# Microsoft C#.NET

SunBeam Infotech

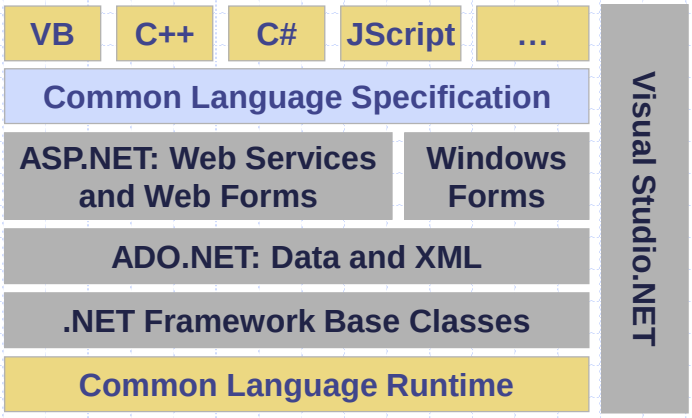


## What Is . NET?

- The combination of:
  - Framework
    - Common language runtime
    - Class libraries
    - ASP.NET
  - Web Services
  - .NET Enterprise Servers
- The means to build the Web the way you want it!

# The .NET Framework

## The .NET Framework and Visual Studio.NET

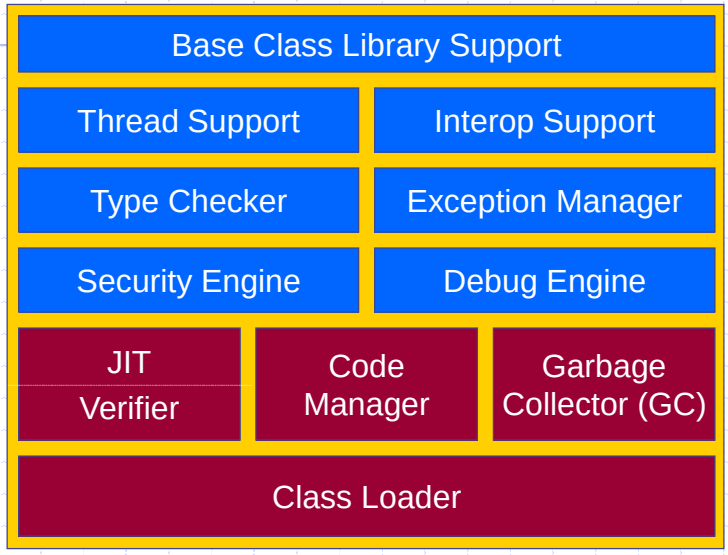





# CLR Highlights

- Common type system
  - Mapping of data types. Programming language ⇨ Framework
- Just-in-time (JIT) compilers
  - JIT compiles intermediary language (MSIL) into native code
  - Highly optimized for platform or device
- Garbage collector
- Permission and policy-based security
- Exceptions
- Threading
- Diagnostics and profiling

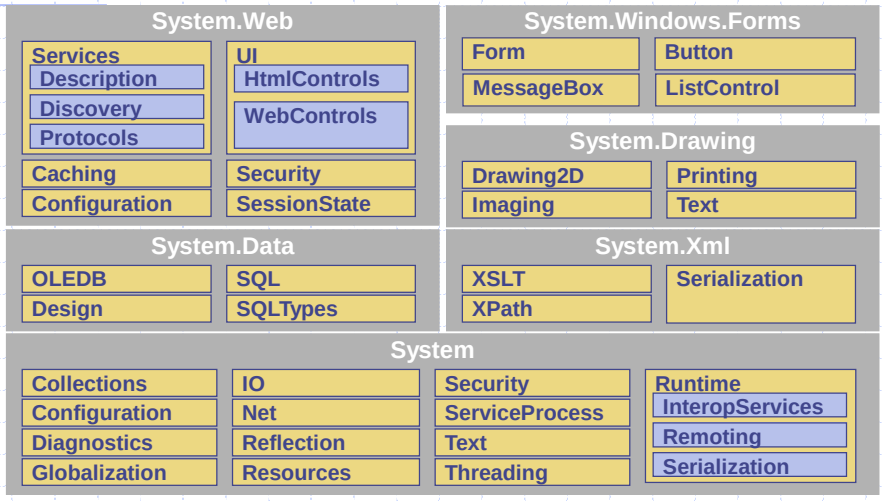
# CLR Diagram





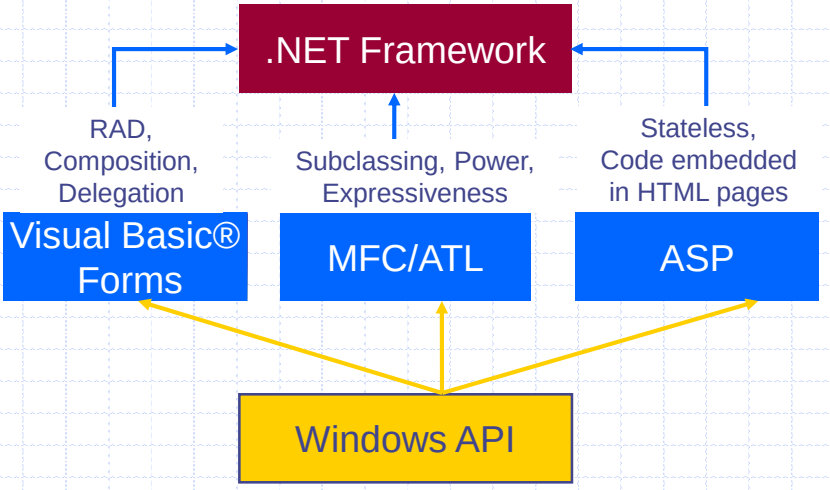

# The .NET Framework

## .NET Framework Classes



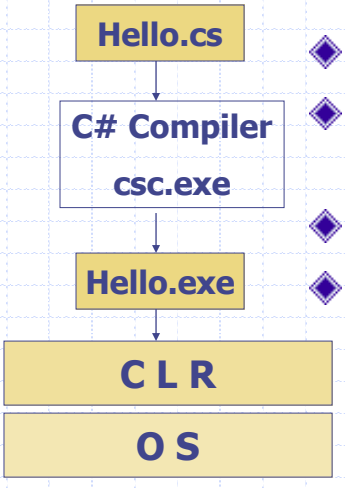
# Unifies Different Programming Models

Consistent API availability regardless of language and programming model






# Compilation of C# Program



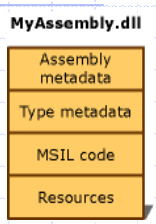

- csc Hello.cs
- Source code written in C# is compiled into an IL.
- That is executed by CLR.
- CLR is OS dependent.

# Managed code

- The CLR also provides other services related to automatic garbage collection, exception handling, and resource management.
- Code that is executed by the CLR is sometimes referred to as "managed code," in contrast to "unmanaged code" which is compiled into native machine language that targets a specific system.



# Assembly




- Source code written in C# is compiled into an intermediate language (IL) that conforms to the CLI specification.
- The IL code, along with resources such as bitmaps and strings, is stored on disk in an executable file called an assembly(.exe/.dll).
- An assembly contains a manifest that provides information on the assembly's types, version, culture, and security requirements.

