DIGITAL NOTES
ON

Programming for Application Development
(R18A1206)

B.TECH IV YEAR - I SEM
(2021-22)

DEPARTMENT OF INFORMATION TECHNOLOGY

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)
(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)
Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, INDIA.

(R18A1206) PROGRAMMING FOR APPLICATION DEVELOPMENT

Course Objectives:

1. To get an overview of the various technologies, which can help in the implementation of the various liveProject.
2. To Understand the Basic Concepts ofC#
3. To Understand the Exception HandlingMechanisms
4. To Understand the Various Concepts of .netAssemblies

UNIT I: MS.NET Framework Introduction: The .NET Framework - an Overview- Framework Components – Framework Versions-Types of Applications ,MS.NET Namespaces - MSIL / Metadata and PE files- Common Language Runtime (CLR) - Managed Code -MS.NET Memory Management / Garbage Collection -Common Type System (CTS) - Common Language Specification (CLS)- Types of JIT Compilers.

UNIT II: Developing Console Application: Introduction to Project and Solution in Studio- Entry point method - Main. - Compiling and Building Projects -Using Command Line Arguments - Importance of Exit code of an application- Different valid forms of Main- Compiling a C# program using command line utility CSC.EXE-Data types - Global, Stack and Heap Memory- Common Type System-Reference Type and Value Type- Data types & Variables Declaration- Implicit and Explicit Casting

UNIT III: Object Oriented Programming: Object -Lifecycle of an Object-relationship between Class and Object-Define Application using Objects-Principles of Object Orientation- Encapsulation –Inheritance-Polymorphism- Encapsulation is binding of State and Behavior together understand the difference between object and reference- Working with Methods, Properties - Copy the reference in another reference variable Exception Handling: Exception -Rules for Handling Exception - Exception classes and its important properties - Understanding & using try, catch keywords -Throwing exceptions-Importance of finally block- & quot ; using & quot; Statement -Writing Custom Exception Classes

UNIT IV: Delegates And Events: Understanding the .NET Delegate type, defining a Delegate Type in C#, The System. Multicast Delegate and System. Delegate Base Classes, PROGRAMMING WITH .NET ASSEMBLIES: Configuring .NET Assemblies, defining Custom Namespaces, The role of .NET Assemblies, Understanding the Format of a .NET assembly, Building and Consuming a Single-File Assembly, Building and Consuming a Multifile Assembly, Understanding Private Assembly, Understanding Shared Assembly, Consuming a Shared Assembly, Configuring Shared assemblies.

UNIT V: ADO.NET PART - I: The Connected Layer: A High-Level Definition of

ADO.NET, Understanding ADO.NET Data Provider, Additional ADO.NET Namespaces, The Types of the System.Data.namespace, Abstracting Data Providers Using Interfaces, Creating the Auto Lot Database. Library.

TEXT BOOKS:

1.Andrew Troelsen (2010), Pro C# and the .NET 4 Platform, 5th edition, Springer (India) Private Limited, New Delhi,India.

REFERENCE BOOKS:

1.E. Balagurusamy (2004), Programming in C#, 5th edition, Tata McGraw-Hill, New Delhi, India.

2.Herbert Schildt (2004), The Complete Reference: C#, Tata McGraw-Hill, New Delhi, India.

3.Simon Robinson, Christian N agel, Karli Watson, Jay Gl (2006), Professional C#, 3rd edition, Wiley &Sons,India.

Course Outcome:

Upon completion of the subject, students will be able to:

1. Implementation of Oops Concepts inASP.net
2. Be able to Implement, Compile, Test & Run ApplicationsPrograms.
3. Demonstrate the ability to use Exception HandlingMechanisms.
4. Able to Develop Applications usingAsp.net,C#.

UNIT I
MS.NET Framework Introduction

The .NET Framework - an Overview:

.NET is a software framework which is designed and developed by Microsoft. The first version of the .Net framework was 1.0 which came in the year 2002. In easy words, it is a virtual machine for compiling and executing programs written in different languages like C#, VB.Net etc.
It is used to develop Form-based applications, Web-based applications, and Web services. There is a variety of programming languages available on the .Net platform, VB.Net and C# being the most common ones. It is used to build applications for Windows, phone, web, etc. It provides a lot of functionalities and also supports industry standards.
.NET Framework supports more than 60 programming languages in which 11 programming languages are designed and developed by Microsoft. The remaining Non-Microsoft Languages which are supported by .NET Framework but not designed and developed by Microsoft.
11 Programming Languages which are designed and developed by Microsoft are:

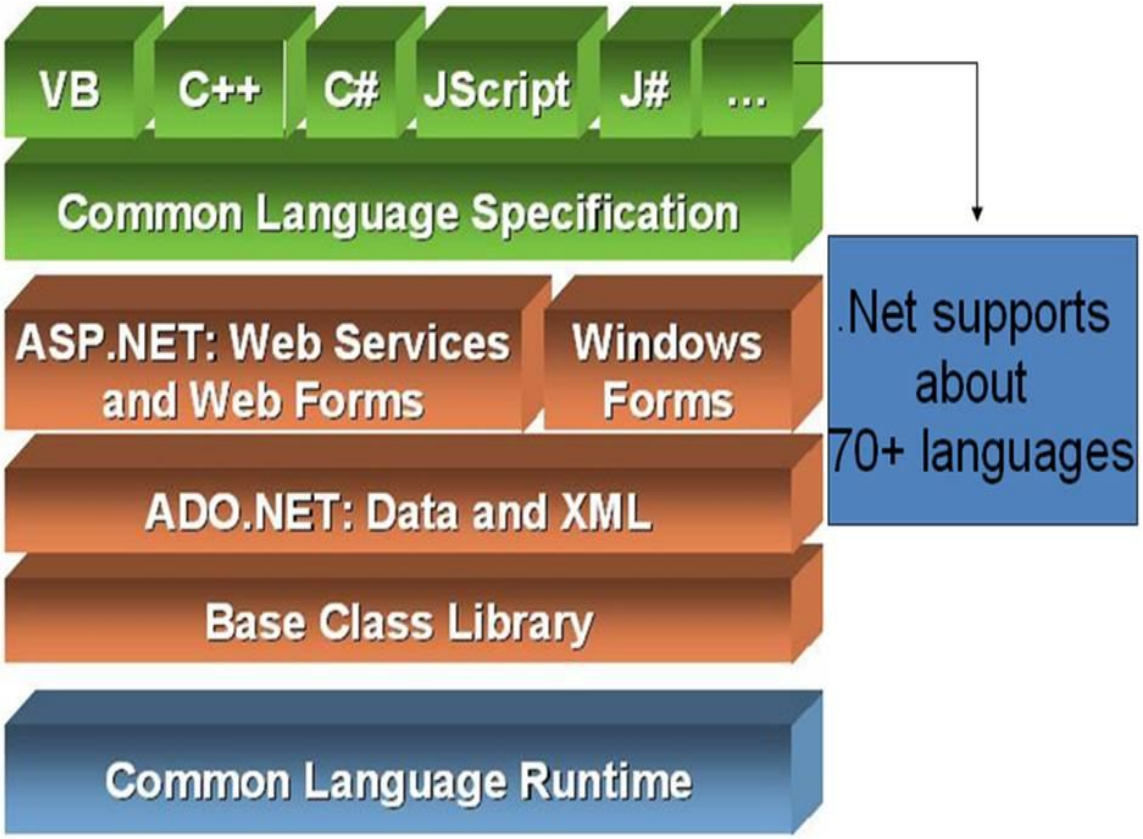
- C#.NET
- VB.NET
- C++.NET
- J#.NET
- F#.NET
- JSCRIPT.NET
- WINDOWS POWERSHELL
- IRON RUBY
- IRON PYTHON
- C OMEGA
- ASML(Abstract State Machine Language)

The .NET Framework is a class of reusable libraries (collection of classes) given by Microsoft to be used in other .Net applications and to develop, build and deploy many types of applications on the Windows platform including the following:

- Console Applications
- Windows Forms Applications
- Windows Presentation Foundation (WPF) Applications
- Web Applications
- Web Services
- Windows Services
- Services-oriented applications using Windows Communications Foundation (WCF)
- Workflow-enabled applications using Windows Workflow Foundation(WF)

The .NET framework is a pure object oriented, that similar to the Java language. But it is not a platform independent as the Java. So, its application runs only to the windows platform. The main objective of this framework is to develop an application that can run on the windows platform. The current version of the .Net framework is 4.8.

Framework Components:



Net Framework is a platform that provides tools and technologies to develop Windows, Web and Enterprise applications.

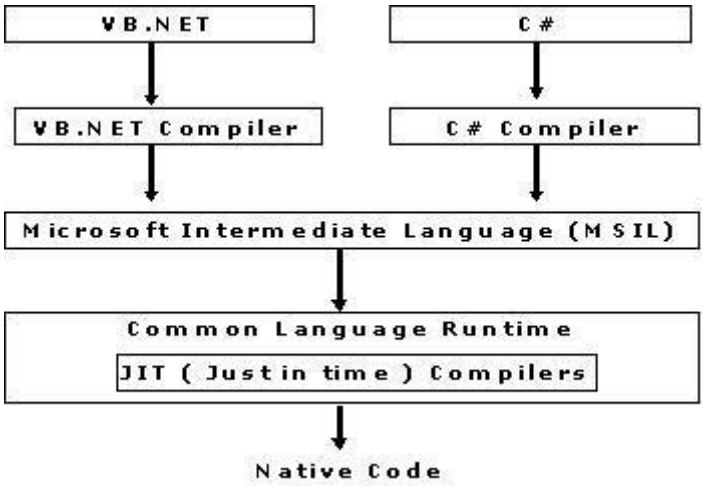
1. Common Language Runtime (CLR)
2. .Net Framework Class Library.
3. Common Type System (CTS)
4. Common Language Specification (CLS)

1. Common Language Runtime (CLR)

.Net Framework provides runtime environment called Common Language Runtime (CLR).It provides an environment to run all the .Net Programs. The code which runs under the CLR is called as Managed Code. Programmers need not to worry on managing the memory if the programs are running under the CLR as it provides memory management and thread management.

Programmatically, when our program needs memory, CLR allocates the memory for scope and de-allocates the memory if the scope is completed.

Language Compilers (e.g. C#, VB.Net, J#) will convert the Code/Program to Microsoft Intermediate Language (MSIL) intern this will be converted to Native Code by CLR.



There are currently over 15 language compilers being built by Microsoft and other companies also producing the code that will execute under CLR.

2. .Net Framework Class Library (FCL)

This is also called as Base Class Library and it is common for all types of applications i.e. the way you access the Library Classes and Methods in VB.NET will be the same in C#, and it is common for all other languages in .NET.

The following are different types of applications that can make use of .net class library.
1. Windows Application.
2. Console Application
3. Web Application.
4. XML Web Services.
5. Windows Services.

In short, developers just need to import the BCL in their language code and use its predefined methods and properties to implement common and complex functions like reading and writing to file, graphic rendering, database interaction, and XML document manipulation.

Below are the few more concepts that we need to know and understand as part of this .Net framework.

3. Common Type System (CTS)

It describes set of data types that can be used in different .Net languages in common. (i.e), CTS ensures that objects written in different .Net languages can interact with each other.

For Communicating between programs written in any .NET complaint language, the types have to be compatible on the basic level.

The common type system supports two general categories of types:

Value types:

Value types directly contain their data, and instances of value types are either allocated on the stack or allocated inline in a structure. Value types can be built-in (implemented by the runtime), user-defined, or enumerations.

Reference types:

Reference types store a reference to the value's memory address, and are allocated on the heap. Reference types can be self-describing types, pointer types, or interface types. The type of a reference type can be determined from values of self-describing types. Self-describing types are further split into arrays and class types. The class types are user-defined classes, boxed value types, and delegates.

4. Common Language Specification (CLS)

It is a sub set of CTS and it specifies a set of rules that needs to be adhered or satisfied by all language compilers targeting CLR. It helps in cross language inheritance and cross language debugging.

Common language specification Rules:

It describes the minimal and complete set of features to produce code that can be hosted by CLR. It ensures that products of compilers will work properly in .NET environment.

Sample Rules:
1. Representation of text strings
2. Internal representation of enumerations
3. Definition of static members and this is a subset of the CTS which all .NET languages are expected to support.
4. Microsoft has defined CLS which are nothing but guidelines that language to follow so that it can communicate with other .NET languages in a seamless manner.

Framework Versions:

.NET Version	CLR Version	Development tool	Windows Support
1.0	1.0	Visual Studio .NET	XP SP1
1.1	1.1	Visual Studio .NET 2003	XP SP2, SP3
2.0	2.0	Visual Studio 2005	N/A
3.0	2.0	Expression Blend	Vista

.NET Version	CLR Version	Development tool	Windows Support
3.5	2.0	Visual Studio 2008	7, 8, 8.1, 10
4.0	4	Visual Studio 2010	N/A
4.5	4	Visual Studio 2012	8
4.5.1	4	Visual Studio 2013	8.1
4.5.2	4	N/A	N/A
4.6	4	Visual Studio 2015	10 v1507
4.6.1	4	Visual Studio 2015 Update 1	10 v1511
4.6.2	4	N/A	10 v1607
4.7	4	Visual Studio 2017	10 v1703
4.7.1	4	Visual Studio 2017	10 v1709
4.7.2	4	Visual Studio 2017	10v 1803
4.8	4	Visual Studio 2019	10v 1809

Types of Applications:

Window client applications: Applications that run on windows O.S Windows forms and WPF(Windows presentation formation) are two major technologies to develop Windows applications.

Eg: MS office, Internet Explorer , Skype, Photoshop ,Paint brush etc

Components and Controls: Libraries used to build something that is easily sharable and distributable.

Eg: chart control, GPS library

Web Applications:

Ex:msn.com,Facebook.com

UWP(Universal window platform) Apps:
These are the apps that run on windows 10 or later platforms.

Mobile Apps:
C# supports native mobile App development Via Xamarin,it is a part of Visual studio 2017 or later versions.

Cloud and Azure:
Visual studio 2017 or later versions provide a complete suite of tools to build cloud based applications for Azure.

Bleeding Edge Technologies:
C# fully supports trending technologies development such as AI, Machine learning, Block Chain, Internet of things and intelligent cloud.

