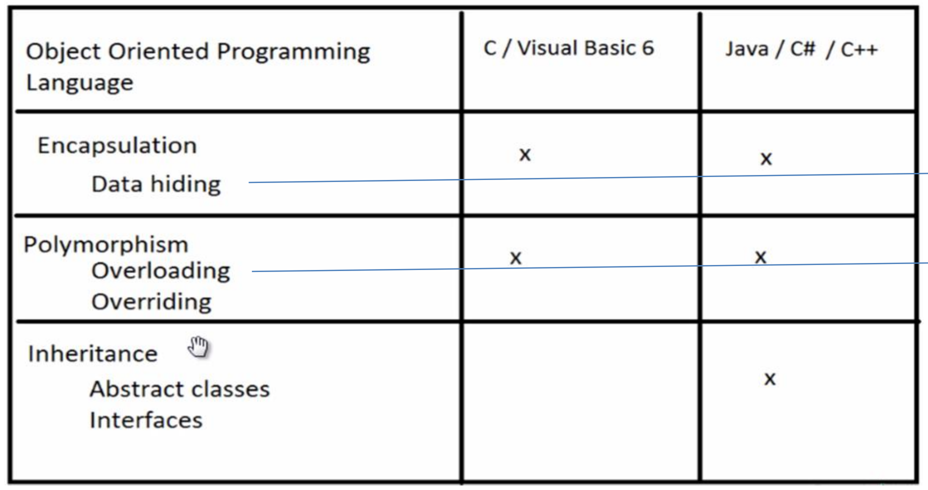
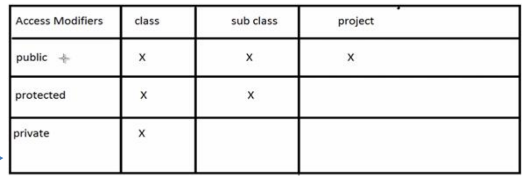


Object Oriented Programming Language	C / Visual Basic 6	Java / C# / C++
Encapsulation Data hiding	x	x
Polymorphism Overloading Overriding	x	x
Inheritance Abstract classes Interfaces		x



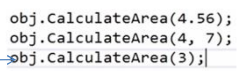
Access Modifiers	class	sub class	project
public	x	x	x
protected	x	x	
private	x		

{ set; get; }

{ private set; get }

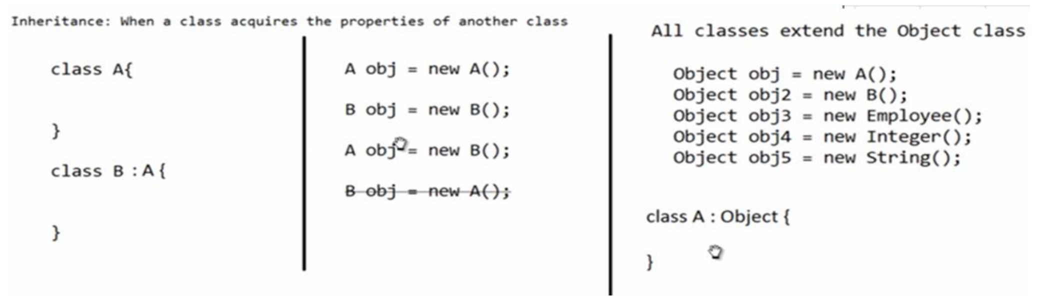
get() set()

Constructor: method with the same name as that of its class. It's executed when an object is created.



```
obj.CalculateArea(4.56);
obj.CalculateArea(4, 7);
obj.CalculateArea(3);
```

depending on the passed parameters, a different method is called



Inheritance: When a class acquires the properties of another class

```
class A{

}

class B :A{

}
```

```
A obj = new A();

B obj = new B();

A obj = new B();

~~B obj = new A();~~
```

All classes extend the Object class

```
Object obj = new A();
Object obj2 = new B();
Object obj3 = new Employee();
Object obj4 = new Integer();
Object obj5 = new String();

class A : Object {

}
```

Overriding

class parent	class child child : parent
method1 {bla}	
public virtual void method2 {bla}	public override void method2 {other bla}

child class can call parent's **private** data & methods using **'base'** keyword **(child already has access to public and protected)**

static : keyword can be applied to a variable, method or class. It has only one instance per class, **the counter is shared.** In instances the counter resets.