# Assembly Functions

- It contains IL code that CLR executes. It contains Manifest that makes PE files executables.
- An assembly is the unit at which permissions are requested and granted.
- Every type's identity includes the name of the assembly in which it resides.
- The assembly's manifest contains assembly metadata that is used for resolving types and satisfying resource requests.
- The assembly is the smallest versionable unit in the common language runtime; all types and resources in the same assembly are versioned as a unit.



# Assembly Functions

- It forms a deployment unit. When an application starts, only the assemblies that the application initially calls must be present. Other assemblies, such as localization resources or assemblies containing utility classes, can be retrieved on demand. This allows applications to be kept simple and thin when first downloaded.
- It is the unit at which side-by-side execution is supported
- Thus Assemblies are a fundamental part of programming with the .NET Framework

# Assembly Contents

- static assembly can consist of four elements:
  * Assembly Manifest (Most Important)
  * Type metadata * IL code * Resources
- Assemby can be Single file assembly or multifile assembly.
- Multifile assembly is linked through Manifest and managed as single unit by CLR.
- multifile assembly allow to combine modules written in different languages and to optimize downloading an application by putting seldom used types in a module that is downloaded only when needed.



# Assembly Manifest

- Each assembly's manifest –
  - Enumerates files that make up the assembly.
  - Governs how references to the assembly's types and resources map to the actual files
  - Enumerates other assemblies on which the assembly depends.
  - Provides isolation between assembly clients and the assembly's implementation details.
- Manifest Contains
  - Assembly name, Version number
  - Culture for sattelite assembly
  - List of All files, Exported types, Referenced assemblies

# Assembly Types

- Assemblies can be static or dynamic.
- Static assemblies can include .NET Framework types and resources.
- Static assemblies are stored on disk in portable executable (PE) files.
- You can also use the .NET Framework to create dynamic assemblies, which are run directly from memory and are not saved to disk before execution.
- You can also use common language runtime APIs, such as System.Reflection.Emit, to create dynamic assemblies.



# Execution of Assembly

- When the C# program is executed, the assembly is loaded into the CLR, which might take various actions based on the information in the manifest.
- Then, if the security requirements are met, the CLR performs just in time (JIT) compilation to convert the IL code into native machine instructions.
- These can be passed to OS.

# Global Assembly Cache

- Each computer where the CLR is installed has a machine-wide code cache called GAC.
- It stores assemblies specifically designated to be shared by several applications on the computer.
- You should share assemblies by installing them into the GAC only when you need to.
- There are several ways to deploy an assembly into the global assembly cache:
  - Use an installer designed to work with the GAC.
  - Use a developer tool Gacutil.exe, provided by the .NET Framework SDK.
  - Use Windows Explorer to drag assemblies into the cache.



# Global Assembly Cache

- It is recommended that only users with Administrator privileges be allowed to delete files from the global assembly cache
- When an assembly is added to the global assembly cache, integrity checks are performed on all files that make up assembly.
- The cache performs these integrity checks to ensure that assembly has not been tampered with.
- Assembly registered with GAC is shared assembly while assembly placed in application folder is local or private assembly
- Assemblies deployed in the global assembly cache must have a strong name.

# Strong named assembly

- A strong name consists of the assembly's identity – its simple text name, version number, and culture information – plus a public key and a digital signature.
- It is generated from an assembly file using the corresponding private key.
- Strong names guarantee name uniqueness by relying on unique key pairs.
- When you reference a strong-named assembly, you expect to get certain benefits, such as versioning and naming protection.



# Side-by-side Execution

- Side-by-side execution is the ability to store and execute multiple versions of an component on the same computer.
- This means that you can have multiple versions of the runtime, and multiple versions of applications that use a version of the runtime, on the same computer at the same time.
- Side-by-side execution gives you more control over what versions of a component and runtime an application binds to.
- Support for side-by-side storage and execution of different versions is possible due to strong naming and is built into the infrastructure of the runtime.

## C# Advantages

- C# syntax simplifies many of the complexities of C++ while providing powerful features such as nullable value types, enumerations, delegates, anonymous methods and direct memory access, which are not found in Java.
- The C# build process is simple compared to C and C++ and more flexible than in Java.
- A C# source file may define any number of classes, structs, interfaces, and events.



## Common Language Runtime

- C# programs run on the .NET Framework, an integral component of Windows that includes a virtual execution system called the common language runtime (CLR) and a unified set of class libraries.
- The CLR is Microsoft's commercial implementation of the common language infrastructure (CLI), an international standard that is the basis for creating execution and development environments in which languages and libraries work together seamlessly.

## // First C# Program

```
/*File Name : Hello.cs*/
namespace HelloWorld {
        class Hello {
                static void Main() {
                        System.Console.WriteLine
        ("Hello World!");
                }
        }
}
```



## Main method

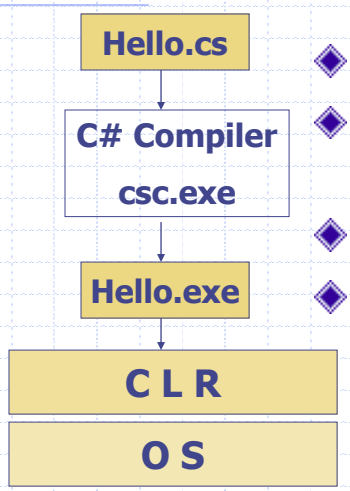
- Possible protptypes for main:
  - static void Main();
  - static int Main();
  - static void Main(string[] args);
  - static int Main(string[] args);
- Command line arguments: agrs[0], ...
- Command line argument count: args.length

# Input / Output in C#

- System.Console.WriteLine("Hello");
  - Namespace – System
  - Class – Console
  - Static Method – WriteLine
- If System namespace is declared already as : using namespace System;
- Console.WriteLine("\"");  is correct.
- string msg=System.Console.ReadLine();
- int num = int.parse(msg);

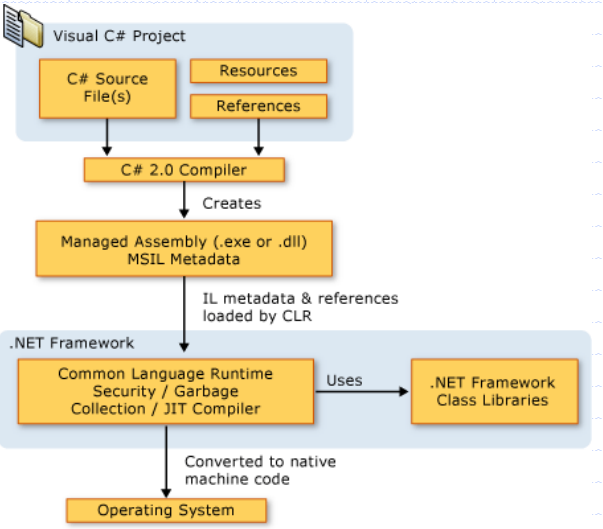


# Compilation of C# Program




- csc Hello.cs
- Source code written in C# is compiled into an IL.
- That is executed by CLR.
- CLR is OS dependent.

