consists of allocating an object instance and copying the value of the value type (from memory stack) into that object instance (to heap storage).

- This is different from a conversion of a reference-type (resides on heap) to type object (resides on heap), in which the value continues to refer the same instance (since both reside on heap storage)

```
Point p = new Point (10,10);  
object box = p;
```

## Point is a struct (in main memory)

- box points to a copy on the heap

```
p.x = 20;
```

- value in heap is not changed since it's a separate copy of p

```
Console.WriteLine(((Point)box).x);
```

- original value 10 is printed

```
Point p = new Point (10,10);  
object box = p;
```

## Point is a struct (in main memory)

- box points to a copy on the heap

```
p.x = 20;
```

- value in heap is not changed since it's a separate copy of p

```
Console.WriteLine(((Point)box).x);
```

- original value 10 is printed

## Point is a class (in heap)

- value is referenced directly

- amended value 20 is printed

Ques 2.4. Write a C# program which contains a chain of derived classes (shown in C++):

```
class base {
    char *msg;
public:
    base(char *);           // prints message only
    ~base();               // prints an exit message
    void print(void);      // print msg
};
class derived1 : base {
public:
    derived1(char *);      // prints a different message
    ~derived1();           // prints an exit message
};
class derived2 : derived1 {
public:
    derived2(char *);      // prints a 3rd message
    ~derived2();           // prints an exit message
};
```

- Execute the following `main ()` program and explain what happens.

```
void main(void) {
    derived2 x("X"); // in scope of main()

    { // establishing a new scope in scope of main()
        derived2 y("Y");
    }

    derived2 z("Z"); // also in scope of main()
}
```

- Re-write the above `main` C# implementations for
- When were the destructors called?
- When were the constructors called?
- When was space allocated?

```
class base {
    char *msg;
public:
    base(char *); // prints message only
    ~base(); // prints an exit message
    void print(void); // print msg
};
class derived1 : base {
public:
    derived1(char *); // prints a different message
    ~derived1(); // prints an exit message
};
class derived2 : derived1 {
public:
    derived2(char *); // prints a 3rd message
    ~derived2(); // prints an exit message
};
```

```
using System;
```

```
namespace DefaultNamespace {
```

```
public class bse {
```

```
    protected string msg;
```

```
    public bse(string mymsg) {
```

```
        msg = mymsg;
```

```
        Console.WriteLine("Base " + msg + " constructed.");
```

```
    }
```

```
    ~bse() {
```

```
        Console.WriteLine("Base " + msg + " destructed.");
```

```
    }
```

```
}
```

```
class base {
    char *msg;
    public:
        base(char *); // prints message only
        ~base();     // prints an exit message
        void print(void); // print msg
};
```

```

public class derived1 : bse {
    public derived1(string d1msg):base(d1msg) {
        this.msg = d1msg;
        Console.WriteLine("d1 " + this.msg + " constructed.");
    }
    ~derived1() {
        Console.WriteLine("d1 " + this.msg + " destructed.");
    }
}

public class derived2 : derived1 {
    public derived2(string d2msg):base(d2msg) {
        this.msg = d2msg;
        Console.WriteLine("d2 " + this.msg + " constructed.");
    }
    ~derived2() {
        Console.WriteLine("d2 " + this.msg + " destructed.");
    }
}

```

```

class derived1 : base {
    public:
        derived1(char *); // prints a different message
        ~derived1(); // prints an exit message
};
class derived2 : derived1 {
    public:
        derived2(char *); // prints a 3rd message
        ~derived2(); // prints an exit message
};

```

```

class MainClass {
    public static void Main(string[] args) {
        derived2 x = new derived2("X");
        {
            derived2 y = new derived2("Y");
        }
        derived2 z = new derived2("Z");

        Console.WriteLine("=====");
    }
}

```

```

}
}

```

```

void main(void) {
    derived2 x("X"); // in scope of main()
    { // establishing a new scope in scope of main()
        derived2 y("Y");
    }
    derived2 z("Z"); // also in scope of main()
}

```

```
C:\WINDOWS\system32\cmd.exe
Base X constructed.
d1 X constructed.
d2 X constructed. — Space allocated for x
Base Y constructed.
d1 Y constructed.
d2 Y constructed. — Space allocated for y
Base Z constructed.
d1 Z constructed. — Space allocated for z
=====
d2 Z destructed.
d1 Z destructed.
Base Z destructed.
d2 Y destructed.
d1 Y destructed.
Base Y destructed.
d2 X destructed.
d1 X destructed.
Base X destructed.
Press any key to continue . . .
```

Question: When were the destructors for x, y, and z called?

Answer: As per C# documentation, we cannot know. However, we know the earliest instance when they would be called: for x, z: at the end of scope of main() of the program.  
For y: at the end of the "block" within which it is defined.

Question: When were the constructors for x, y, and z called?

Answer: When the classes were instantiated.  
e.g. `derived2 x = new derived2("X");`

Question: When was space allocated for the objects x, y, and z?

Answer: Memory space allocated when the corresponding constructors were called (note: if it is derived class, it needs to chain till "base" class where memory is allocated), as shown in the program output.

Finished Tut 2...