MS.NET Namespaces:

Namespaces are heavily used in C# programming in two ways

1. NET uses namespaces to organize its many classes, as follows:

```
System.Console.WriteLine("Hello World!");
```

System is a namespace and Console is a class in that namespace. The using keyword can be used so that the complete name is not required, as in the following example:

```
using System;
Console.WriteLine("Hello");
Console.WriteLine("World!");
```

2. Declaring your own namespaces can help you control the scope of class and method names in larger programming projects. Use the namespace keyword to declare a namespace, as in the following example:

C# namespace example: by fully qualified name

```
using System;
namespace First {
public class Hello
{
public void sayHello()
{
Console.WriteLine("Hello First Namespace"); }
} }
```

```
namespace Second
{
public class Hello
{
public void sayHello()
{
Console.WriteLine("Hello Second Namespace"); }
}
}


public class TestNamespace
{
public static void Main()
{
First.Hello h1 = new First.Hello();
Second.Hello h2 = new Second.Hello();
h1.sayHello();
h2.sayHello();
}
}
```

C# namespace example: by using keyword

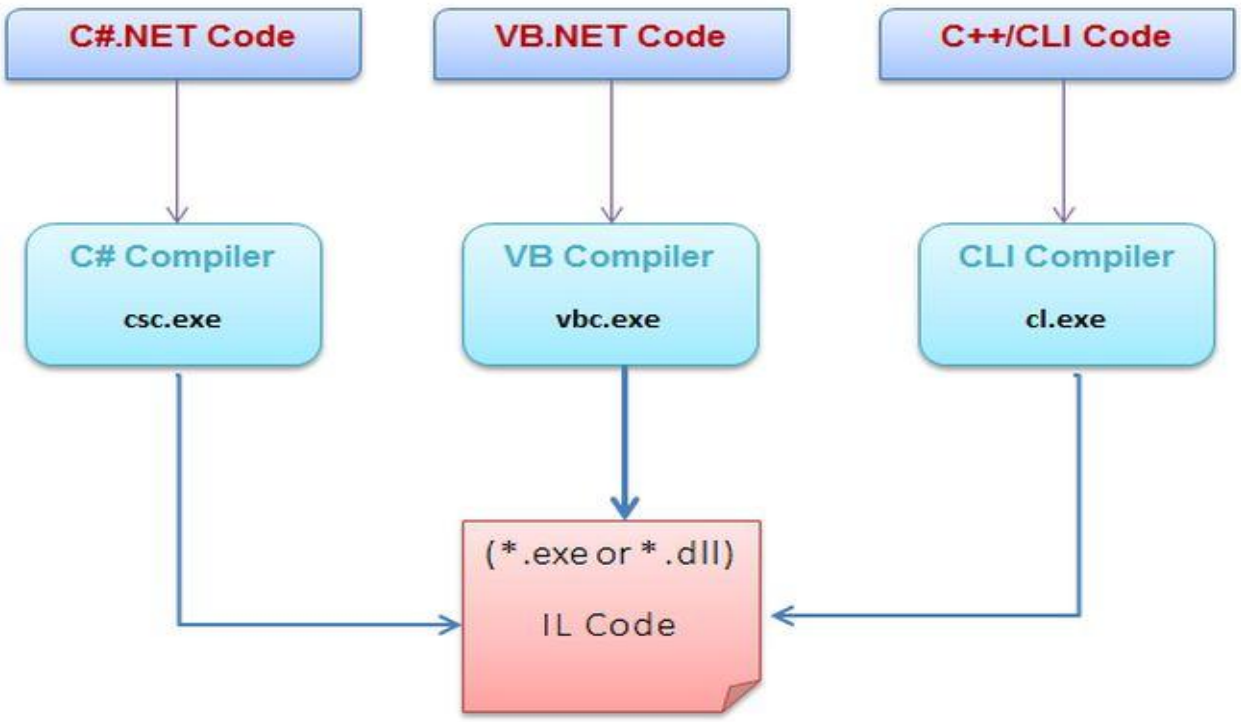
```
using System;
using First;
using Second;
namespace First {
public class Hello
{
  public void sayHello() { Console.WriteLine("Hello Namespace"); }
}
}



namespace Second
{
   public class Welcome
   {
     public void sayWelcome()
{
 Console.WriteLine("Welcome Namespace");
 }
   }
}
```

```
public class TestNamespace
{
   public static void Main()
   {       Hello h1 = new Hello();
     Welcome w1 = new Welcome();
     h1.sayHello();
     w1.sayWelcome();
  } }
```

MSIL / Metadata and PE files:

MSIL

MSIL-Microsoft Intermediate language also known as common Intermediate Language(CIL)or IL. When a *.dll or *.exe has been created using a .NET-aware compiler, the binary blob is termed an assembly, an assembly contains CIL code and metadata



Metadata

Metadata describes the characteristics of every "type" living within the binary.

For example, if you have a class named Sports Car, the type metadata describes details such as Sports Car's base class, which interfaces are implemented by Sports Car (if any), as well as a full description of each member supported by the Sports Car type.

NET metadata is always presented within an assembly, and is automatically generated by a .NET-aware language compiler.

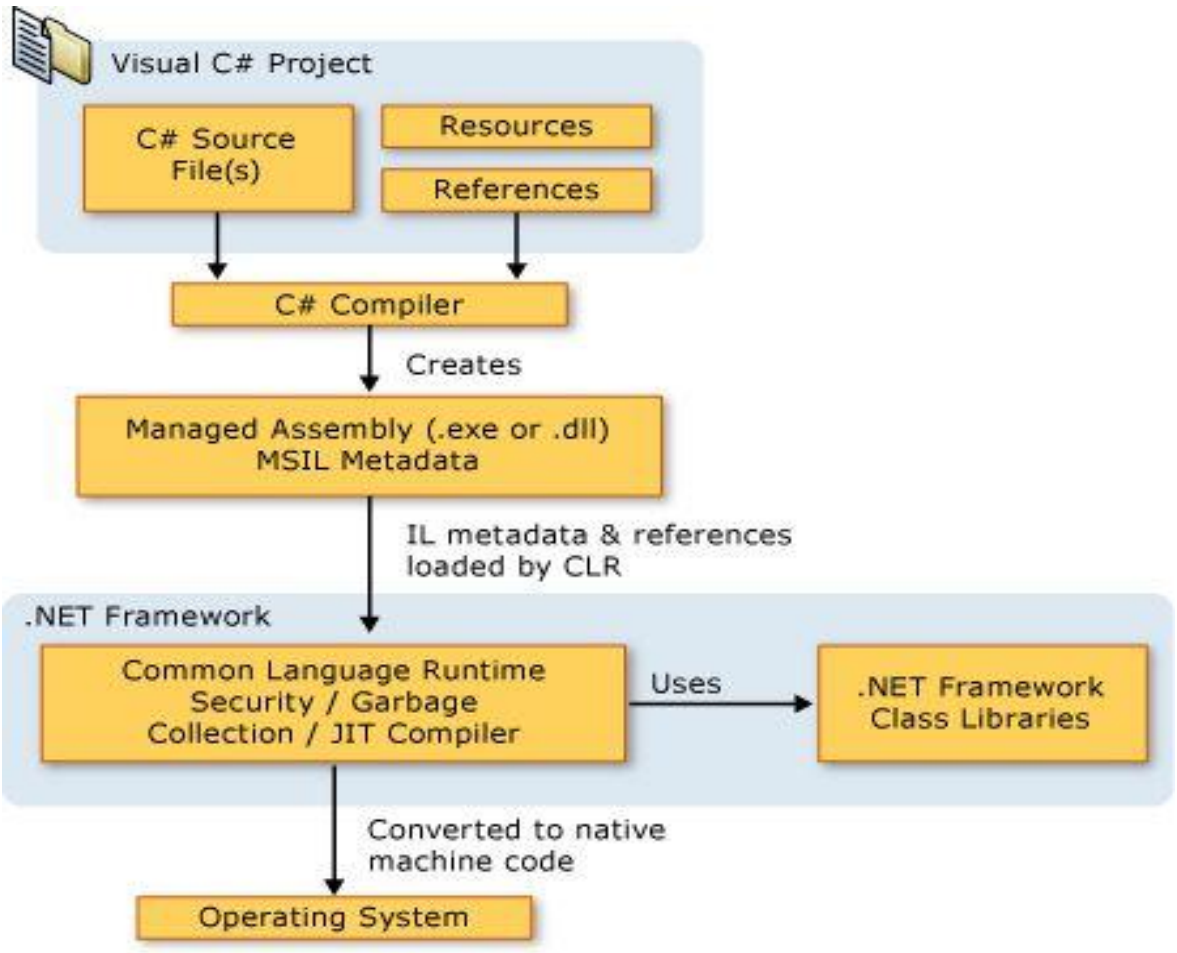
In addition to CIL and type metadata, assemblies themselves are also described using metadata, which is officially termed a manifest. The manifest contains information about the current version of the assembly.

PE Files

The Portable Executable (PE) format is a file format for executables, object code, and DLLs, used in 32-bit and 64-bit versions of Windows operating systems. The PE file format was defined to provide the best way for the Windows Operating System to execute code and also to store the essential data which is needed to run a program.

COMMON LANGUAGE RUNTIME(CLR):

CLR is the heart of .NET framework. Common Language Runtime (CLR) manages the execution of programs written in any language that uses the .NET Framework, for example C#, VB.Net, F# and so on. Programmers write code in any language, including VB.Net, C# and F# when they compile their programs into an intermediate form of code called CIL in a portable execution file (PE) that can be managed and used by the CLR and then the CLR converts it into machine code to be executed by the processor.



Functions of .NET CLR

- Converts CIL CODE
- Exception handling
- Type safety
- Memory management (using the Garbage Collector)
- Security
- Improved performance
- Language independency
- Platform independency

Components of .NET CLR

- Class Loader - Used to load all classes at run time.
- MSIL to Native code - The Just In Time (JIT) compiler will convert MSIL code into native code.
- Code Manager - is responsible for loading .net application code in to memory and manage it until .net application is closed.
- Garbage Collector - It manages the memory. Collect all unused objects and de allocate them to reduce memory.
- Thread Support - It supports multithreading of our application.
- Exception Handler - It handles exceptions at run time.

Types of codes:

- Managed Code
- Unmanaged Code

Managed Code

The code, which is developed in .NET framework, is known as Managed code. This code is directly executed by CLR with help of managed code execution. Managed code uses CLR which in turns looks after your applications by managing memory, handling security, allowing cross - language debugging, and so on.

Unmanaged Code

The code, which is developed outside .NET Framework is known as unmanaged code.
Unmanaged code is executed with help of wrapper classes.

MS.NET Memory Management / Garbage Collection:

In the common language runtime (CLR), the garbage collector (GC) serves as an automatic memory manager. When a new process is started, the runtime reserves a region of address space for the process called the managed heap. Objects are allocated in the heap contiguously one after another.

Memory Management-The Basics of Object Lifetime

The process of releasing memory is called garbage collection. The golden rule of .NET memory management is simple: Allocate a class instance onto the managed heap using the new keyword and forget about it.

```
void MakeACar()
{
// If myCar is the only reference to the Car object,
 // it *may* be destroyed when this method returns.
Car myCar = new Car();
 }
```

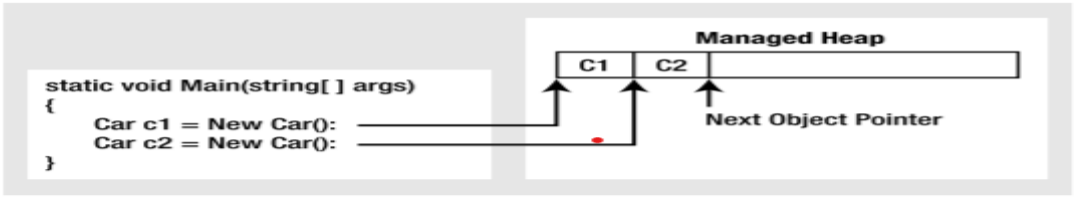
once this method call completes, the myCar reference is no longer reachable, and the associated Car object is now a candidate for garbage collection. When the C# compiler encounters the new keyword, it emits a CIL newobj instruction into the method implementation.

The newobj instruction tells the CLR to perform the following core operations:

• Calculate the total amount of memory required for the object to be allocated (including the memory required by the data members and the base classes).

• Examine the managed heap to ensure that there is indeed enough room to host the object to be allocated. If there is, the specified constructor is called and the caller is ultimately returned a reference to the new object in memory.

• Finally, before returning the reference to the caller, advance the next object pointer to point to the next available slot on the managed heap.



Application Roots :

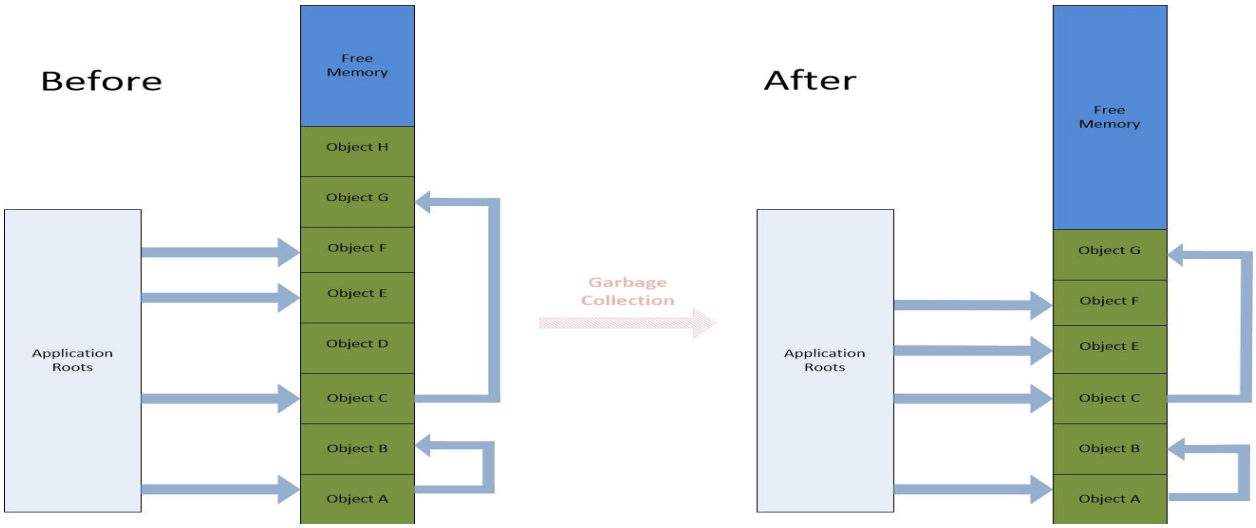
a root is a storage location containing a reference to an object on the managed heap.

or

Roots are memory locations that are designated to be always reachable and which contains references to objects created by the program.