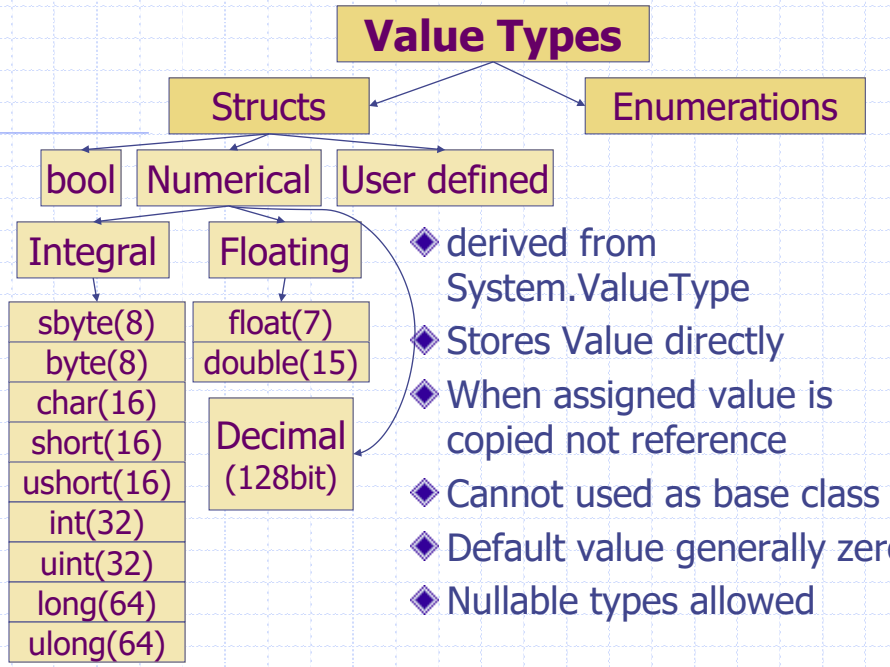
# Execution of C# program






# Data types

- C# is a strongly typed language
- Classified as
  - Built-in type
  - User-defined type
- Or Classified as
  - Value type – stores values.
  - Reference type – stores references to data.




- derived from System.ValueType
- Stores Value directly
- When assigned value is copied not reference
- Cannot used as base class
- Default value generally zero
- Nullable types allowed



## Built in types And Framework Types

| | |
|---|---|
| bool : System.Boolean | uint : System.UInt32 |
| byte : System.Byte | long : System.Int64 |
| sbyte : System.SByte | ulong : System.UInt64 |
| char : System.Char | short : System.Int16 |
| decimal : System.Decimal | ushort : System.UInt16 |
| double : System.Double | |
| float : System.Single | object : System.Object |
| int : System.Int32 | string : System.String |

# Reference Types

| Class |
| --- |
| Interface |
| object |
| string |
| delegate |

- store references to actual data
- Classes and Interfaces are user defined.
- object is cosmic super class
- string is also built in type represents immutable strings
- delegate represent object oriented function pointer



# Boxing and Unboxing

- By Boxing Value types can be treated as objects.
- This allows the value type to be stored on the garbage collected heap as an Object.
- Unboxing extracts the value type from the object.

# Boxing and Unboxing

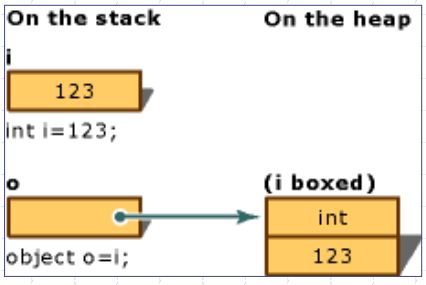
- example, the integer variable i is *boxed* and assigned to object o.
- int i = 123;

  object obj = (object) i; // boxing
- The object o can then be unboxed and assigned to integer variable i:
- object obj = 123;

  i = (int) obj; // unboxing

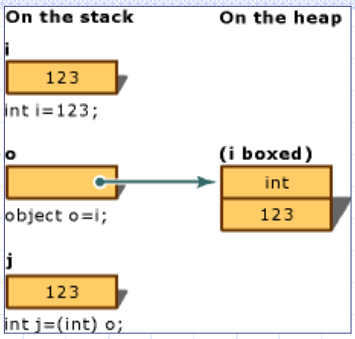


# Boxing




- object o = i; // implicit boxing
- object o=(object) i; //explicit boxing(not required)
- The result of this statement is creating an object reference o, on the stack, that references a value of the type **int**, on heap.
- This value is a copy of the value-type value assigned to the variable i.

# Unboxing




- int i = 123; // a value type
- object o = i; // boxing
- int j = (int) o; // unboxing
- explicit conversion from type object to value type
- 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.
- Attempting to unbox null or a reference to an incompatible value type will result in an InvalidCastException.



# Arrays

- int[] a={1,2,3,4,5};
- int[] a=new int[5];
- int[] a=new int[5]{1,2,3,4,5};
- string[] weekDays = new string[] { "Sun", "Mon", "Tue"};
- int[,] a=new int[4,2];
- int[,] a=new int[,]{{1,2},{3,4},{5,6}};

# Arrays

- int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 }; foreach (int i in numbers) { System.Console.WriteLine(i);
  }
- Console.WriteLine(numbers.Length);



# Arrays

- A jagged array is an array whose elements are arrays.
- The elements of a jagged array can be of different dimensions and sizes.
- A jagged array is sometimes called an "array of arrays."
- int[][] jaggedArray = new int[3][];
- jaggedArray[0] = new int[5];
- jaggedArray[1] = new int[4];
- jaggedArray[2] = new int[2];

# Passing Array to Function

```
static void FillArray(int[] arr) {
        if (arr == null)
                arr = new int[5];
        arr[4] = 5555;
}
static void Main() {
        int[] theArray={1, 2, 3, 4, 5};
        FillArray(theArray);
        for (int i = 0; i < theArray.Length; i++)
                Console.Write(theArray[i] + " ");
        // Result: 1, 2, 3, 4, 5555
}
```

Array is a reference type i.e. the Array objects are passed by ref. The changes in array object are reflected into the calling fn.



# Passing Array to Function

```
static void FillArray(int[] arr) {
        if (arr == null)
                arr = new int[5];
        arr[4] = 5555;
}
static void Main() {
        int[] theArray=null;
        FillArray(theArray);
        if(theArray != null) // condition false – no output
                for (int i = 0; i < theArray.Length; i++)
                        Console.Write(theArray[i] + " ");
}
```

However array reference is passed by Value i.e. reference is copied to fn arg. Now any changes done in the reference by the called fn will not reflect back in the calling fn.

# Passing Array to Function by ref

```
static void FillArray(ref int[] arr) {
        if (arr == null)
                arr = new int[5];
        arr[4] = 5555;
}
static void Main() {
        int[] theArray=null;
        FillArray(ref theArray);
        for (int i = 0; i < theArray.Length; i++)
                Console.Write(theArray[i] + " ");
        // Result: 0, 0, 0, 0, 5555
}
```

Here array reference is passed by ref, so changes done in reference by the called fn will reflect back in the calling fn. The "ref" arg must be initialized by the calling fn.