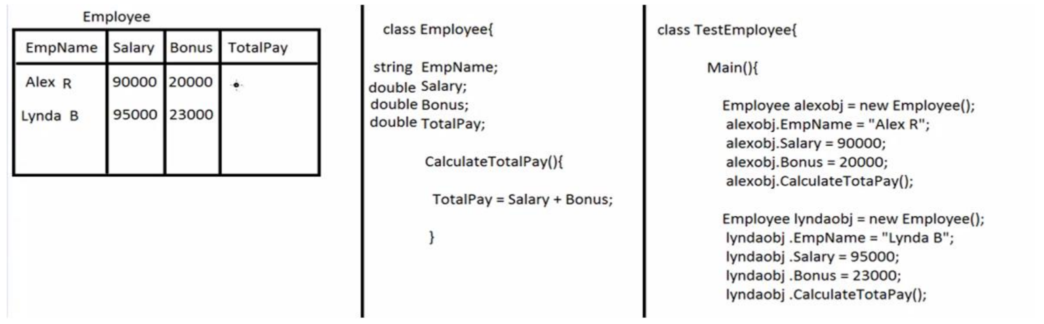
Abstract class

abstract class parent	class child child : parent
method1 {bla}	
public **abstract** void method2 {**empty**}	public **override** void method2 {other bla}

Interface

interface parent	class child child : parent
void method1 {**empty**}	public void method1 {other bla}
void method2 {**empty**}	public void method2 {other bla}
void method3 {**empty**}	public void method3 {other bla}

You can think of classes as tables
attributes as parameters of the class
any value they take in Main as parameter values

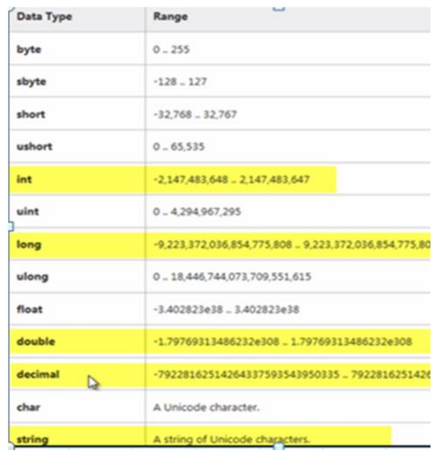


Employee

EmpName	Salary	Bonus	TotalPay
Alex R	90000	20000	
Lynda B	95000	23000	

```
class Employee{

 string  EmpName;
double Salary;
 double Bonus;
 double TotalPay;

        CalculateTotalPay(){

         TotalPay = Salary + Bonus;

        }
```

```
class TestEmployee{

    Main(){

        Employee alexobj = new Employee();
         alexobj.EmpName = "Alex R";
         alexobj.Salary = 90000;
         alexobj.Bonus = 20000;
         alexobj.CalculateTotaPay();

        Employee lyndaobj = new Employee();
         lyndaobj .EmpName = "Lynda B";
         lyndaobj .Salary = 95000;
         lyndaobj .Bonus = 23000;
         lyndaobj .CalculateTotaPay();
```



Data Type	Range
byte	0 .. 255
sbyte	-128 .. 127
short	-32,768 .. 32,767
ushort	0 .. 65,535
int	-2,147,483,648 .. 2,147,483,647
uint	0 .. 4,294,967,295
long	-9,223,372,036,854,775,808 .. 9,223,372,036,854,775,80
ulong	0 .. 18,446,744,073,709,551,615
float	-3.402823e38 .. 3.402823e38
double	-1.79769313486232e308 .. 1.79769313486232e308
decimal	-79228162514264337593543950335 .. 792281625142
char	A Unicode character.
string	A string of Unicode characters.

COLLECTIONS
array alternative lists**, no fixed size**

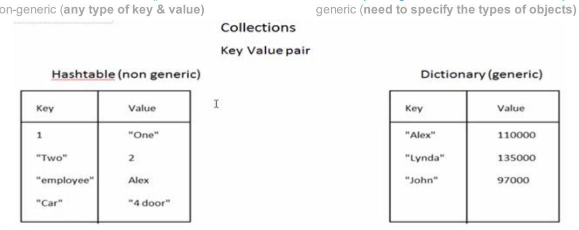
ArrayList aLst = new ArrayList();
non-generic (**any type of object)**