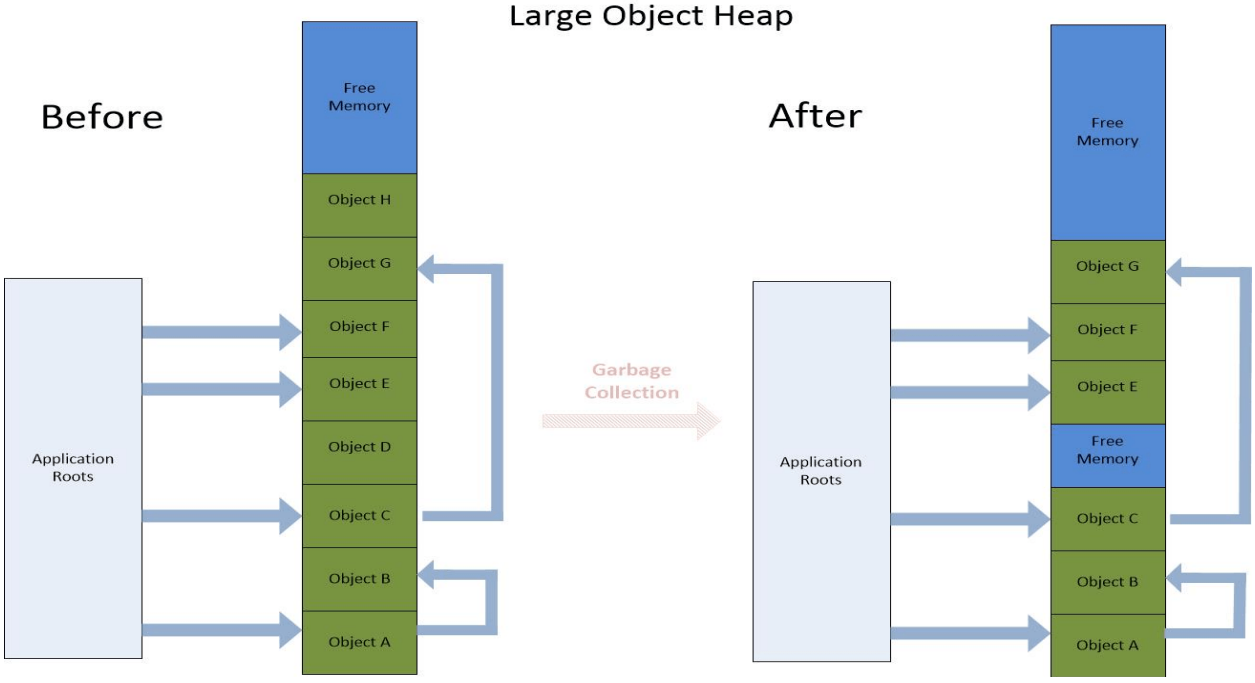
A root can fall into any of the following categories:

- References to any static objects/static fields.
- References to local objects within an application's code base
- References to object parameters passed into a method



Garbage collection -Large objects

The Common Language Runtime (CLR) allocates large objects on the Large Object Heap (LOH).garbage collector never compacts this heap. There may not be enough room for a new object even with enough available memory. This causes the CLR to throw an inappropriately named OutOfMemoryException.

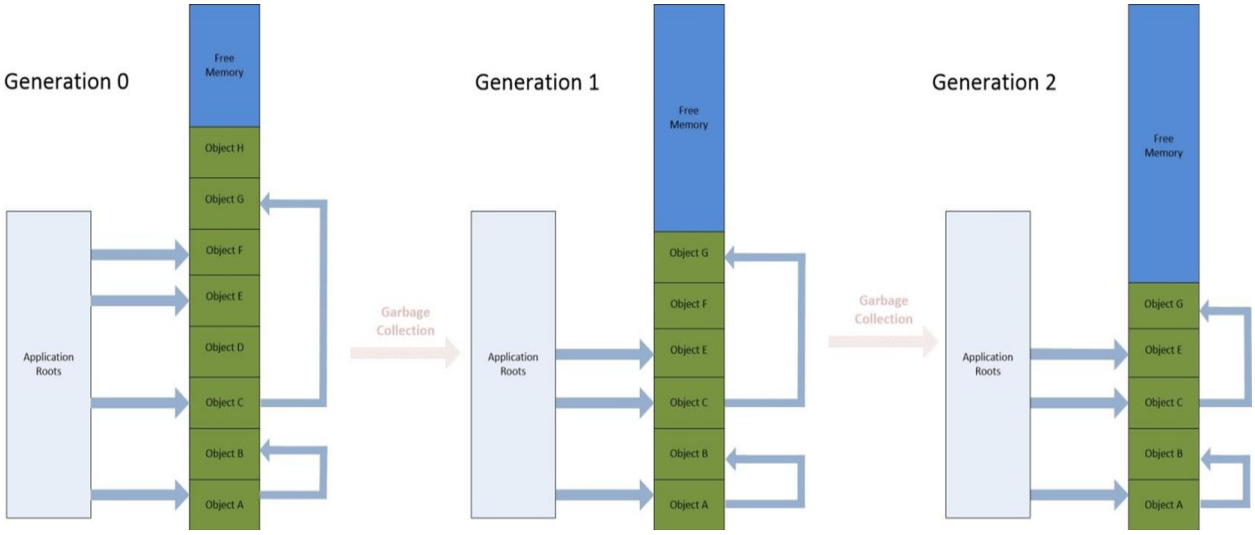


Garbage collection -Small objects

The garbage collector uses three segments, called generations, for small objects:

- Generation 0
- Generation 1
- Generation 2

The CLR allocates memory for new objects in Generation 0. When this heap is full, the garbage collector discards objects no longer in use and promotes surviving objects (referred to as live objects) to Generation 1.
The Generation 0 heap is then available to hold more newly created objects. When insufficient, the garbage collector repeats this process for Generation 1, and if still insufficient, continues with Generation 2.
The bottom line is newer objects (such as local variables) will be removed quickly, while older objects (such as a program's application object) are not "bothered" as often.



Common Type System:

At the time of compilation, all language-specific data types are converted into CLR's data type.
This data type system of CLR which is common to all programming languages of .NET is known as CTS.

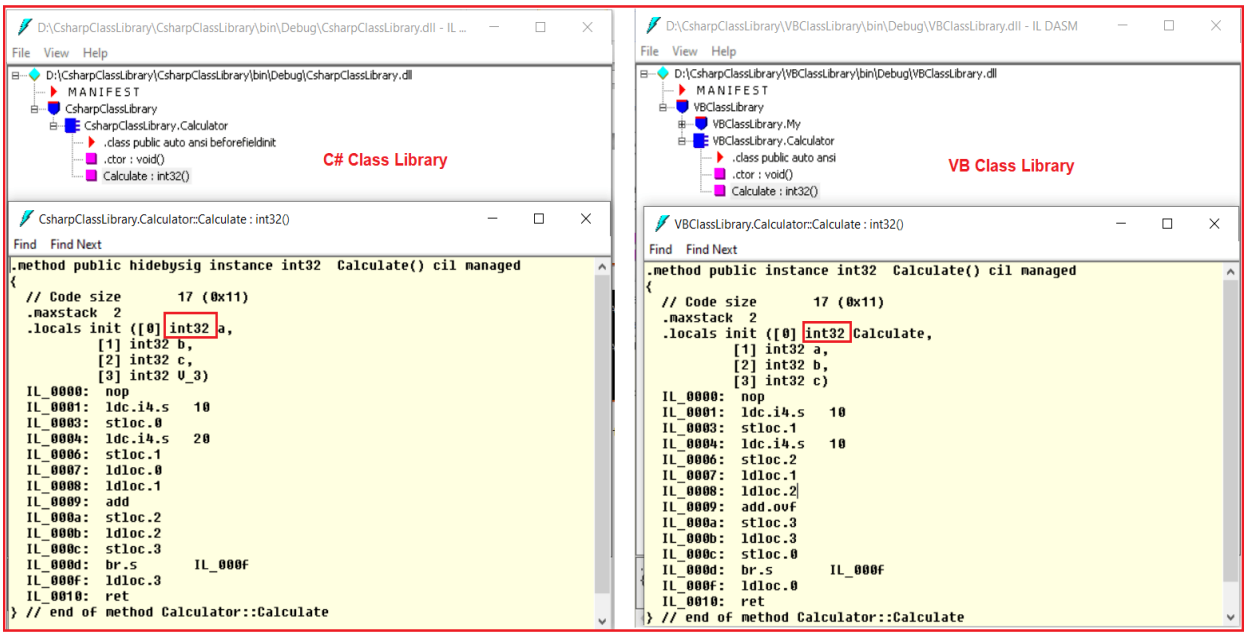
Understanding CTS in .NET:

```
// Creating a C# class library project:
namespace CsharpClassLibrary
{
  public class Calculator
   {
    public int Calculate()
    {
       int a = 10, b = 20;
       int c = a + b;
        return c;
    }
 }}
```

Creating VB Class Library Project:

```
Public Class Calculator
Public Function Calculate() As Integer
Dim a As Integer = 10
Dim b As Integer = 10
Dim c As Integer
c = a + b
Return c
End Function
End Class
```

IL code of both the programs



Understanding CTS

In the world of .NET, type is simply a general term used to refer to a member from the set {class, interface, structure, enumeration, delegate}.

CTS Class Types

.In C#, classes are declared using the class keyword.

```
// A C# class type with 1 method.
class Calc
{ public int Add(int x, int y)
{
return x + y; }
}
```

Table 1-1. *CTS Class Characteristics*

Class Characteristic	Meaning in Life
Is the class sealed or not?	Sealed classes cannot function as a base class to other classes.
Does the class implement any interfaces?	An interface is a collection of abstract members that provide a contract between the object and object user. The CTS allows a class to implement any number of interfaces.
Is the class abstract or concrete?	Abstract classes cannot be directly instantiated, but are intended to define common behaviors for derived types. Concrete classes can be instantiated directly.
What is the visibility of this class?	Each class must be configured with a visibility keyword such as `public` or `internal`. Basically, this controls if the class may be used by external assemblies or only from within the defining assembly.

CTS Interface Types

Interfaces are nothing more than a named collection of abstract member definitions, which may be supported (i.e., implemented) by a given class or structure. In C#, interface types are defined using the interface keyword. By convention, all .NET interfaces begin with a capital letter I

```
// A C# interface type is usually // declared as public, to allow types in other // assemblies to implement their behavior.
public interface IDraw
{
void Draw();
}
```

CTS Structure Types

A structure can be thought of as a lightweight class type having value-based semantics. Typically, structures are best suited for modeling geometric and mathematical data and are created in C# using the struct keyword.

```
// A C# structure type.
struct Point {
 // Structures can contain fields.
public int xPos, yPos;
 // Structures can contain parameterized constructors.
public Point(int x, int y)
{ xPos = x; yPos = y;}
// Structures may define methods.
public void PrintPosition()
 { Console.WriteLine("({0}, {1})", xPos, yPos);
 }
 }
```

CTS Enumeration Types

Enumerations are a handy programming construct that allow you to group name/value pairs.

```
// A C# enumeration type.
enum CharacterType
{
Wizard = 100, Fighter = 200, Thief = 300
 }
```

Intrinsic CTS Data Types

Table 1-2. *The Intrinsic CTS Data Types*

CTS Data Type	VB .NET Keyword	C# Keyword	C++/CLI Keyword
System.Byte	Byte	byte	unsigned char
System.SByte	SByte	sbyte	signed char
System.Int16	Short	short	short
System.Int32	Integer	int	int or long
System.Int64	Long	long	__int64

Common Language Specification:

CLS stands for Common Language Specification and it is a subset of CTS. It defines a set of rules and restrictions that every language must follow which runs under the .NET framework. The languages which follow these set of rules are said to be CLS Compliant. For example, a method with parameter of "unsigned int" type in an object written in C# is not CLS-compliant, just as some languages, like VB.NET, do not support that type it would be ideal to have a baseline to which all .NET-aware languages are expected to conform(follow certain rules).

The CLS is ultimately a set of rules that compiler builders must conform to if they intend their products to function seamlessly within the .NET universe.

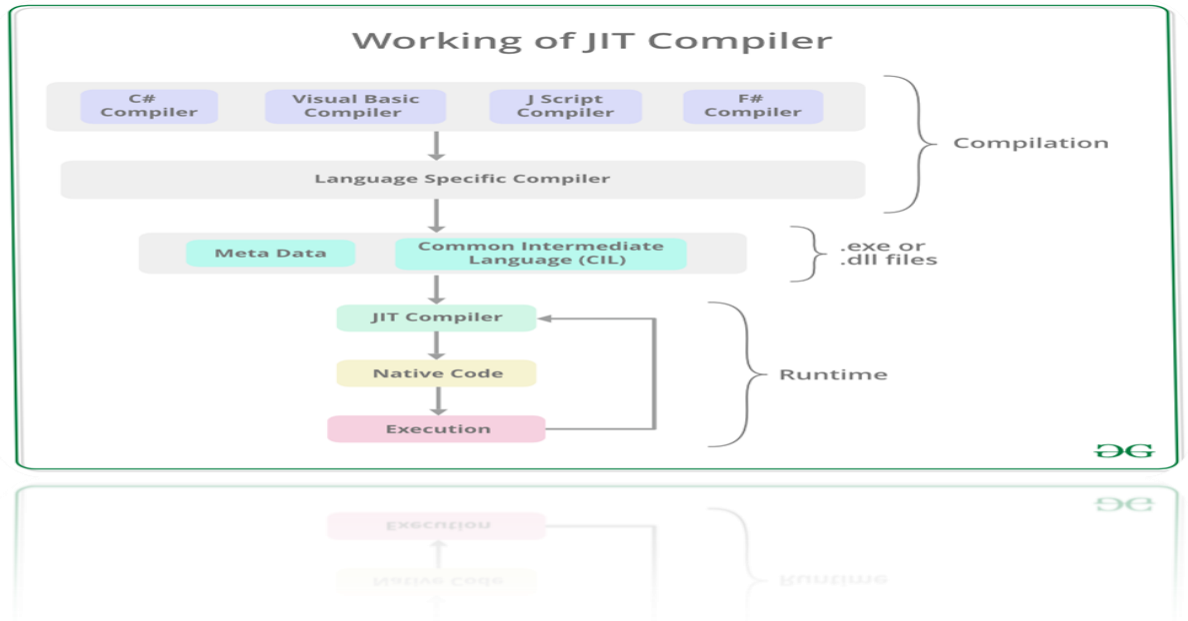
Rule 1: CLS rules apply only to those parts of a type that are exposed outside the defining assembly.
The implementation logic for a member may use any number of non-CLS techniques, as the outside world won't know the difference.

```
class Calc
 { // Exposed unsigned data is not CLS compliant!
public ulong Add(ulong x, ulong y)
 { return x + y; }
}
```

However, if you were to only make use of unsigned data internally in a method as follows,,
class Calc
{
public int Add(int x, int y)
{ // As this ulong variable is only used internally
// we are still CLS compliant.
ulong temp = 0; ... return x + y;
} }

Just-In-Time(JIT) Compiler in .NET:

Just-In-Time compiler(JIT) is a part of Common Language Runtime (CLR) in .NET which is responsible for managing the execution of .NET programs regardless of any .NET programming language. The JIT compiler converts the Microsoft Intermediate Language(MSIL) or Common Intermediate Language(CIL) into the machine code.