# Passing Array to Function by out

```
static void FillArray(out int[] arr) {
        if (arr == null)
                arr = new int[5];
        arr[4] = 5555;
}
static void Main() {
        int[] theArray;
        FillArray(out theArray);
        for (int i = 0; i < theArray.Length; i++)
                Console.Write(theArray[i] + " ");
        // Result: 0, 0, 0, 0, 5555
}
```

Here array reference is passed by out, so changes done in reference by the called fn will reflect back in the calling fn. The "out" arg need not to be initialized by the calling fn. It must be initialized by the called fn.

## Passing array to function

- In example 1, object contents modified by fn will be seen in calling fn – objects are passed by reference.
- In example 2, reference modified in fn is not reflected back as reference is passed by value (copied)
- In example 3, a ref parameter of an array type must be definitely assigned by the calling fn.
- Therefore, there is no need to be definitely assigned by the called fn. A ref parameter of an array type may be altered as a result of the call (i.e. it will hold new address).
- In example 4, an out parameter of an array type must be assigned before it is used; that is, it must be assigned by the called fn. Not mandatory to be initialized by the calling fn.



## string type [ mscorlib.dll ]

- sequence of Unicode characters
- alias for reference type System.String
- == and != operators check for values even though it is reference type
- +, [], += operators can be used
- To avoid escape seq: str=@"c:\file.txt";
- To modify the same string object use System.Text.StringBuilder class
- string[] split(char[] seperators);

# Operator Precedance Chart

| | |
|---|---|
| Primary | x.y, f(x), a[x], x++, x--, new, typeof, checked, unchecked |
| Unary | +, -, !, ~, ++x, --x, (T)x |
| Arithmetic — Multiplicative | *, /, % |
| Arithmetic — Additive | +, - |
| Shift | <<, >> |
| Relational and type testing | <, >, <=, >=, is, as |
| Equality | ==, != |
| Logical, in order of precedence | &, ^, \| |
| Conditional, in order of precedence | &&, \|\|, ?: |
| Assignment | =, +=, -=, *=, /=, %=, &=, \|=, ^=, <<=, >>= |



# checked & unchecked

- C# statements can execute in either checked or unchecked context.
- In a checked context, arithmetic overflow raises an exception.
- In unchecked context, arithmetic overflow is ignored & result is truncated
- Expressions using the following predefined operators on integral types:

  ++ — - (unary) + - * /
- Explicit numeric conversions between integral types

# checked & unchecked demo

- int z = 0;
  short x=32767, y=32767;
  try {
  z = **checked**((short)(x + y));
  z = un**checked**((short)(x + y));
  }
  catch (System.OverflowException e){
  Console.WriteLine(e.ToString());
  }
- blocks can be used for expressions



# Objects, classes and structs

- Objects are instances of a given data type. The data type provides a blueprint for the object that is instantiated when application is executed.
- New data types are defined using classes & structs.
- Classes and structs form the building blocks of C# applications, containing code and data. C# application will always contain of at least one class.
- A struct can be considered a lightweight class, ideal for creating data types that store small amounts of data, and does not represent a type that might later be extended via inheritance.
- C# classes support inheritance, meaning they can derive from a previously defined class.

# class

- class may have
  - Methods, Fields, Properties, Operators, Indexers, Events & other user defined types
- Access specifiers
  - private, protected, public
  - internal, protected internal
- Class definitions can be splitted in files
- Static classes are sealed classes with static methods
- Only single inheritance is allowed
- Multiple interface inheritance is allowed



# struct

- Structs are value types; classes are reference types
- When passing a struct to a method, it is passed by value instead of as a reference.
- Unlike classes, structs can be instantiated without using a new operator.
- Structs can declare constructors, but they must take parameters.
- A struct cannot inherited from other stuct.A struct can implement interfaces.
- All structs are inherit from System.ValueType that inherits System.Object
- It is an error to initialize an instance field in struct

```
public class test {
        int x, y;
        public test() { }
        public test(int x,int y)
        {       this.x=x;       this.y=y;       }
        ~test() { }
        public int X {
                        get { return x; }
            protected   set { x=value; }
        }
        public static test operator*(test a, test b)
        {       return new test(a.x*b.x, a.y*b.y);       }
        public static explicit operator complex(test t)
        {       return new complex(t.x, t.y);       }
}
```

can more strict if needed



# Inheritance & Polymorphism

- public class derived : base { ... }
- is operator checks for hierarchy relation
- bool flag = der_obj is base;
- as operator is used for dynamic type casting
- derived newobj = obj as derived;
- abstract classes & members are solely used for inheritance - to define features of derived, non-abstract classes.
- sealed members cannot be overridden
- sealed classes cannot be further inherited; they are final.

# Operator overloading

- By operator overloading, we can extend the meaning of the operator. By this we can use operator with user defined classes.
- The operators are implemented as static functions of the class.
- &&, ||, [ ], ( ), shorthand,=, ., ?:, →, new, is, sizeof operators cannot be overloaded.
- Relational operators like ==, != should be overloaded in pair.
- public static Complex operator +(Complex c1, Complex c2);



# Operator overloading

- Conversion operators are used to convert a class object into any other type and vice versa.
- Conversions declared as implicit occur automatically when required.
- Conversions declared as explicit require a cast to be called.
- public static explicit operator int(Complex c) {
- return c.real;
- }
- public static explicit operator int(Complex c) {
- return c.real;
- }

# new Keyword

- new operator is used to allocated memory.
- When function is declared as new, it hides all the functions in base class with the same name.
- Using a derived class reference we can't access any function with same name in base class
- We can even declare a function as new virtual, which makes function as a new virtual function hence forth in hierarchy that can be overridden in further derived classes using override keyword.
- Using a base class reference we cannot access any function with same signature in the derived class.



# sealed Keyword

- Function declared as sealed cannot be overriden in the derived class.
- Thus sealed functions finalizes functionality provided by the base class, that cannot be modified or extended in the derived class.
- The specifier sealed override tells that function is overridden in this class but cannot be overridden further
- If we wish to have a new function in the derived class with same name we should use new.
- If a class is declared as sealed, then that class cannot be further inherited.

# const, readonly Keywords

- The variable declared as const, is a constant variable and cannot be modified anywhere.
- The const field's value should be given at compile time, where it is declared in class.
- const double PI = 3.142;
- Now value of PI is finalized to 3.142.
- The variable declared as readonly, is assigned in constructor and cannot be modified in any other method afterwards.
- The readonly field's value may not be known at the compile time.



# abstract Keyword

- If function is declared as abstract within a class then we cannot create object of that class. Such class must be declared as abstract.
- Functions declared abstract in base class must be overriden in the derived class. Otherwise we cannot even create object of the derived class.
- Abstract classes are used to implement some common functionality that can be shared by the objects of the derived classes.
- Function declared as abstract override overrides function in base class but make it abstract in derived class.

# virtual, override, sealed, new

```
class A {
        int x;
        public A(int x) { this.x = x; }
        public virtual void f1() { }
        public virtual void f2() { }
        public sealed void f3() { }
        public virtual void f4() { }
}
class B : A {
        int y;
        public B(int x, int y):base(x, y){ this.y=y;}
        public override void f1() { }
        public new virtual void f2() { }
        public sealed override void f4() { }
}
```



# abstract & sealed classes

```
abstract class A {
        public abstract void f1() { }
        public abstract string Name{ get; set; }
        public virtual void f3() { }
}
abstract class B : A {
        public override void f1() { }
        string nm;
        public override string Name {
                get { return nm; }
                set { nm=value; }
        }
        public abstract override void f3() { }
}
sealed class C : B {
        public override void f1() { }
        public override void f3() { }
}
```

# Use of abstract classes

- Abstract class references are used to refer to objects of the derived classes.
- Thus they work like generic references and give specific implementation depending on type of object.
- Also they implement the common functionality used by the derived class objects.
- Generally we create collection like array of base class references and now we can store any objects derived from this common base class.
- Using this technique we can achieve polymorphic behavior.