List<Employee> Lst = new List<Employee>();
generic (**need to specify the types of objects)**

for storing **key-value pairs**

Hashtable ht = new Hashtable();
non-generic (**any type of key & value)**

Dictionary<string, double> dct = new Dictionary<string, double>();
generic (**need to specify the types of objects)**



Collections

Key Value pair

Hashtable (non generic)

Key	Value
1	"One"
"Two"	2
"employee"	Alex
"Car"	"4 door"

Dictionary (generic)

Key	Value
"Alex"	110000
"Lynda"	135000
"John"	97000

GENERICS

regular (primitive datatype)

public void IsEqual (int var1, int var)
// hard coded datatype have a disadvantage,
// locked into predefined datatype

object

public void IsEqual (Object var1, Object var)
// you could use Object
// but what if you need SAME type of data?

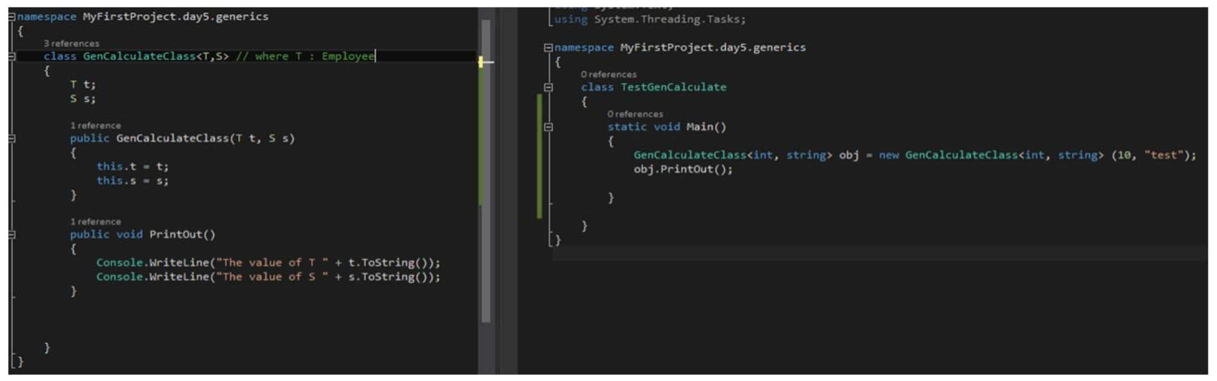
generic

public T IsEqual<T> (T var1, T var2)
// with generic you can ensure the same type of data
// you can use T (type) or any letter

public S IsEqual3<T, S>(T var1, T var2, S var3)
// takes T or S, returns S

public S IsEqualRestrict<T, S>(T var1, T var2, S var3) where T : **Employee**
// can restrict the datatype

generics can apply to classes just like dictionary Dictionary<string, double> dct = new Dictionary<string, double>();

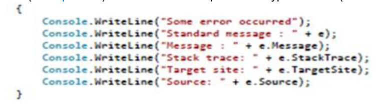


```
namespace MyFirstProject.day5.generics
{
    3 references
    class GenCalculateClass<T,S> // where T : Employee
    {
        T t;
        S s;

        1 reference
        public GenCalculateClass(T t, S s)
        {
            this.t = t;
            this.s = s;
        }

        1 reference
        public void PrintOut()
        {
            Console.WriteLine("The value of T " + t.ToString());
            Console.WriteLine("The value of S " + s.ToString());
        }


    }
}
```

```
using System.Threading.Tasks;

namespace MyFirstProject.day5.generics
{
    0 references
    class TestGenCalculate
    {
        0 references
        static void Main()
        {
            GenCalculateClass<int, string> obj = new GenCalculateClass<int, string> (10, "test");
            obj.PrintOut();

        }

    }
}
```

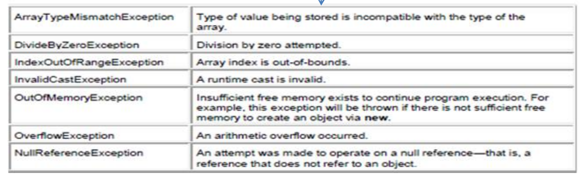
EXCEPTION HANDLING

try { potential exception in **code** }
catch (Exception e)   or specialized type: catch (DivideByZeroException e)



```
{
    Console.WriteLine("Some error occurred");
    Console.WriteLine("Standard message : " + e);
    Console.WriteLine("Message : " + e.Message);
    Console.WriteLine("Stack trace: " + e.StackTrace);
    Console.WriteLine("Target site: " + e.TargetSite);
    Console.WriteLine("Source: " + e.Source);
}
```