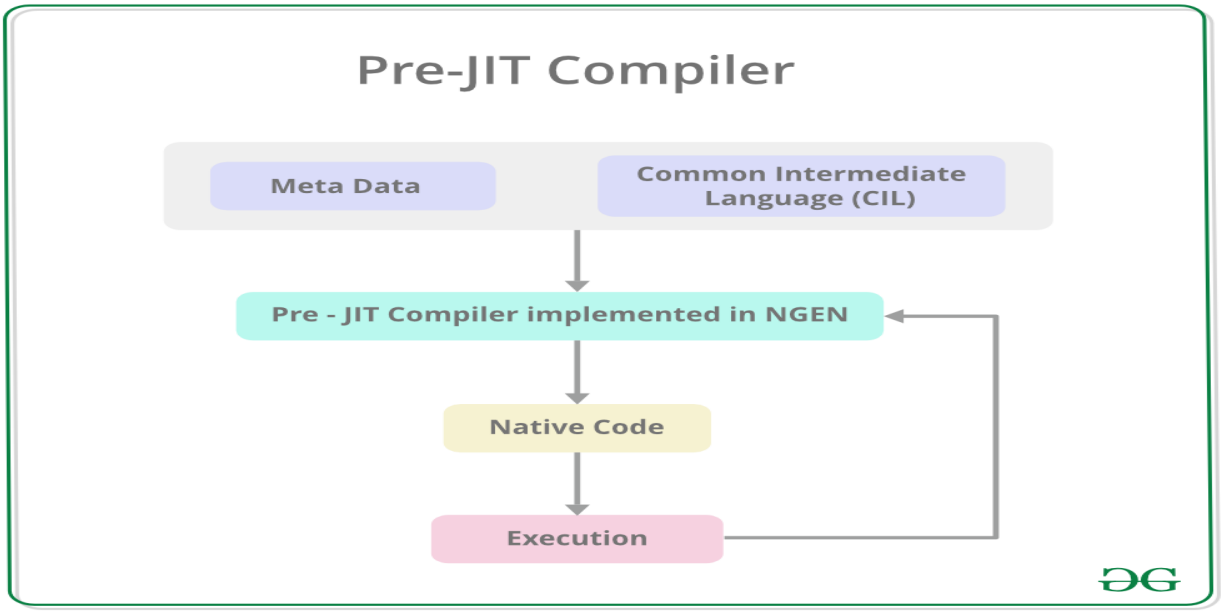


Types of Just-In-Time Compiler:

There are 3 types of JIT compilers which are as follows:
Pre-JIT Compiler
Normal JIT Compiler
Econo JIT Compiler

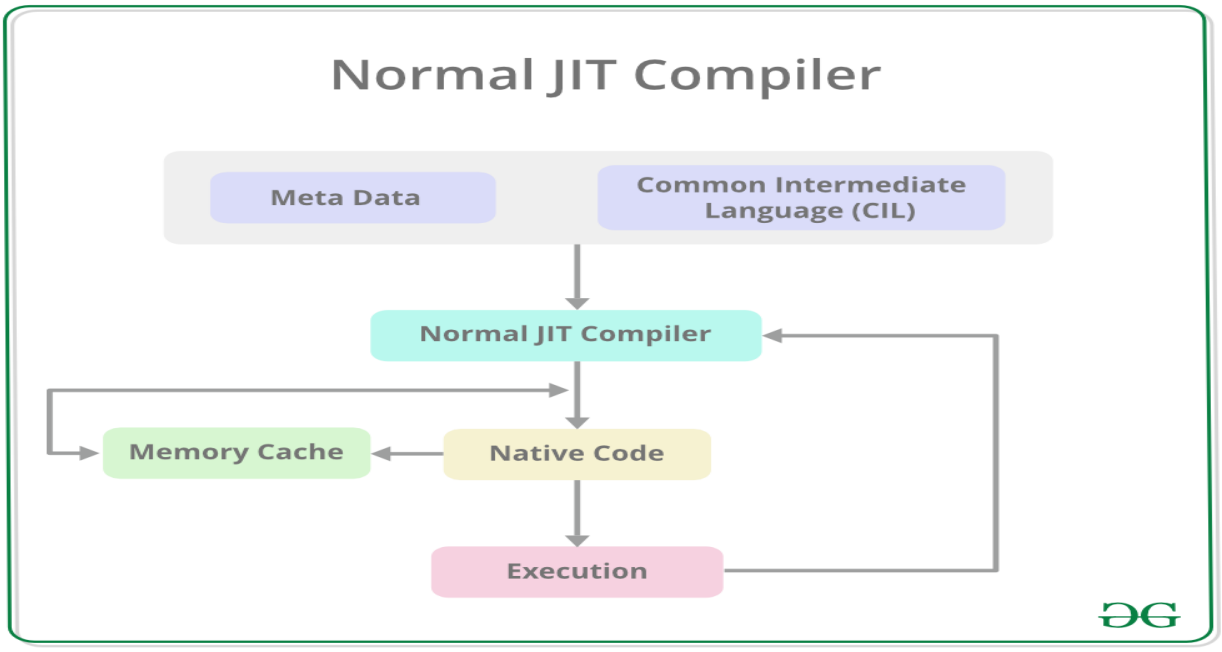
Pre-JIT Compiler

Pre-JIT Compiler: All the source code is compiled into the machine code at the same time in a single compilation cycle using the Pre-JIT Compiler.



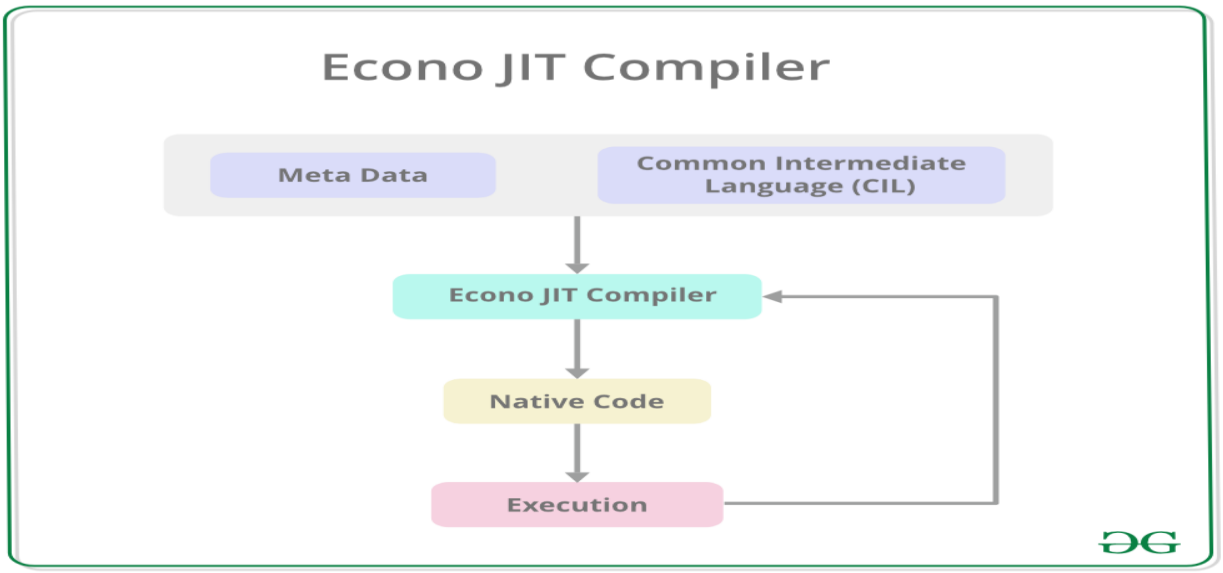
Normal JIT Compiler

Normal JIT Compiler: The source code methods that are required at run-time are compiled into machine code the first time they are called by the Normal JIT Compiler. After that, they are stored in the cache and used whenever they are called again.



Econo JIT Compiler

Econo JIT Compiler: The source code methods that are required at run-time are compiled into machine code by the Econo JIT Compiler. After these methods are not required anymore, they are removed.



UNIT II
Developing Console Application

Command Line arguments:

The arguments which are passed by the user or programmer to the Main() method is termed as Command-Line Arguments.

Main() method is the entry point of execution of a program. Main() method accepts array of strings. But it never accepts parameters from any other method in the program.

In C# the command line arguments are passed to the Main() methods by stating as follows:

static void Main(string[] args)

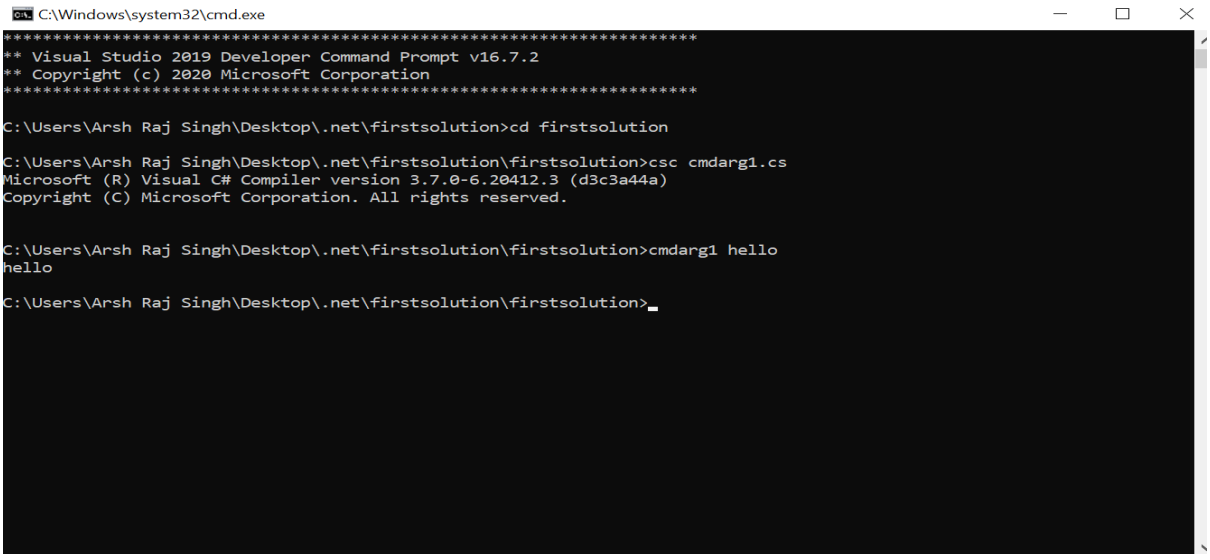
or

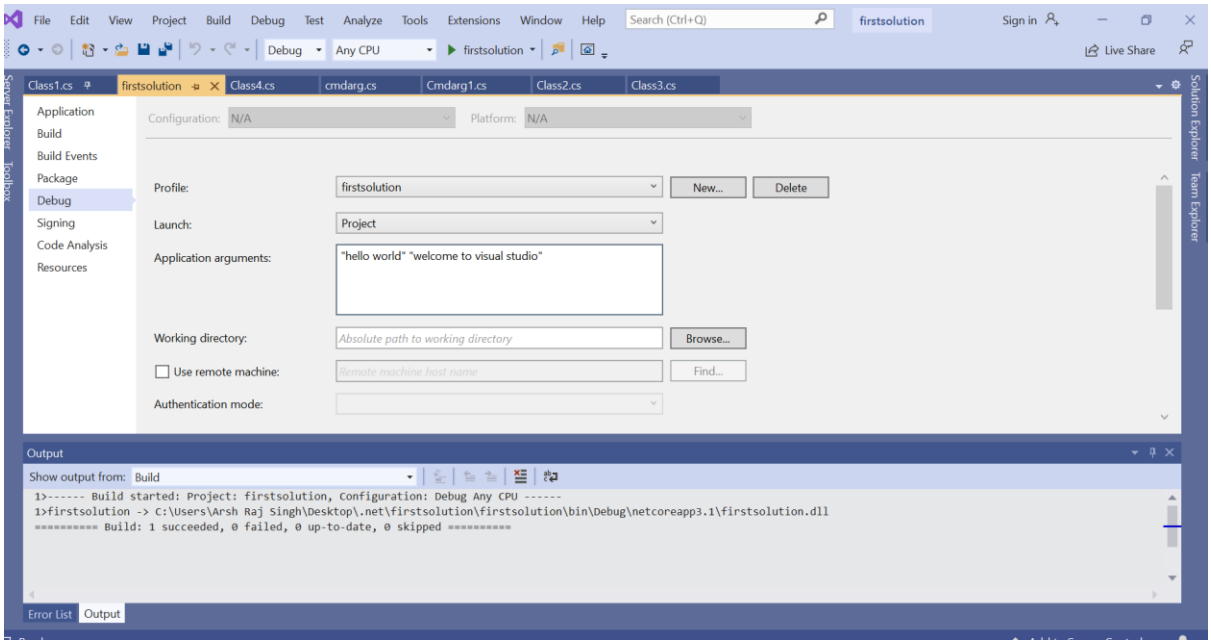
static int Main(string[] args)

EX:

```
using System;
namespacefirstsolution
{
classCmdarg1
   {
staticvoidMain(String[] args)
      {
Console.WriteLine(args[0]);
      }
   }
}
```

The above program can be executed in the command prompt

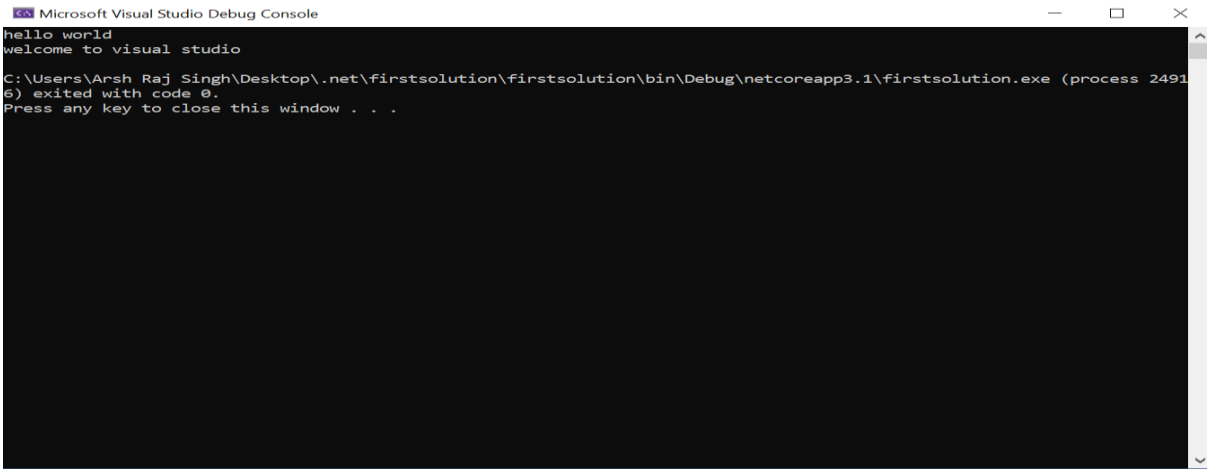




To specify the command line arguments in Visual Studio , double-click the Properties icon from Solution Explorer and select the Debug tab on the left side. From there, specify values using the command-line arguments text box.

/* displaying multiple command line arguments*/

```
namespacefirstsolution
{
classCmdarg2
   {
staticvoidMain(String[] args)
         {
for (inti = 0; i<args.Length; i++)
Console.WriteLine(args[i]);
         }
      }
   }
```



C# | foreach Loop

foreach loop is used to iterate over the elements of the collection. The collection may be an array or a list. It executes for each element present in the array.