# interface

- interface is full abstraction of a class.
- In interface all methods are by default public & abstract.
- Interface cannot have any non-abstract method or fields.
- The fields can be declared only as public static const.
- Multiple interface inheritance is allowed, even though implementation inheritance is not allowed.
- If class A is implementing I1 and I2 interfaces, A should override all methods from I1 & I2.
- Fragile Base Class Problem: If base class defination is modified, then all derived classes must be recompiled.
- To avoid such problem we can go for interafaces, as interfaces are immutable.

# interface

- Interfaces are service based. It is a contract between service provider and service user.
- It declares set of rules in form of its methods, that will be implemented by the service provider.
- Interfaces ensure that rules or services are implemented by the service provider.
- Using interface you can collect non-related objects together, by getting a service common to both.
- Example: IPrintable interface has Print() method, that can be implemented by Point or Employee class.
- The implementation will differ according to actual class.



# interface

```
interface A {
        void f1();
        void f2();
}
interface B {
        void f1();
        void f2();
}
class C : A, B {
        public void f1() { }
        void A.f2() { }
        void B.f2() { }
}
```

Common impl of f1() – for intf A & B

Explicit impl of f2() – for intf A & B

## interface property

```
interface X {
    public string Name {
        get;
        set;
    }
}
class Y : X {
    string nm;
    public string Name {
        get{ return nm; }
        set{ nm=value; }
    }
}
```

# Interfaces

- Classes and structs can inherit from more than one interface.
- An interface can itself inherit from multiple interfaces.
- IComparable interface, ICloneable are some of the standard interfaces.
- Interface can have only static and const members.
- const should be known at compile time.
- readonly can be assigned during intialization or constructor



# Constructors

- Ctors can be parameterized or parameterless.
- classname() { } or classname(args) { }
- If constructurs are private, object of class cannot be created outside class.
- We can have copy ctor as well.
- classname(classname other);
- Assuming that f1(), f2(), f3(), f4(), f5(), f6() as static methods of respective classes what will be sequence of their execution?

```
class A {
    int x= f1();
    A() { f2(); }
}
class B : A {
    int y = f3();
    B() { f4(); }
}
class C : B {
    int z = f5();
    C() { f6(); }
}
C ref = new C();
```

# static constructors

- static classname() { }
- static constrcuctors has no access specifiers and also no parameters. And a class can have only one static constructor.
- Static constrcutor can access only static members.
- It is called automatically by CLR to initialize class before instance is created or any static members are referenced
- It cannot be called explicitly.
- It is used for one time initialization, when class is going to be used for any purpose.
- use: keep track in log file about class usage.



# Nested types

- A type defined within a class or struct is called a nested type
- public class Container {

```
    public class Nested {
            private Container m_parent;
            public Nested() { }
            public Nested(Container parent)
            {       m_parent = parent;          }
    }
}
Container.Nested obj = new Container.Nested();
```

- Nested types can access private and protected members of the containing type, including any inherited private or protected members.

# Indexers

- Indexers enable objects to be indexed in a similar way to arrays.
- Indexers do not have to be indexed by an integer value.
- Indexers can be overloaded.
- Indexers can have more than one formal parameter, for example, when accessing a two-dimensional array.
- It can be declared as abstract & can be interface member.

```
class test
{
    int[] a=new int[5];
    public int this[int i]
    {
        get
        { return a[i]; }
        set
        { a[i] = value; }
    }
}
test obj = new test();
obj[2] = 20;
num = obj[2];
```



# Partial class

- Using the partial keyword indicates that other parts of the class, struct, or interface can be defined within the namespace with same access
- public partial class Employee {
    public void DoWork() { }
  }
  public partial class Employee {
    public void GoToLunch() { }
  }

# Partial class

- If any of the parts are declared abstract, then the entire type is considered abstract.
- If any of the parts are declared sealed, then entire type is considered sealed.
- If any of the parts declare a base type, then the entire type inherits that class.
- When working on large projects, spreading a class over separate files allows multiple programmers to work on it simultaneously.
- When working with automatically generated source, code can be added to the class without having to recreate the source file. Visual Studio uses this approach



# Static class

- Static classes and class members are used to create data and functions that can be accessed without creating an instance of the class.
- Static class members can be used to separate data and behavior that is independent of any object identity.
- Static classes are loaded automatically by the CLR when the program or namespace containing the class is loaded.
- Creating a static class is therefore much the same as creating a class that contains only static members and a private constructor.

# Static class

- static class CompanyInfo {
  public static string GetCompanyName()
  { return "CompanyName"; }
  public static string GetCompanyAddress()
  { return "CompanyAddress"; }
  }
- It is useful to organize the methods inside the class in meaningful way, like methods of System.Math
- It contains only static members.
- Its object cannot be created.
- It is sealed(final) class.
- It cannot have instance constructor.



# Object class

- Supports all classes in the .NET Framework class hierarchy and provides low-level services to derived classes.
- This is the ultimate base class of all classes in the .NET Framework; it is the root of the type hierarchy.

| | |
|---|---|
| ToString | Returns a string that represents the current Object. |
| Equals | Determines equility. |
| GetType | Returns type of current Object |

# Exceptions

- Exception is flexible way to deal with any unexpected or exceptional situations that arise while a program is running.
- Exception handling uses the try, catch, and finally keywords to attempt actions that may not succeed, to handle failures and to clean up resources at end
- When exceptional circumstance is encountered, such as a division by zero or low memory warning, an exception is generated by CLR.
- Once an exception occurs, the flow of control immediately jumps to an associated exception handler, if one is present program stops executing with an error message.



# Exceptions

- try block contain code that may result in exception
- An exception handler is catch block of code that is executed when an exception occurs. One try may have mutiple catch blocks. The first **catch** statement that can handle the exception is executed.
- Exceptions can be explicitly generated by a program using the throw keyword.
- Exception objects contain detailed information about the error, including the state of the call stack and a text description of the error.
- Code in a finally block is executed even if an exception is thrown, thus allowing a program to release resources

# Exceptions

- Exceptions are represented by classes derived from Exception.
- If the statement that throws an exception is not within a try block, the runtime checks the calling method for a try statement and catch blocks.
- The runtime continues up the calling stack, searching for a compatible catch block. After the catch block is found and executed, control is passed to the first statement after the catch block.
- Few compiler generated exceptions are OverflowException, DivideByZeroException, IndexOutOfRangeException, InvalidCastException, NullReferenceEception, OutOfMemoryException, ...



# Garbage Collection - GC

- GC manages the allocation and release of memory for your application.
- Each time you use the new operator to create an object, the runtime allocates memory for the object from the managed heap.
- The GC's optimizing engine determines the best time to perform a collection, based upon allocations done.
- When the GC performs a collection, it checks for objects in the managed heap that are no longer being used by the application.