finally { will always run }



ArrayTypeMismatchException	Type of value being stored is incompatible with the type of the array.
DivideByZeroException	Division by zero attempted.
IndexOutOfRangeException	Array index is out-of-bounds.
InvalidCastException	A runtime cast is invalid.
OutOfMemoryException	Insufficient free memory exists to continue program execution. For example, this exception will be thrown if there is not sufficient free memory to create an object via **new**.
OverflowException	An arithmetic overflow occurred.
NullReferenceException	An attempt was made to operate on a null reference—that is, a reference that does not refer to an object.

throwing exception to another class

ClassB

try { potential exception in **ClassA method** }
catch (Exception e)

```
{
    Console.WriteLine("Message : " + e.Message);
}
```

finally { will always run }

ClassA
public int ClassA ( )
{
    try { potential exception in code }
    catch

```
{
    throw new Exception("Please check your nums");
}
return c;
```

}

VAR

var implicitly typed variable

var x = "Hi" // with var the variable starts acting as the datatype it is set to

ENUM

string s // you can assign any text to it
int a // you can assign any number to it

What if you need a **custom datatype with a fixed set of data**

ex: public enum Days { Sat, Sun, Mon, Tue, Wed, Thu, Fri };

```
Days day = Days.Tue;
Console.WriteLine(day + " " + (int)day);
```

day can only have one of the above values
(int) gives the order in the above values (i.e. 4 for Tue)

3 ALTERNATIVE WAYS OF METHODS

DELEGATES

a delegate is an o**bject that refers to a method**, one restriction: **signatures need to match**

```
public delegate int Calculate(int x, int y);

class DelegateClass
{
        public static int Add(int x, int y)

        public int Divide(int x, int y)
}
```

regular way of calling methods

```
DelegateClass.Add(6, 2);

DelegateClass calcNormal = new DelegateClass();
calcNormal.Divide(6, 2);
```

calling methods with delegate

```
Calculate calcDelegate = DelegateClass.Add;
calcDelegate(6, 2);

DelegateClass calcDelegate = new DelegateClass();
calc = calcDelegate.Divide;
calc(6,2)
```

Multicasting used for the delegate object to have a **chain of methods**

```
Calculate calcMulticast = DelegateClass.Add;
calcMulticast += DelegateClass.Subtract;
calcMulticast += calcDelegate.Divide

calcMulticast(6,2)          // a calculation of all in chain
```

ANONYMOUS METHOD // eliminates the middle man methods

```
public delegate int Calculate(int x, int y);

class AnonMeth

        Calculate calc =  delegate (int x, int y)

            {
                Console.WriteLine("Add ");
                return x + y;
            };
            Console.WriteLine(calc(6, 2));

            calc += delegate(int x, int y) {
                Console.WriteLine("Subtract ");
                return x - y; };

            Console.WriteLine(calc(6, 2));
```

// no name associated with it, it's anonymous

// you can also do multicasting with it

LAMBDA EXPRESSION // it's another way of anonymous method

regular way of calling methods

```
int MethodName (int x, int y)
{
        return x*y
}
```

lambda expression

```
(x,y) => x*y;
```

if multiple lines

```
(x, y) =>
{
        Console.WriteLine("if method has multiple lines
                just use curly braces");
        int result x*y;
        return result;
};
```

EXTENSION METHODS

functionality can be added to a class withoud using the inheritance mechanism
useful if a **parent class is sealed** (can't inherit from it) and you don't have the source code and y**ou need to add methods to it**

main class

```
class ExtensionDemo
{
        static void Main()
        {
                double a=10;
                a.divide(2);   // double does not have a divide method

                string str = "Word";
                str.ReverseText();  // string does not have a reverse method
```

extenSion class

```
static class myExtensions
{               // extension method is a static method that must be contained in a static class

                public static double divide (this double a, double b)
                {
                                return a/b;
                }

                public static string ReverseText (this string a, double b)
                {
                        .....
                }
}
```

SEALED, CONSTANT, READONLY

sealed:

Used in front of a class, prevents a class from being inherited.