- It is necessary to enclose the statements of foreach loop in curly braces {}.
- Instead of declaring and initializing a loop counter variable, you declare a variable that is the same type as the base type of the array, followed by a colon, which is then followed by the array name.
- In the loop body, you can use the loop variable you created rather than using an indexed array element.

Syntax:

foreach(data_typevar_name in collection_variable)

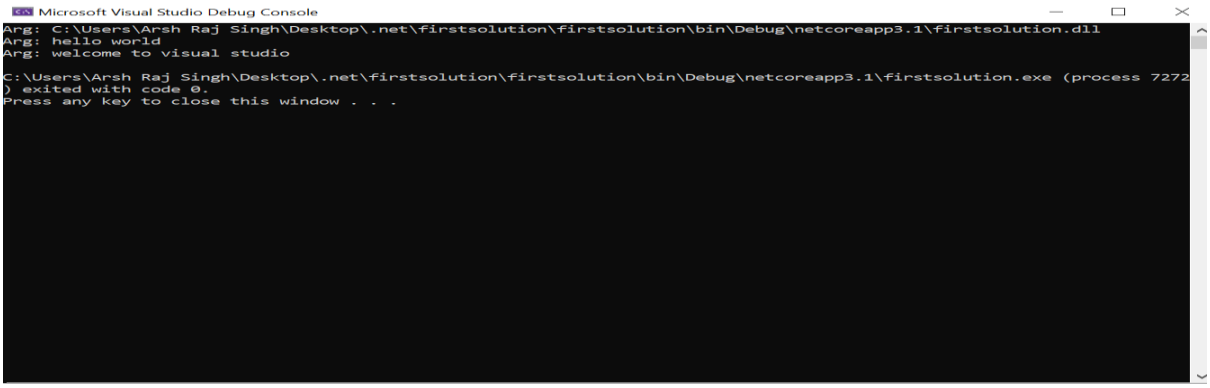
{

// statements to be executed

}

Finally, you are also able to access command-line arguments using the static GetCommandLineArgs() method of the System. Environment type. The return value of this method is an array of strings. The first index identifies the name of the application itself, while the remaining elements in the array contain the individual command-line arguments. Note that when using this approach, it is no longer necessary to define Main() as taking a string array as the input parameter, although there is no harm in doing so.

```
using System;
namespace firstsolution
{
   class cmdarg
   {
      static void Main()
      {
string[] theArgs = Environment.GetCommandLineArgs();
         foreach (string arg in theArgs)
Console.WriteLine("Arg: {0}", arg);
      }
   }
}
```



Different valid forms of Main:

By default, Visual Studio will generate a Main() method that has a void return value and an array of string types as the single input parameter. This is not the only possible form of Main(), however. It is permissible to construct your application's entry point using any of the following signatures (assuming it

is contained within a C# class or structure definition):

```
// int return type, array of strings as the parameter.
static int Main(string[] args)
{
 // Must return a value before exiting!
 return 0;
}
// No return type, no parameters.
static void Main()
{
}
// int return type, no parameters.
static int Main()
{
 // Must return a value before exiting!
 return 0;
}
```

C# Data Types:

A data type specifies the size and type of variable values. It is important to use the correct data type for the corresponding variable; to avoid errors, to save time and memory, but it will also make your code more maintainable and readable. The most common data types are:

Table 3-4. *The Intrinsic Data Types of C#*

C# Shorthand	CLS Compliant?	System Type	Range	Meaning in Life
bool	Yes	System.Boolean	true or false	Represents truth or falsity
sbyte	No	System.SByte	–128 to 127	Signed 8-bit number
byte	Yes	System.Byte	0 to 255	Unsigned 8-bit number
short	Yes	System.Int16	–32,768 to 32,767	Signed 16-bit number
ushort	No	System.UInt16	0 to 65,535	Unsigned 16-bit number
int	Yes	System.Int32	–2,147,483,648 to 2,147,483,647	Signed 32-bit number
uint	No	System.UInt32	0 to 4,294,967,295	Unsigned 32-bit number
long	Yes	System.Int64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit number
ulong	No	System.UInt64	0 to 18,446,744,073,709,551,615	Unsigned 64-bit number
char	Yes	System.Char	U+0000 to U+ffff	Single 16-bit Unicode character
float	Yes	System.Single	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	32-bit floating-point number
double	Yes	System.Double	$\pm 5.0 \times 10^{324}$ to $\pm 1.7 \times 10^{308}$	64-bit floating-point number
decimal	Yes	System.Decimal	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$	96-bit signed number
string	Yes	System.String	Limited by system memory	Represents a set of Unicode characters
Object	Yes	System.Object	Can store any data type in an object variable	The base class of all types in the .NET universe

Intrinsic Data Types and the new Operator

All intrinsic data types support what is known as a default constructor This featureallows you to create a variable using the new keyword, which automatically sets the variable to its defaultvalue.

• bool variables are set to false.

• Numeric data is set to 0 (or 0.0 in the case of floating-point data types).

• char variables are set to a single empty character.

• BigInteger variables are set to 0.

• DateTime variables are set to 1/1/0001 12:00:00 AM.

• Object references (including strings) are set to null.

```
static void NewingDataTypes()
{
 Console.WriteLine("=> Using new to create variables:");
 bool b = new bool(); // Set to false.
 int i = new int(); // Set to 0.
 double d = new double(); // Set to 0.
DateTime dt = new DateTime(); // Set to 1/1/0001 12:00:00 AM
}
```

The Data Type Class Hierarchy

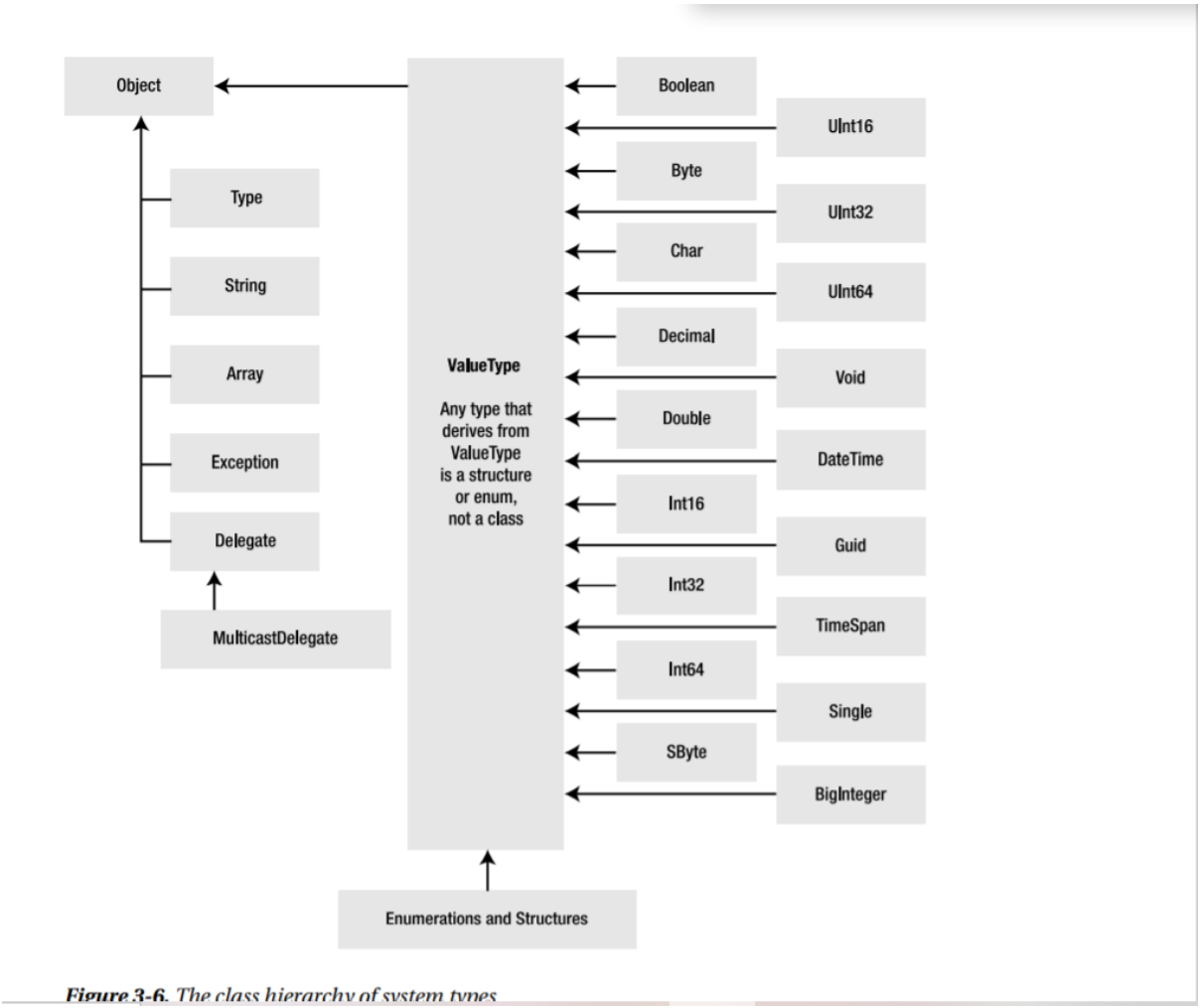


Figure 3-6. The class hierarchy of system types

Note that many numerical data types derive from a class named System.ValueType. Descendents of Value Type are automatically allocated on the stack and therefore have a very predictable lifetime and are quite efficient. On the other hand, types that do not have System.ValueType in their inheritance chain (such as System.Type, System.String, System.Array, System.Exception, and System.Delegate) are not allocated on the stack, but on the garbage-collected heap.

C# keyword (such as int) is simply shorthand notation for the corresponding system type (in this case, System.Int32), the following is perfectly legal syntax, given that System.Int32 (the C# int) eventually derives from System.Object and therefore can invoke any of its public members, as illustrated by this additional helper function:

```
static void ObjectFunctionality()

{

Console.WriteLine("=>System.Object Functionality:");
```

```
// A C# int is really a shorthand for System.Int32.
// which inherits the following members from System.Object.
Console.WriteLine("12.GetHashCode() = {0}", 12.GetHashCode());
Console.WriteLine("12.Equals(23) = {0}", 12.Equals(23));
Console.WriteLine("12.ToString() = {0}", 12.ToString());
Console.WriteLine("12.GetType() = {0}", 12.GetType());
Console.WriteLine();
}
```

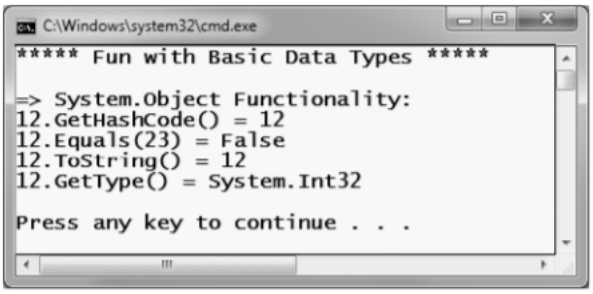


Figure 3-7. *All types (even numerical data) extend System.Object*

Members of Numerical Data Types

```
static void DataTypeFunctionality()
{
Console.WriteLine("=> Data type Functionality:");
Console.WriteLine("Max of int: {0}", int.MaxValue);
Console.WriteLine("Min of int: {0}", int.MinValue);
Console.WriteLine("Max of double: {0}", double.MaxValue);
Console.WriteLine("Min of double: {0}", double.MinValue);
}
```

Members of System.Char

C# textual data is represented by the string and char keywords, which are simple shorthand notations for System.String and System.Char, both of which are Unicode under the hood. As you may already know, a string represents a contiguous set of characters (e.g., "Hello"), while the char can represent a single slot in a string (e.g., 'H').

The System.Char type provides you with a great deal of functionality beyond the ability to hold a single point of character data. Using the static methods of System.Char, you are able to determine whether a given character is numerical, alphabetical, a point of punctuation, or whatnot. Consider the following method:

```
static void CharFunctionality()
{
 Console.WriteLine("=> char type Functionality:");
 char myChar = 'a';
Console.WriteLine("char.IsDigit('a'): {0}", char.IsDigit(myChar));
Console.WriteLine("char.IsLetter('a'): {0}", char.IsLetter(myChar));
Console.WriteLine("char.IsWhiteSpace('Hello There', 5): {0}",
char.IsWhiteSpace("Hello There", 5));
Console.WriteLine("char.IsWhiteSpace('Hello There', 6): {0}",
char.IsWhiteSpace("Hello There", 6));
Console.WriteLine("char.IsPunctuation('?'): {0}",
char.IsPunctuation('?'));
 Console.WriteLine();
}
```

Working with String Data:

System.String provides a number of methods you would expect from such a utility class, including methods that return the length of the character data, find substrings within the current string, and convert to and from uppercase/lowercase. Table 3-5 lists some (but by no means all) of the interesting members.

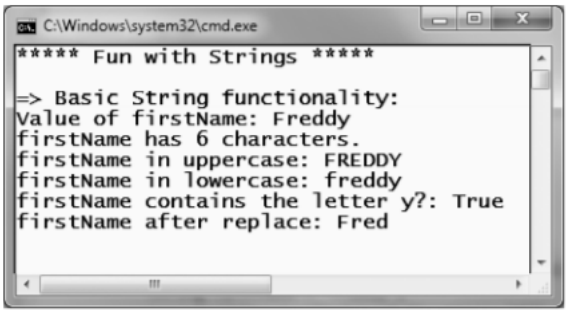
Table 3-5. *Select Members of System.String*

String Member	Meaning in Life
`Length`	This property returns the length of the current string.
`Compare()`	This static method compares two strings.
`Contains()`	This method determines whether a string contains a specific substring.
`Equals()`	This method tests whether two string objects contain identical character data.
`Format()`	This static method formats a string using other primitives (e.g., numerical data, other strings) and the `{0}` notation examined earlier in this chapter.
`Insert()`	This method inserts a string within a given string.
`PadLeft()` `PadRight()`	These methods are used to pad a string with some characters.

Table 3-5. *Select Members of System.String (continued)*

String Member	Meaning in Life
`Remove()` `Replace()`	Use these methods to receive a copy of a string with modifications (characters removed or replaced).
`Split()`	This method returns a `String` array containing the substrings in this instance that are delimited by elements of a specified `char` array or `string` array.
`Trim()`	This method removes all occurrences of a set of specified characters from the beginning and end of the current string.
`ToUpper()` `ToLower()`	These methods create a copy of the current string in uppercase or lowercase format, respectively.

```
static void BasicStringFunctionality()
{
 Console.WriteLine("=> Basic String functionality:");
 string firstName = "Freddy";
Console.WriteLine("Value of firstName: {0}", firstName);
Console.WriteLine("firstName has {0} characters.", firstName.Length);
Console.WriteLine("firstName in uppercase: {0}", firstName.ToUpper());
Console.WriteLine("firstName in lowercase: {0}", firstName.ToLower());
Console.WriteLine("firstName contains the letter y?: {0}",
firstName.Contains("y"));
Console.WriteLine("firstName after replace: {0}", firstName.Replace("dy", ""));
 Console.WriteLine();
}
```



String Concatenation

string variables can be connected together to build larger strings via the C# + operator. As you mayknow, this technique is formally termed string concatenation. Consider the following new helper

function:

```
static void StringConcatenation()
{
 Console.WriteLine("=> String concatenation:");
 string s1 = "Programming with ";
 string s2 = "C#";
 string s3 = s1 + s2;
 Console.WriteLine(s3);
 Console.WriteLine();
}
```

You may be interested to know that the C# + symbol is processed by the compiler to emit a call to the static String.Concat() method.