# Destructors and IDisposable

```
using System;
class Class1 {
   static void Main(string[] args) {
        Widget wg = new Widget();
        Console.WriteLine(wg.ToString());
        wg.Dispose(); //specific call to cleanup
        wg = null; //let go of reference and wait for GC
   }
}
public class Widget : IDisposable {
   ~Widget(){
        //cleanup code called by GC
   }
   public void Dispose()   {
        //perform cleanup
   }
}
```

Destructor called by GC

Dispose called by client!



# Delegates

- delegate is a type that references a method. It can be treated as object oriented function pointer.
- public delegate int PerformCalculation(int x, int y);
- Any method that matches the delegate's signature, which consists of the return type and parameters, can be assigned to the delegate.
- It makes delegates ideal for defining callback methods.
- Delegates can be chained together called multicast delegate.
- Delegates are mainly used with Eventing design pattern.

# Delegates

- Delegates without name are called as anonymous methods which reduces coding.
- It can access the variables in the outer class or method.
- When a delegate method has a return type that is more derived than the delegate signature, it is said to be covariant.
- When a delegate method signature has one or more parameters of types that are derived from the types of the method parameters, that method is said to be contravariant.
- Because the method's return type or parameters are more specific than the delegate signature's return type, they can be implicitly converted.



# Events

- Events are used on classes to notify the users of object when some interesting happens related to the object.
- This notification is called raising an event.
- An object raising an event is referred to as the source or sender of the event.
- Objects that represent user interface elements usually raise events in response to user actions like a button click.
- Event has a signature that includes a name and a parameter list & defined by a delegate type.
- Event can be static, virtual, abstract, sealed.

# Events

- Ex: delegate void TestEventDelegate();
  public event TestEventDelegate TestEvent;
- .NET framework provide guideline for creating event. The first parameter should be sender object while second should provide information about event derived from EventArgs.
- Event may have more than one handlers that are executed sequentially like multicast delegate.
- add or remove event accessors can written that are executed when += or -= operator is used on event.
- Event can be declared in interface and overridden into derived class.



# ICloneable

- Clone method creates new instance of class with same value as existing instance
- It can be implemented either as a deep copy or a shallow copy.
- In a deep copy, all objects are duplicated.
- In a shallow copy, only the top-level objects are duplicated and the lower levels contain references.

```
class Date:ICloneable {
public int dd,mm,yy;
public Object Clone() {
    return this.MemberwiseClone();
}
}
class Person:ICloneable {
public String nameFirst,
    nameLast;
public Date birthDate;
public Object Clone() {
Person p =
    this.MemberwiseClone() as
    Person;
p.birthDate=
    this.birthDate.Clone() as Date;
return p;
}
}
```

# Collection classes

- Collection classes are defined as part of the System.Collections or System.Collections.Generic namespace.
- Most collection classes derive from the interfaces ICollection, IComparer, IEnumerable, IList, IDictionary, and IDictionaryEnumerator.
- Some important classes are ArrayList, SortedList, Stack, Queue, HashTable, etc.
- Using generic collection classes provides increased type-safety and in some cases can provide better performance, especially when storing value types.



# Generics in C#

- Generics are template classes or functions written by programmer.
- C# generic type substitutions are performed at runtime.
- Use generic types to maximize code reuse, type safety, and performance.
- Collection classes declared in System.Collections.Generic
- Generic interfaces, classes, methods, events and delegates can be created.

```
class MyList<T> {
   void Add(T val) { }
   void Del(T val) { }
}

static void main() {
   MyList<int> list1 =
   new MyList<int>();

   MyList<string> list2=
   new MyList<string>();

}
```

## Generics

- For every new value type parameter specialized version is created at runtime
- For reference types only version in created for storing refernces as refereces of any type have same size.
- To use any collection with foreach loop it should support iterator or IEnumerable interface.
- Enumerator can be returned by "yield" keyword. We can write more than one yield return statement.
- Yield statement cannot be there in unsafe blocks or try-catch blocks, anonymous methods, method with ref & out params.



## Nullable types

- Nullable types are instances of the System.Nullable struct.
- A nullable type can represent the normal range of values for its underlying value type, plus an additional **null** value. e.g.
- A Nullable<bool> can be assigned the values **true** or **false**, or **null**.

# IComparable, IComparer

- IComparable. CompareTo member functions return 0 for equality and –ve of +ve value for inequality.
- IComparer.Compare compare two objects. Throws exception if any of two objects won't implement IComparable.
- Used with Array.Sort
- The functions should have identity, symmetry, transitivity

```
class Person:
    IComparable,IComparer{
string firstName;
public int CompareTo
    (Object o2){
return
    firstName.CompareTo
    (o2.firstName);
}
public int Compare
    (object o1, object o2) {
return o1.CompareTo(o2);
}
}
```



# Attributes

- Attributes provide a powerful method of associating declarative information with types, methods, properties, etc.
- Attributes add metadata to your program. Metadata is information embedded in your program such as compiler instructions or descriptions of data.
- They are classified as predefined(given in framework) and user defined attributes.
- Predefined attributes are used in PInvoke, Serialization, Interoperability, etc.

# Custom Attributes

- You can create your own custom attributes by defining an attribute class, a class that derives directly or indirectly from Attribute, which makes identifying attribute definitions in metadata fast and easy.
- System.AttributeUsage attribute decides usage of the attribute.
  - AttributeTargets – Class, Struct, Field, Property, All, etc.
  - AllowMultiple = true / false
  - Inherited = true / false



# Serialization

- Serialization is the process of converting the state of an object into a form that can be persisted or transported.
- The complement of serialization is deserialization, which converts a stream into an object.
- Binary serialization preserves type fidelity, which is useful for preserving the state of an object between different invocations of an application.
- In newer versions, Binary Formatter is deprecated.
- XML serialization serializes only public properties and fields and does not preserve type fidelity. This is useful when you want to provide or consume data without restricting the application that uses the data.
- Similarly, JSON serializer converts C# object into JSON string. This feature is internally used by REST services.

# Serialization

- The CLR manages object memory management and provides an automated serialization.
- When an object is serialized, the name of the class, the assembly, and all the data members of the class instance are written to storage.
- When the class is serialized, the serialization engine ensure that the same object is not serialized more than once, if object has references.
- The only requirement placed on object graphs is that all objects, referenced by the serialized object, must also be marked as Serializable.
- When the serialized class is de-serialized, the class is recreated and the values of all the data members are automatically restored.



# Serialization

- Serialization can be done as Binary or XML or SOAP or JSON serialization.
- Steps for serialization
  - Mark the class with Serializable attribute.
  - Create a formatter object – Binary or SOAP
  - Create FileStream object to store data
  - Serialize object into file using formatter
  - Close the stream
- If you don't want to store any member then it should be marked with NonSerialized attribute.
- You can also use XMLSerializer object with StreamWriter object to write in file.