Can also be used on virtual methods to prevent further overrides.

for class and methods
useful in case a child class has another child class and you do not want further overriding

const: public const double pi = 3.14;

- Can't be static.
- Value is evaluated at compile time.
- Initiailized at declaration only.

for data/attributes/variables

for constant, you **need to initialize the value when declaring it** in the beginning

readonly: public readonly double theNum = 1.618;

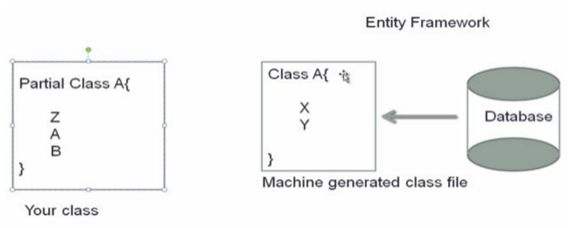
- Can be either instance-level or static.
- Value is evaluated at run time.
- Can be initialized in declaration or by code in the constructor.

for readonly, you **do not need to initialize the value in the beginning**

PARTIAL CLASSES & METHODS

partial: A class, structure, or interface definition can be broken into two or more pieces, with each piece residing in a separate file. When your program is compiled, the pieces are united

partial method: partial method has its declaration in one part and its implementation in another part. The key aspect of a partial method is that the implementation is not required! When the partial method is not implemented by another part of the class or structure, then all calls to the partial method are silently ignored.



REF & OUT

ref : Used to allow a method to return more than one value.

```
obj.SomeMethodA(i);                 i=10
obj.SomeMethodB(ref i);
// instead of i = 10, the memory location of i is passed which could be 100 or something else at this time
```

out: used when you want to receive a value from a method, but not pass in a value.

```
public int GetParts(double n, out double frac)
{
    int whole;
    whole = (int)n;    // get integer part of n
    frac = n - whole; // get fractional part of n, return frac through out double frac
    return whole;
}
```

```
double f;    // for the return value coming out of  out double
i = obj.GetParts(10.125, out f);
```

gives 10 for I and 0.125 for f

VARIABLE NUMBER OF ARGUMENTS

```
public void ShowArgs( string msj, params int[] nums)          // take in a string and after that an undefined number of integers

                // fixed data first, variable data later
```

OPTIONAL AND NAMED ARGUMENTS

```
public void OptArgMethod(int alpha, int beta = 10, int gamma=20)
        // what if you wanted to pass less arguments ?
        // need to initialize the optional ones in the method
        optional arg
        OptArgMethod (1, 2)     // pass 1 and 2 , and the 20 is taken from default

        named arg
        OptArgMethod (alpha:1, gamma:2);          // force in 1 to 1st position, 2 to 3rd position, and 10 is taken for 2nd position
```

STRUCT

```
MyStruct a;  MyStruct a;  MyStruct a;  MyStruct a;  MyStruct a;  MyStruct a;  MyStruct a;   // no need to initialize like:   MyStruct a = new MyStruct();
```

Structures cannot inherit from a class or another struct. They can implement interfaces.

Structures cannot have a no-argument constructor. They can have constructors that take arguments but all instance variables must be assigned.

struct

```
static void Main()
{
    MyStruct a;
    MyStruct b;
    a.x = 10;
    b.x = 20;
    Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);
    // When you assign one structure to another,
    // a copy of the object is made.
    a = b; // a and b refer to different object.
          //if b changes, it doesn't affect a.
    b.x = 30;
    Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);
}
```

class

```
static void Main()
{
    MyClass a = new MyClass();
    MyClass b = new MyClass();
    a.x = 10;
    b.x = 20;
    Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);
    a = b; // now a and b refer to same object.
          //if b changes a changes.
    b.x = 30;
    Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);
}
```

READ & WRITE

write

```
//Pass the filepath and filename to the StreamWriter Constructor
StreamWriter sw = new StreamWriter("C:\\Test.txt");
//Write a line of text
sw.WriteLine("Hello World!!");
//Write a second line of text
sw.WriteLine("From the StreamWriter class");
//Close the file
sw.Close();
```

read

```
//Pass the file path and file name to the StreamReader constructor
StreamReader sr = new StreamReader("C:\\Sample.txt");
//Read the first line of text
line = sr.ReadLine();
//Continue to read until you reach end of file
while (line != null)
{
    //write the lie to console window
    Console.WriteLine(line);
    //Read the next line
    line = sr.ReadLine();
}
//close the file
sr.Close();
```