Given this, it is possible to perform string concatenation by calling String.Concat() directly (although you really have not gained anything by doing so—in fact, you have incurred additional keystrokes!).

```
static void StringConcatenation()
{
 Console.WriteLine("=> String concatenation:");
 string s1 = "Programming with ";
 string s2 = " C#";
 string s3 = String.Concat(s1, s2);
 Console.WriteLine(s3);
 Console.WriteLine();
}
```

Strings and Equality

The C# equality operators perform a case-sensitive, character-by-character equality test on string objects. Therefore, "Hello!" is not equal to "HELLO!", which is different from "hello!". Also, keeping the connection between string and System.String in mind, notice that we are able to test for equality using the Equals() method of String as well as the baked-in equality operators

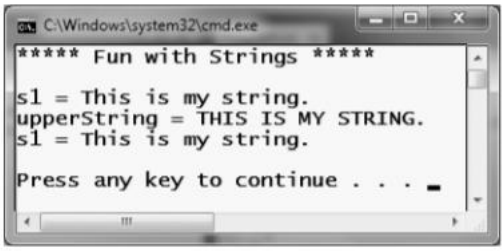
```
static void StringEquality()
{
 Console.WriteLine("=> String equality:");
 string s1 = "Hello!";
 string s2 = "Yo!";
 Console.WriteLine("s1 = {0}", s1);
 Console.WriteLine("s2 = {0}", s2);
 Console.WriteLine();
 // Test these strings for equality.
 Console.WriteLine("s1 == s2: {0}", s1 == s2);
```

```
 Console.WriteLine("s1 == Hello!: {0}", s1 == "Hello!");
 Console.WriteLine("s1 == HELLO!: {0}", s1 == "HELLO!");
 Console.WriteLine("s1 == hello!: {0}", s1 == "hello!");
 Console.WriteLine("s1.Equals(s2): {0}", s1.Equals(s2));
Console.WriteLine("Yo.Equals(s2): {0}", "Yo!".Equals(s2));
 Console.WriteLine();
}
```

Strings Are Immutable

One of the interesting aspects of System.String is that once you assign a string object with its initial value, the character data cannot be changed.

```
static void StringsAreImmutable()
{
 // Set initial string value.
 string s1 = "This is my string.";
 Console.WriteLine("s1 = {0}", s1);
 // Uppercase s1?
 string upperString = s1.ToUpper();
Console.WriteLine("upperString = {0}", upperString);
 // Nope! s1 is in the same format!
 Console.WriteLine("s1 = {0}", s1);
}
```



C# Type Casting

type casting is when you assign a value of one data type to another type.

In C#, there are two types of casting:

Implicit Casting (automatically) - converting a smaller type to a larger type size

char -> int -> long -> float -> double

Explicit Casting (manually) - converting a larger type to a smaller size type

double -> float -> long -> int -> char

Implicit Casting

Implicit casting is done automatically when passing a smaller size type to a larger size type:

Example

```
int myInt = 9;
double myDouble = myInt;       // Automatic casting: int to double
Console.WriteLine(myInt);      // Outputs 9
Console.WriteLine(myDouble);   // Outputs 9
```

Explicit Casting

Explicit casting must be done manually by placing the type in parentheses in front of the value:

```
double myDouble = 9.78;
int myInt = (int) myDouble;    // Manual casting: double to int
Console.WriteLine(myDouble);   // Outputs 9.78
Console.WriteLine(myInt);      // Outputs 9
```

Type Conversion Methods

It is also possible to convert data types explicitly by using built-in methods, such as Convert.ToBoolean, Convert.ToDouble, Convert.ToString, Convert.ToInt32 (int) and Convert.ToInt64 (long):

Example

```
int myInt = 10;
double myDouble = 5.25;
bool myBool = true;
Console.WriteLine(Convert.ToString(myInt));    // convert int to string
Console.WriteLine(Convert.ToDouble(myInt));    // convert int to double
Console.WriteLine(Convert.ToInt32(myDouble));  // convert double to int
Console.WriteLine(Convert.ToString(myBool));   // convert bool to string
```

This type of conversion is required when working with the user input.

Get User Input

You have already learned that Console.WriteLine() is used to output (print) values.
Now we will use Console.ReadLine() to get user input.
In the following example, the user can input his or hers username, which is stored in the variable userName. Then we print the value of userName:

Example

```
// Type your username and press enter
Console.WriteLine("Enter username:");
// Create a string variable and get user input from the keyboard and store it in the variable
string userName = Console.ReadLine();
// Print the value of the variable (userName), which will display the input value
Console.WriteLine("Username is: " + userName)
```

User Input and Numbers

The Console.ReadLine() method returns a string. Therefore, you cannot get information from another data type, such as int. The following program will cause an error:

Example

```
Console.WriteLine("Enter your age:");
int age = Console.ReadLine();
Console.WriteLine("Your age is: "+ age);
```

The error message will be something like this:
Cannot implicitly convert type 'string' to 'int'
Luckily, for you, you just learned, that you can convert any type explicitly, by using one of the Convert. To methods:

Example

```
Console.WriteLine("Enter your age:");
int age = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("Your age is: "+ age);
```

C# structures:

The C# structures have the following features −

- Structures can have methods, fields, indexers, properties, operator methods, and events.
- Structures can have defined constructors, but not destructors. However, you cannot define a default constructor for a structure. The default constructor is automatically defined and cannot be changed.
- Unlike classes, structures cannot inherit other structures or classes.
- Structures cannot be used as a base for other structures or classes.
- A structure can implement one or more interfaces.

- Structure members cannot be specified as abstract, virtual, or protected.
- When you create a struct object using the **New** operator, it gets created and the appropriate constructor is called. Unlike classes, structs can be instantiated without using the New operator.
- If the New operator is not used, the fields remain unassigned and the object cannot be used until all the fields are initialized.

Class versus Structure

Classes and Structures have the following basic differences −

- classes are reference types and structs are value types
- structures do not support inheritance
- structures cannot have default constructor

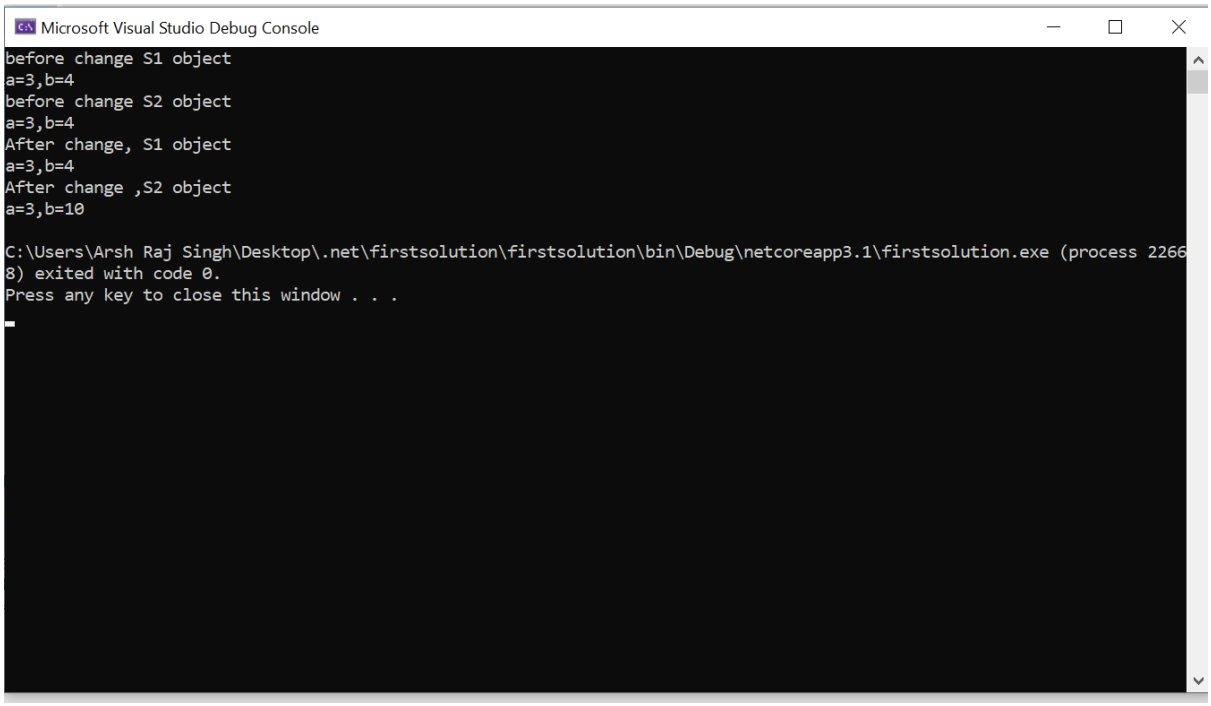
Example: Parameterized Constructor in Struct

```
using System;

public class Program
{
        public static void Main()
        {
                Coordinate point = new Coordinate(10, 20);
                Console.WriteLine(point.x); //output: 10
                Console.WriteLine(point.y); //output: 20
        }
}

struct Coordinate
{
   public int x;
   public int y;

   public Coordinate(int x, int y)
   {
this.x = x;
this.y = y;
   }
}
```

Value Types, References Types, and the Assignment Operator

1. Assigning Value types

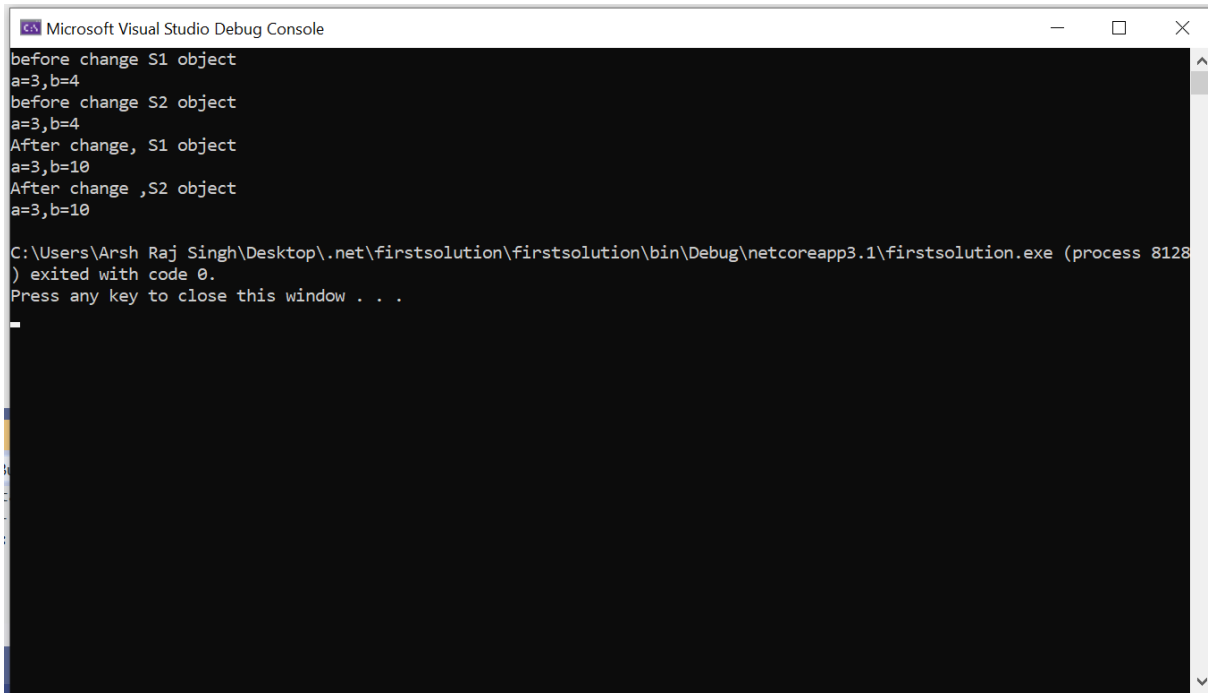
```
using System;

namespacefirstsolution
{
structsample1
   {
publicint a, b;
publicsample1(intx,int y)
      {
         a = x;b = y;
      }
publicvoiddisplay()
      {
Console.WriteLine("a={0},b={1}", a, b);
      }
   }
classstruct2cs
   {
staticvoidMain()
      {
         sample1 s1 = new sample1(3, 4);
         sample1 s2;
         s2 = s1;
Console.WriteLine("before change,S1 object");
         s1.display();
Console.WriteLine("before change, S2 object");
         s2.display();
         s2.b = 10;
Console.WriteLine("After change, S1 object");
         s1.display();
Console.WriteLine("After change ,S2 object");
         s2.display();
      }
   }
}
```



2. Assigning reference types

```
using System;

namespacefirstsolution
{
classsample2
   {
publicint a, b;
publicsample2(int x, int y)
      {
         a = x; b = y;
      }
publicvoiddisplay()
      {
Console.WriteLine("a={0},b={1}", a, b);
      }
   }
classClassref
   {
staticvoidMain()
      {
         sample2 s1 = new sample2(3, 4);
         sample2 s2;
         s2 = s1;
Console.WriteLine("before change S1 object");
         s1.display();
```

```
Console.WriteLine("before change S2 object");
        s2.display();
        s2.b = 10;
Console.WriteLine("After change, S1 object");
        s1.display();

Console.WriteLine("After change ,S2 object");
        s2.display();
    }
  }
        }
```