# Platform Invoke

- Platform invoke is a service that enables managed code to call unmanaged functions implemented in dynamic link libraries (DLLs), such as those in the Win32 API.
- It locates and invokes an exported function and marshals its arguments across the interoperation boundary as needed



# Unsafe Code Blocks

- Must set compiler option in project properties
  - Project->Properties->Configuration Properties->Build ->Allow Unsafe Code blocks – set to true

```
[DllImport("kernel32", SetLastError=true)]
static extern unsafe bool ReadFile(int hFile,
      void* lpBuffer, int nBytesToRead,
      int* nBytesRead, int overlapped);
public unsafe int Read(byte[] buffer, int index, int
  count) {
  int n = 0;
  fixed(byte* p = buffer) {
      ReadFile(handle, p + index, count, &n, 0);
  }
  return n;
}
```

Marked as Unsafe due to pointers

Marked fixed to keep object from moving when GC runs

# Reflection

- CLR loads the assembly into appdomain.
- Assemblies contain modules, modules contain types, and types contain members.
- Reflection provides objects that encapsulate assemblies, modules, and types.
- Reflection can be used to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object.
- The related classes are declared in System.Reflection namespace.
- The classes of the System.Reflection.Emit namespace provide a specialized form of reflection that enables you to build types at run time.



# Reflection basics

- The GetType method or typeof operator provides Type structure that keeps information of type.
- Providing access to nonpublic information involves security risks. Code that is allowed to discover nonpublic information on a type can potentially access code, data, and other information you want to keep private.
- Therefore, .NET Framework security enforces rules that determine to what degree reflection can be used to discover type info and access types.
- Depending on the operation being performed, a ReflectionPermission or the SecurityPermission for serialization might be required.

# Reflection classes

- Use Assembly to define and load assemblies, load modules that are listed in the assembly manifest, and locate a type from this assembly and create an instance of it.
- Use Module to discover information such as the assembly that contains the module and the classes in the module, all global methods or other specific, nonglobal methods defined on the module.
- Use ConstructorInfo to discover information such as the name, parameters, access modifiers and implementation details (**abstract** or **virtual**) of a constructor. Use the GetConstructors or GetConstructor method of a Type to invoke a specific constructor.



# Reflection classes

- Use MethodInfo to discover information such as the name, return type, parameters, access modifiers and implementation details of a method. Use the GetMethods or GetMethod method of a Type to invoke a specific method.
- Use FieldInfo to discover information such as the name, access modifiers and implementation details of a field, and to get or set field values.
- Use EventInfo to discover information such as the name, event-handler data type, custom attributes, declaring type, and reflected type of an event, and to add or remove event handlers.

# Reflection classes

- Use PropertyInfo to discover information such as the name, data type, declaring type, reflected type, and read-only or writable status of a property, and to get or set property values.
- Use ParameterInfo to discover information such as a parameter's name, data type, whether a parameter is an input or output parameter, and the position of the parameter in a method signature.
- Use CustomAttributeData to discover information about custom attributes when you are working in the reflection-only context of an application domain. CustomAttributeData allows you to examine attributes without creating instances of them.



# Thread basics

- Preemptive multitasking OS divides the available processor time among the threads that need it, allocating a processor time slice to each thread one after another.
- The currently executing thread is suspended when its time slice elapses, and another thread resumes running.
- When the system switches from one thread to another, it saves the thread context of the preempted thread and reloads the saved thread context of the next thread in the thread queue.
- The length of the time slice is optimized to get good performance.

# Threading

- Operating systems use processes to separate the different applications that they are executing.
- Threads are the basic unit to which an OS allocates processor time, and more than one thread can be executing code inside that process.
- The .NET Framework subdivides an OS process into lightweight managed subprocesses, called application domains(System.AppDomain).
- One or more threads (System.Threading.Thread) can run in one or any number of application domains within the same managed process.
- Each Appdomain is having at least one thread.
- Thread can easily switch into Appdomains



# Thread basics

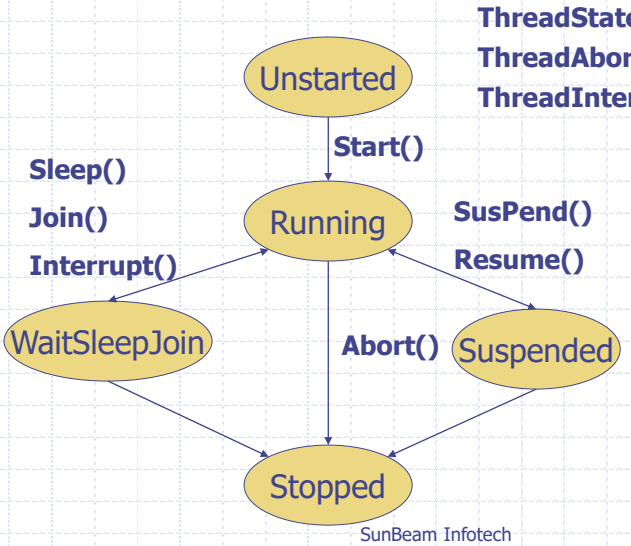
- Operating systems use processes to separate the different applications that they are executing.
- Thread is basic unit to which OS allocates cpu time
- Process may have more than one thread
- Framework subdivides OS process into lightweight managed subprocesses called AppDomains
- One or more managed threads can run in one or more of appdomains within the same process
- Each appdomain starts with a single thread
- It can create additional application domains and additional threads.
- Managed thread can move freely between appdomains inside the same managed process.

# Win32 API Vs .NET Fns

| | |
|---|---|
| CreateThread | new Thread() & Thread.Start() |
| TerminateThread | Thread.Abort |
| SuspendThread | Thread.Suspend |
| ResumeThread | Thread.Resume |
| Sleep | Thread.Sleep |
| WaitForSingleObject | Thread.Join |
| GetCurrentThread | Thread.CurrentThread |
| SetThreadPriority | Thread.Priority |
| no equivalent | Thread.Name |
| no equivalent | Thread.IsBackground |



# Thread States & Priorities




ThreadPriority

Highest

AboveNormal

Normal

BelowNormal

Lowest

# Thread Synchronization

- To avoid conflicts, you must synchronize, or control the access to, shared resources.
- Failure to synchronize access properly can lead to problems such as deadlocks and race conditions
- The system provides synchronization objects that can be used to coordinate resource sharing among multiple threads.
- Resources that require synchronization include:
  - System resources (communications ports)
  - Resources shared by multiple processes (such as file handles).
  - The resources of a single application domain accessed by multiple threads.



# Thread synchronization

- Thread synchronization is done by lock keyword. It is internally using Monitor object for that.
- lock(obj) { }
- This is equivalent to
  - Monitor.Enter or Monitor.TryEnter
  - Monitor.Exit
- The object passed to Enter() should be same as Exit(), otherwise Monitor throws SynchronizationLockException.
- You can also use extra methods like Wait(), Pulse() and PulseAll().

# Advantages of Thread

- Using more than one thread is most powerful technique available to increase responsiveness to the user and process the data necessary to get the job done at almost the same time.
- Your single application domain could use multiple threads to accomplish:
  - Communicate over a network, to a Web server, and to a database.
  - Perform operations that take large amount of time
  - Distinguish tasks of varying priority. For example, a high-priority thread manages time-critical tasks, and a low-priority thread performs other tasks.
  - Allow the user interface to remain responsive, while allocating time to background tasks.