Value Types Containing Reference Types

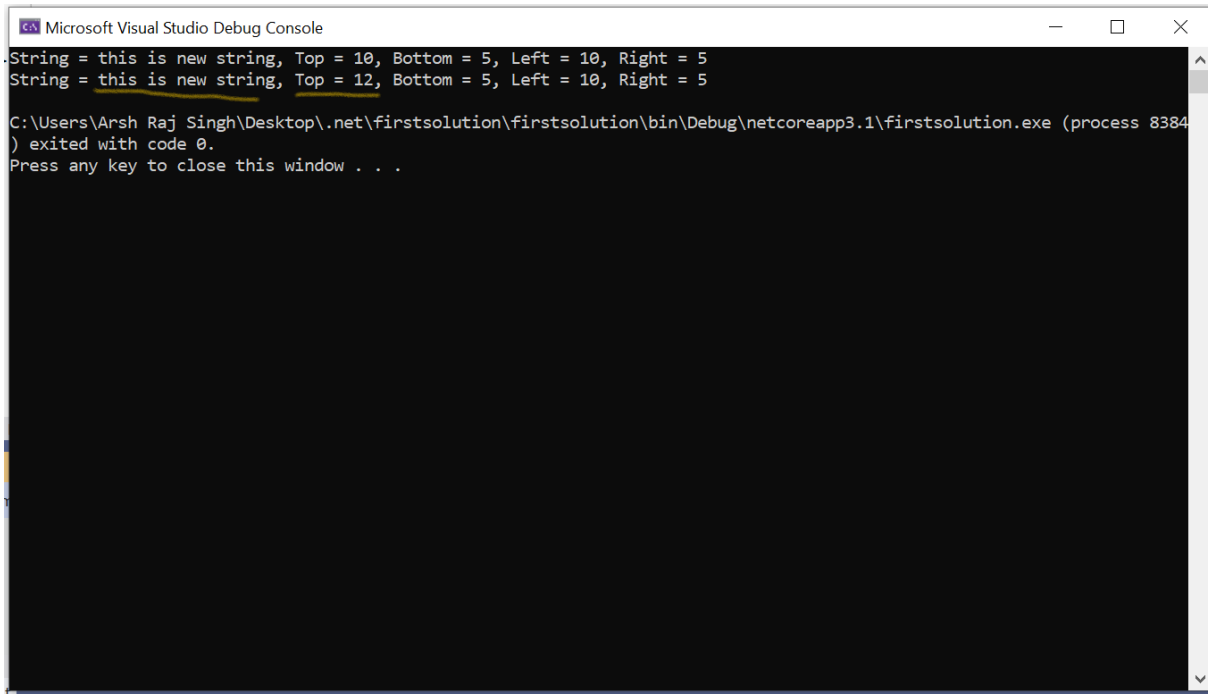
```
using System;
namespace firstsolution
{
Class ShapeInfo
  {
Public string infoString;
Public ShapeInfo(string info)
      {
infoString = info;
      }
  }
```

/*Now assume that you want to contain a variable of this class type within a value type named Rectangle. To allow the caller to set the value of the inner ShapeInfo member

variable, you also provide a custom constructor. Here is the complete definition of the Rectangle type:/*

```
Struct Rectangle
   {
// The Rectangle structure contains a reference type member.
publicShapeInfo rectInfo;
public intrectTop, rectLeft, rectBottom, rectRight;
public Rectangle(string info, int top, int left, int bottom, int right)
      {
rectInfo = newShapeInfo(info);
rectTop = top; rectBottom = bottom;
rectLeft = left; rectRight = right;
      }
Public void Display()
      {
Console.WriteLine("String = {0}, Top = {1}, Bottom = {2}, " +
"Left = {3}, Right = {4}",
rectInfo.infoString, rectTop, rectBottom, rectLeft, rectRight);
      }
   }
classVR2
   {
Static void Main()
      {
          Rectangle r1 = newRectangle("this is rectangle",10,10,5,5);
         Rectangle r2 = r1;
         r2.rectInfo.infoString = "this is new string";
         r2.rectTop = 12;
         r1.Display();
         r2.Display();
      }
   }
}
```

As you can see, when you change the value of the informational string using the r2 reference, the r1 reference displays the same value. By default, when a value type contains other reference types, assignment results in a copy of the references.

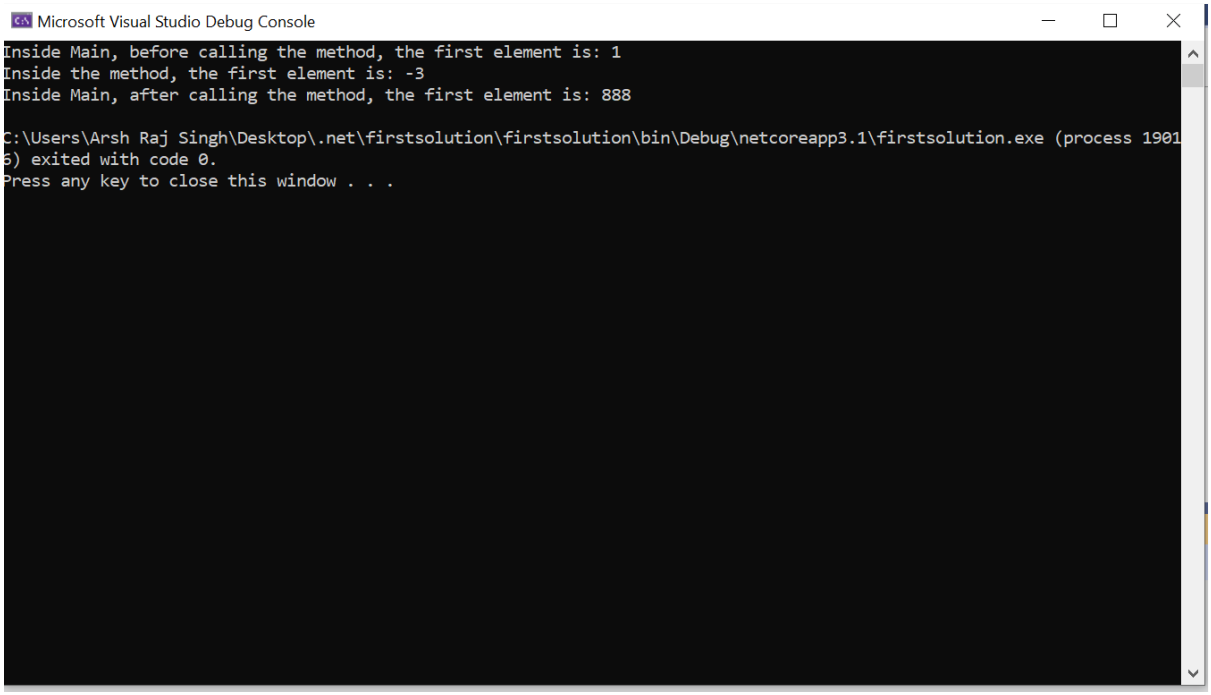


Passing Reference Types by Value

The following example demonstrates passing a reference-type parameter, arr, by value, to a method, Change. Because the parameter is a reference to arr, it is possible to change the values of the array elements. However, the attempt to reassign the parameter to a different memory location only works inside the method and does not affect the original variable, arr.

classPassingRefByVal

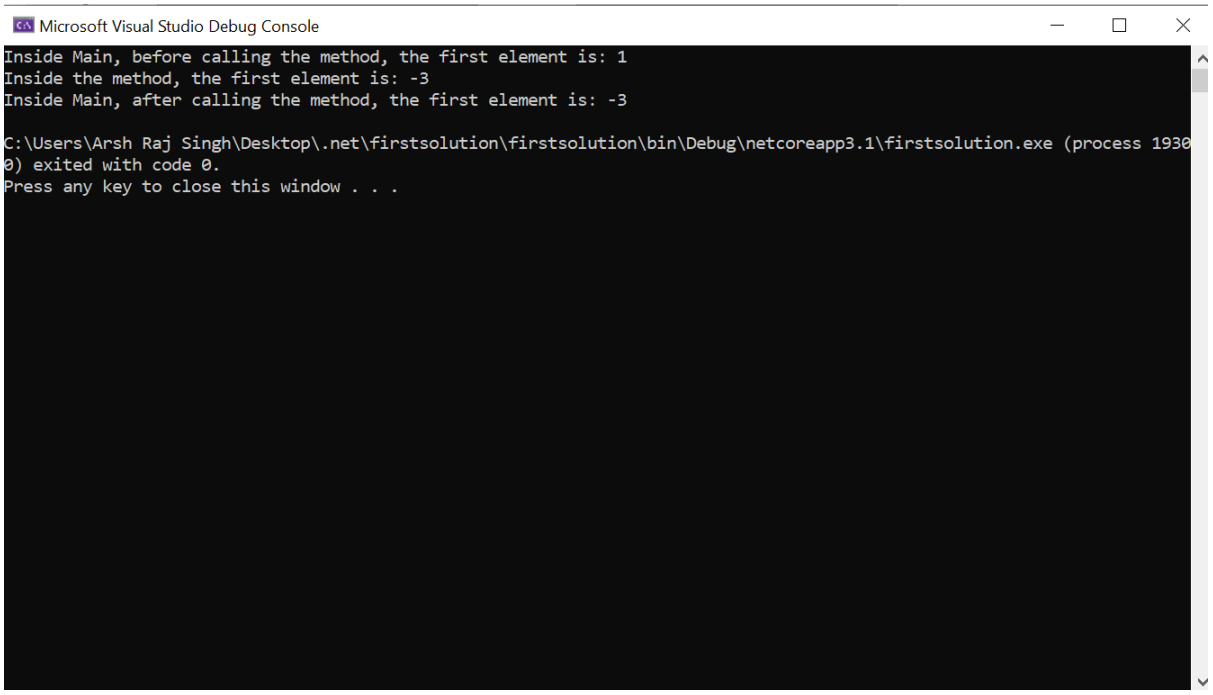
```
{
staticvoidChange(int[] pArray)
   {
      pArray[0] = 888;  // This change affects the original element.
      pArray = newint[5] {-3, -1, -2, -3, -4};   // This change is local.
      System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
   }
staticvoidMain()
   {
int[] arr = {1, 4, 5};
      System.Console.WriteLine("Inside Main, before calling the method, the first element is:
{0}", arr [0]);
      Change(arr);
      System.Console.WriteLine("Inside Main, after calling the method, the first element is:
{0}", arr [0]);
   }
}
```



Passing Reference Types by Reference

The following example is the same as the previous example, except that the ref keyword is added to the method header and call. Any changes that take place in the method affect the original variable in the calling program.

```
Class PassingRefByRef
{
Static void Change(refint[] pArray)
   {
// Both of the following changes will affect the original variables:
      pArray[0] = 888;
      pArray = newint[5] {-3, -1, -2, -3, -4};
      System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
   }
Static void Main()
   {
int[] arr = {1, 4, 5};
      System.Console.WriteLine("Inside Main, before calling the method, the first element is:
{0}", arr[0]);
Change(ref arr);
      System.Console.WriteLine("Inside Main, after calling the method, the first element is:
{0}", arr[0]);
   }
}
```



Stack and Heap memory:

There are two places the .NET framework stores items in memory as your code executes.

1.Stack

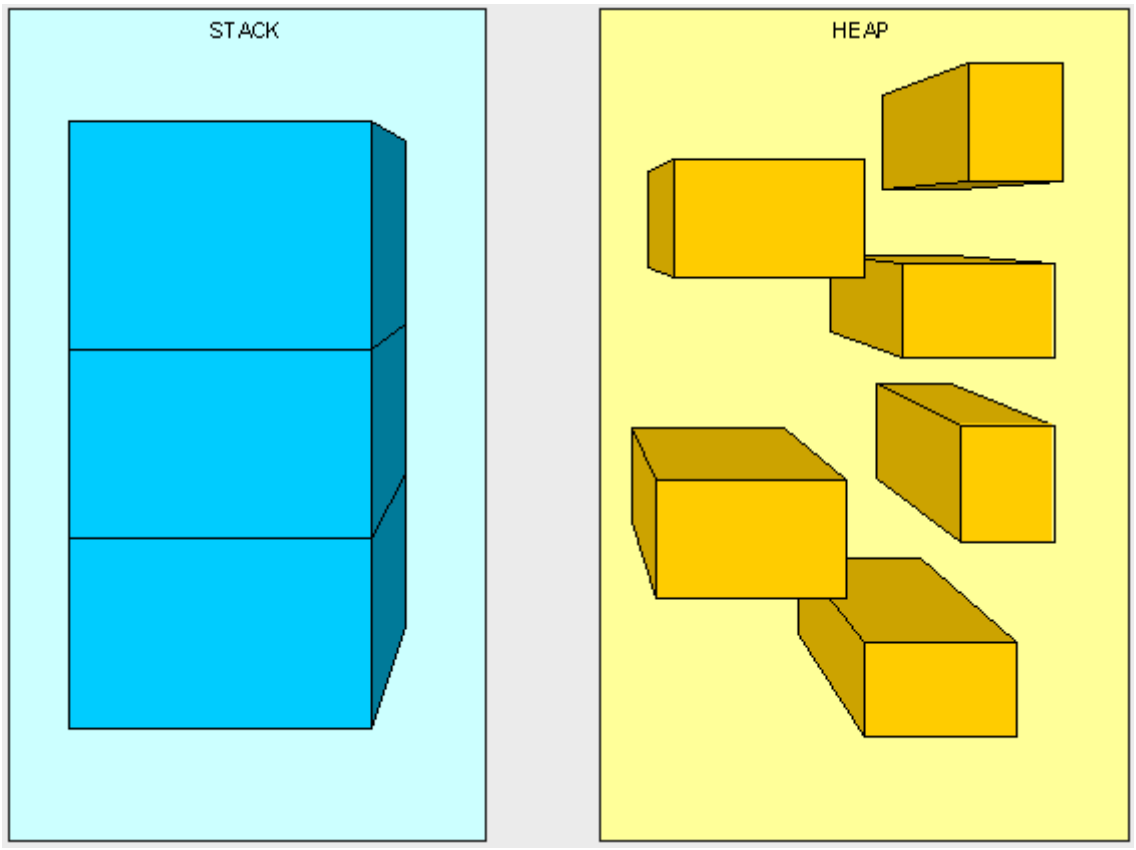
2.Heap

The Stack is more or less responsible for keeping track of what's executing in our code (or what's been "called").

The Heap is more or less responsible for keeping track of our objects.

Think of the Stack as a series of boxes stacked one on top of the next. We keep track of what's going on in our application by stacking another box on top every time we call a method.

We can only use what's in the top box on the stack. When we're done with the top box (the method is done executing) we throw it away and proceed to use the stuff in the previous box on the top of the stack.



The Heap is similar except that its purpose is to hold information (not keep track of execution most of the time) so anything in our Heap can be accessed at any time.

With the Heap, there are no constraints as to what can be accessed like in the stack.

The Stack is self-maintaining, meaning that it basically takes care of its own memory management. When the top box is no longer used, it's thrown out.

The Heap, on the other hand, has to worry about Garbage collection (GC) - which deals with how to keep the Heap clean .

What goes on the Stack and Heap?

We have three main types of things we'll be putting in the Stack and Heap as our code is executing: Value Types, Reference Typesand Pointers.

Value Types

In C#, all the "things" declared with the following list of type declarations are Value types

- bool
- byte
- char
- decimal
- double

- enum
- float
- int
- long
- sbyte
- short
- struct
- uint
- ulong
- ushort

Reference Types

All the "things" declared with the types in this list are Reference types

- class
- interface
- delegate
- object
- string

Pointers

The third type of "thing" to be put in our memory management scheme is a Reference to a Type. A Reference is often referred to as a Pointer.
We don't explicitly use Pointers, they are managed by the Common Language Runtime (CLR).

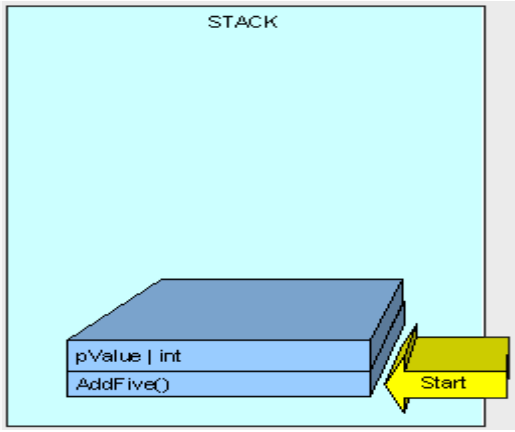
How is it decided what goes where?