# Disadvantages of Thread

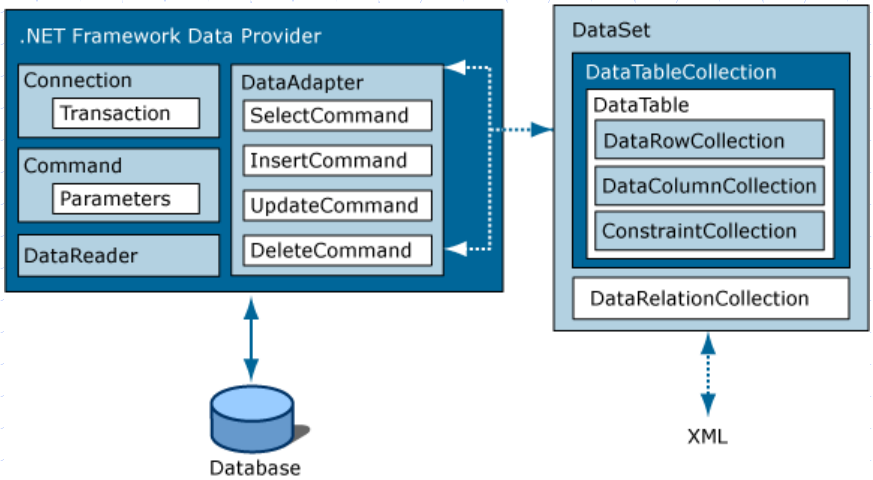
- It is recommended that you use as few threads as possible, thereby minimizing the use of OS resources and improving performance.
- The resource requirements are as follows:
  - The system consumes memory for the context information required by processes, AppDomain objects, and threads.
  - Keeping track of a large number of threads consumes significant processor time.
  - Controlling code execution with many threads is complex, and can be a source of many bugs.
  - Destroying threads requires knowing what could happen and handling those issues.
  - Shared access to resources can create conflicts.

# ADO.NET Components

- Two components of ADO.NET that you can use to access and manipulate data:
  - .NET Framework data provider
  - The DataSet
- .NET Framework data provider
  - For SQL Server : System.Data.SqlClient
  - For OLEDB : System.Data.OleDb
  - For ODBC : System.Data.
  - For Oracle : System.Data.OracleClient
- DataSet is an in-memory cache of data retrieved from a data source. It is a virtual data store.



# ADO.NET Architecture

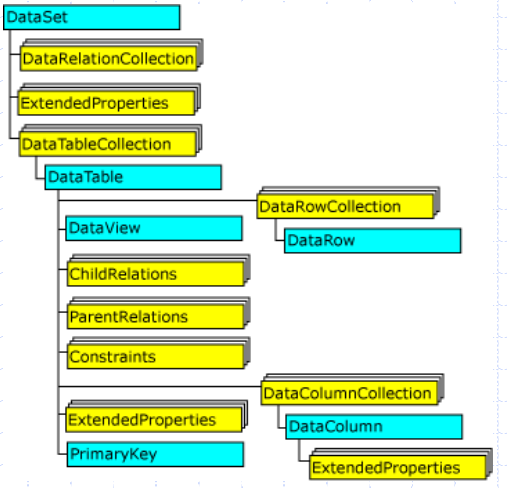


# Core Objects of Data Providers

- Connection : Establishes a connection to a specific data source
- Command : Executes a command against a data source. Exposes Parameters and can execute within the scope of a Transaction
- DataReader : Reads a forward-only, read-only stream of data from a data source
- DataAdapter : Populates a DataSet and resolves updates with the data source
- Important objects: Transaction, CommandBuilder, Parameter, Exception



# DataSet Object Model



# File IO

- The abstract base class Stream supports reading and writing bytes.
- Streams involve these fundamental operations:
  - Streams can be read from.
  - Streams can be written to.
  - Streams can support seeking.
- Directory and File class provides static methods for create, move, copy, delete dirs and files.
- DirectoryInfo and FileInfo classes provide instance methods for the same.
- FileStream class actually open the file.



# Common Stream classes

- FileStream supports random access to files through its Seek method. FileStream opens files synchronously by default, but supports asynchronous operation as well.
- A BufferedStream is a Stream that adds buffering to another Stream such as a NetworkStream.
- A CryptoStream links data streams to cryptographic transformations.
- A MemoryStream is a nonbuffered stream whose encapsulated data is directly accessible in memory.
- A NetworkStream represents a Stream over a network connection.

# Reader and Writer classes

- Basic Stream class provide byte IO support.
- BinaryReader and BinaryWriter read and write encoded strings and primitive data types from and to Streams.
- StreamReader reads characters from Streams, using Encoding to convert characters to and from bytes. StreamWriter writes characters to Streams, using Encoding to convert characters to bytes.
- StringReader reads characters from Strings. StringWriter writes characters to Strings.
- TextReader/TextWriter are the abstract base classes for Stream & String Reader/Writer classes. They provide unicode char IO.



# Auto Implemented Props (3.0)
# Object Initializer (3.0)

- Properties are used to provide controlled access to fields.
- Simplified properties (just get/set) can be easily implemented using auto implemented properties.
- They can also be used for initialization while creating object of the class.
- When using this initializer syntax, a default paramless constructor followed by setter properties.

# Auto Implemented Props (3.0)
# Object Initializer (3.0)

```
class Box {
    public int Length
    {   get;  set;  }
    public int Breadth
    {   get;  set;  }
    public int Height
    {   get;  set;  }
}
```

```
class Program {
    static void Main() {
        Box b = new Box {
                Length=6,
                Breadth=4,
                Height=2
                };
        // …
    }
}
```



# Implicitly typed Local Var (3.0)
# Anonymous Types (3.0)

- Local variables can be declared without specifying its data type using **var** .
- In this case compiler detect the type of the variable during compilation.
- var a=10;  same as  int a=10;
- var p = new Person();
- Objects of anonymous types should be accessed using var keyword.
- var p = new {Roll=12, Name="Nilesh"};

# Named Parameters (4.0)

- C# allows specifying params names during method call
- It increases readability of program
- Also specifying names for params allows you to change the order of params.

```
void Fun(string name,
    string msg) {
    String res = msg + ",
    "+ name;
    Console.WriteLine(res);
}
static void Main() {
    Fun(msg:"Hi",
    name="Nilesh");
}
```



# Default Parameters (4.0)

- C# supports default params (like C++).
- If params aren't passed during method call, their default values are used.
- The default params should be rightmost params of method.

```
static int Sum(int a, int
    b=0, int c=0) {
    return a + b + c;
}
static void Main() {
    int r;
    r=Sum(1,2,3);
    r=Sum(1,2);
    r=Sum(1);
}
```

# Extension Methods (3.0)

- Extension methods are static methods (in static class) written outside the class; But can used as (looks like) member functions.
- This feature is typically used to provide extra functionality to existing class; even if its source code is not available.
- Using this feature we can associate extra functionality with user-defined as well as pre-defined class.



# Extension Methods (3.0)

```
public static class MyExt {
    public static int Sqr(this int num) {
        return num * num;
    }
}
public class Program {
    static void Main() {
        int abc = 12;
        Console.WriteLine("Sqr : {0}", abc.Sqr());
    }
}
```

Nilesh Ghule <nilesh@sunbeaminfo.com>

# THANK YOU!