Here are our two golden rules:

1. A Reference Type always goes on the Heap.
2. Value Types and Pointers always go where they were declared.

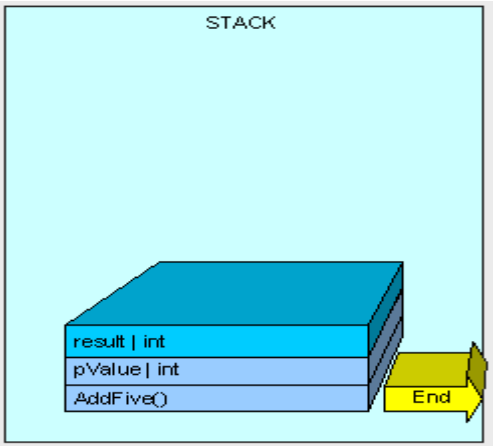
Take the following method.

```
1. public int AddFive(int pValue)
2. {
3.     int result;
4.     result = pValue + 5;
5.     return result;
6. }
```

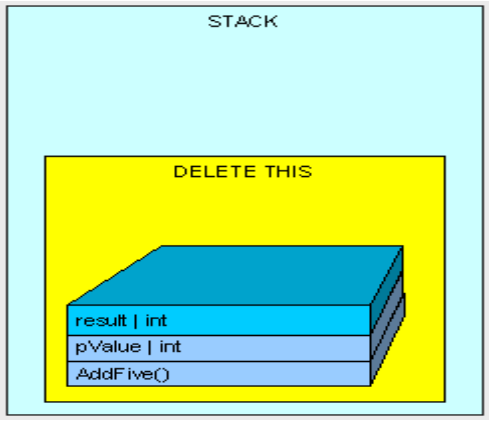


Once we start executing the method, the method's parameters are placed on the stack

As the method executes, we need some memory for the "result" variable and it is allocated on the stack.



And all memory allocated on the stack is cleaned up by moving a pointer to the available memory address where AddFive() started and we go down to the previous method on the stack (not seen here).



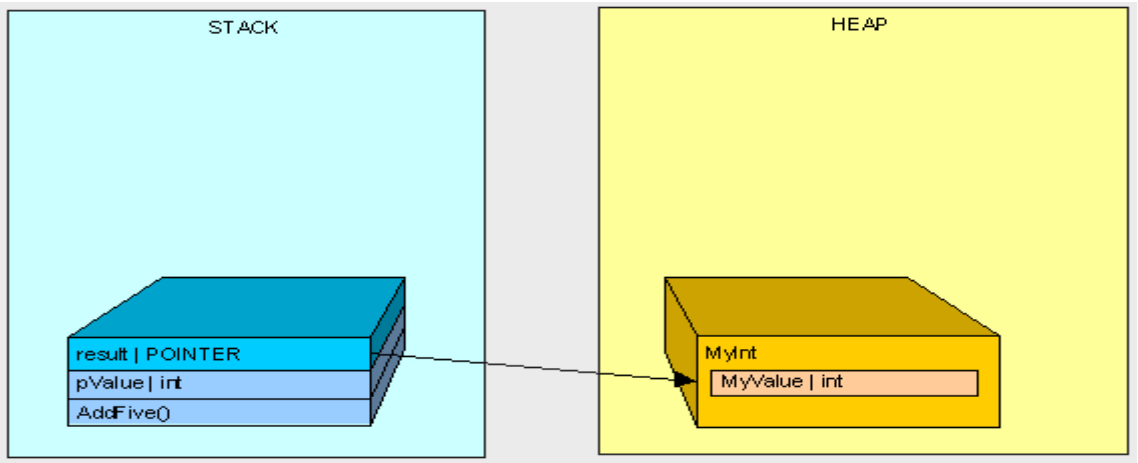
Now, Value Types are also sometimes placed on the Heap.

Remember the rule, Value Types always go where they were declared? Well, if a Value Type is declared outside of a method, but inside a Reference Type, it will be placed within the Reference Type on the Heap.
Here's another example.
If we have the following MyInt class (which is a Reference Type because it is a class):

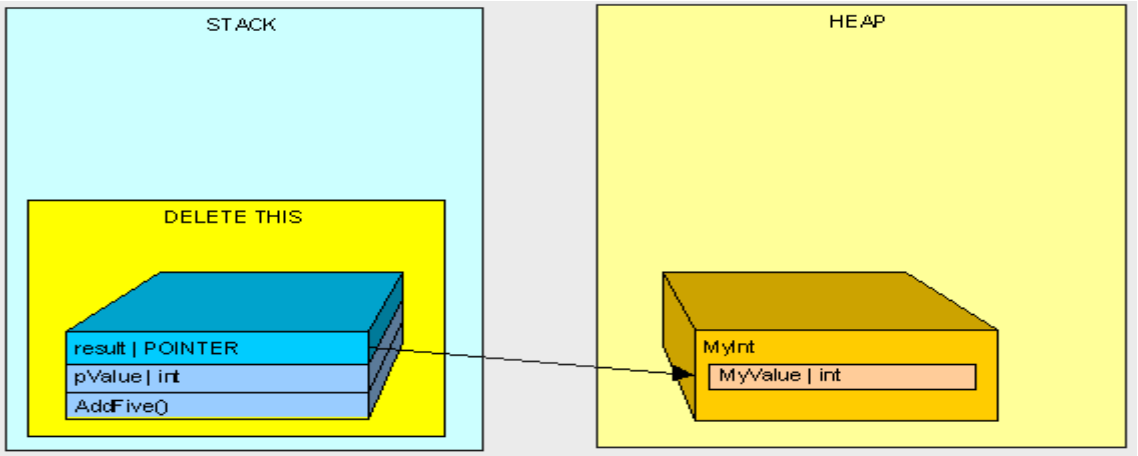
```
1.  public class MyInt
2.  {
3.     public int MyValue;
4.  }
```

and the following method is executing:

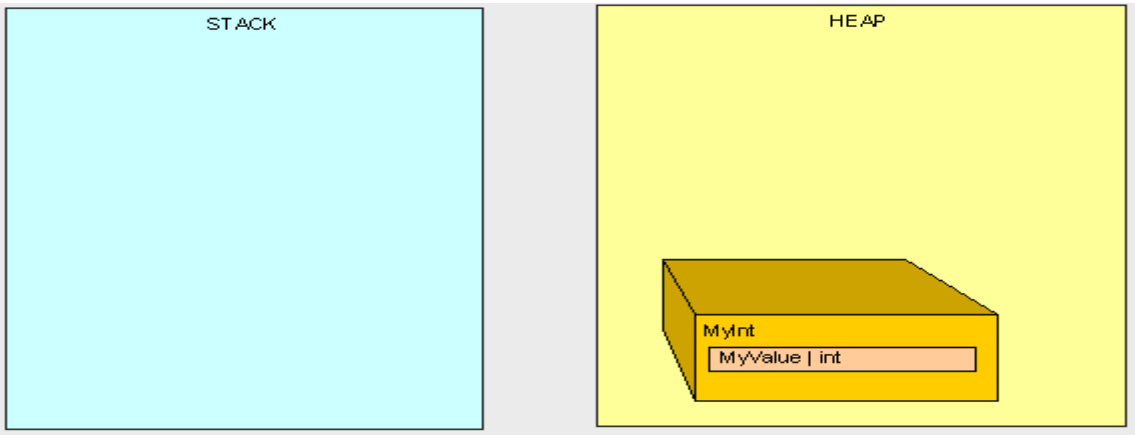
```
1.  public MyInt AddFive(int pValue)
2.  {
3.        MyInt result = new MyInt();
4.        result.MyValue = pValue + 5;
5.        return result;
```



After AddFive() is finished executing (like in the first example), and we are cleaning up...



we're left with an orphaned MyInt in the heap (there is no longer anyone in the Stack standing around pointing to MyInt)!



This is where the Garbage Collection (GC) comes into play. Once our program reaches a certain memory threshold and we need more Heap space, our GC will kick-off.

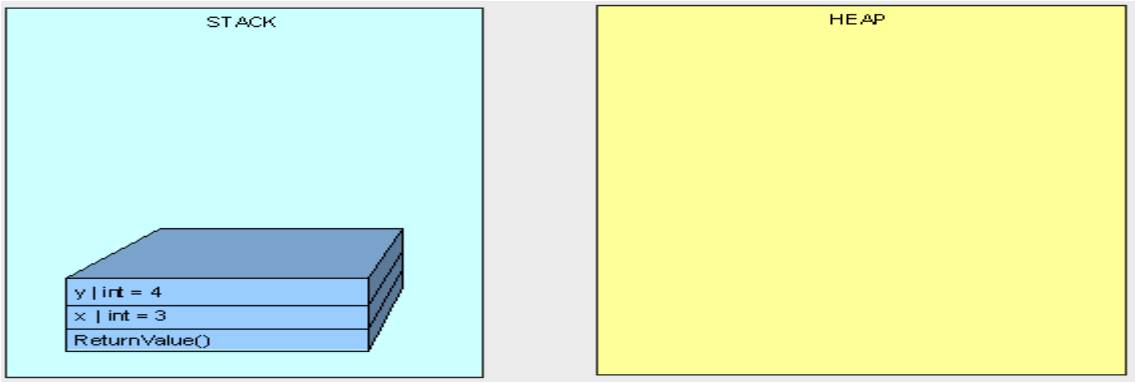
When we are using Reference Types, we're dealing with Pointers to the type, not the thing itself. When we're using Value Types, we're using the thing itself.

Again, this is best described by example.

If we execute the following method:

```
1.  public int ReturnValue()
2.  {
3.      int x = new int();
4.      x = 3;
5.      int y = new int();
6.      y = x;
7.      y = 4;
8.      return x;
9.  }
```

We'll get the value 3



However, if we are using the MyInt class from before

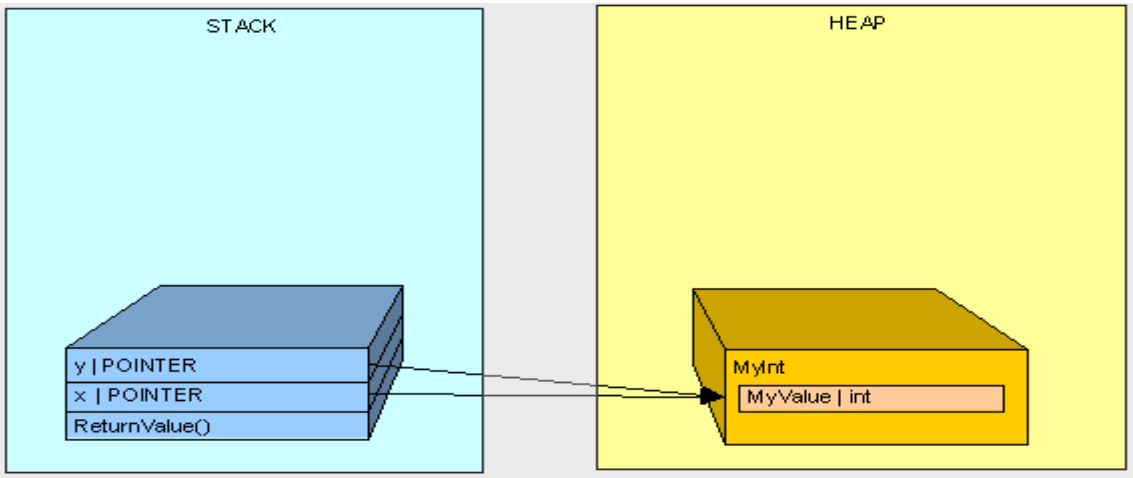
```
1. public class MyInt
2. {
3.       public int MyValue;
4. }
```

and we are executing the following method:

```
1. public int ReturnValue2()
2. {
3.       MyInt x = new MyInt();
4.       x.MyValue = 3;
5.       MyInt y = new MyInt();
6.       y = x;
7.       y.MyValue = 4;
8.       return x.MyValue;
9. }
```

What do we get?... 4!

Why?... How does x.MyValue get to be 4?... Take a look at what we're doing and see if it makes sense:
In the next example, we don't get "3" because of both variables "x" and "y" point to the same object in the Heap.



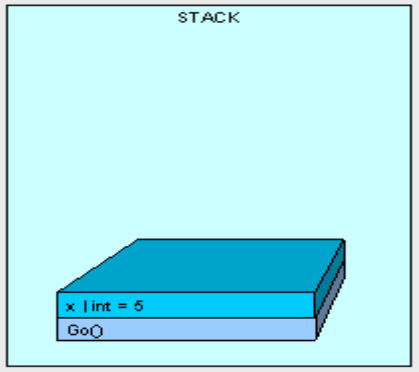
Parameters
When we make a method call here's what happens:

1. Space is allocated for information needed for the execution of our method on the stack (called a Stack Frame). This includes the calling address (a pointer) which is basically a GOTO instruction so when the thread finishes running our method it knows where to go back to in order to continue execution.
2. Our method parameters are copied over. This is what we want to look at more closely.

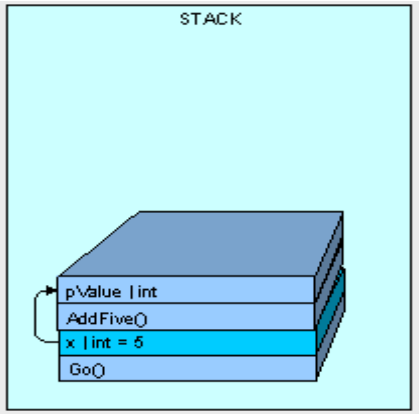
Passing Value Types:

```
1.  class Class1
2.  {
3.      public void Go()
4.      {
5.          int x = 5;
6.          AddFive(x);
7.          Console.WriteLine(x.ToString());
8.      }
9.      public int AddFive(int pValue)
10.     {
11.         pValue += 5;
12.         return pValue;
13.     }
14. }
```

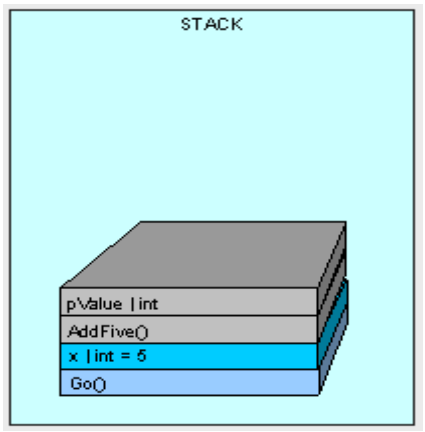
As the method executes, space for "x" is placed on the stack with a value of 5.



Next, AddFive() is placed on the stack with space for it's parameters and the value is copied, bit by bit from x.



When AddFive() has finished execution, the thread is passed back to Go() and because AddFive() has completed, pValue is essentially "removed":

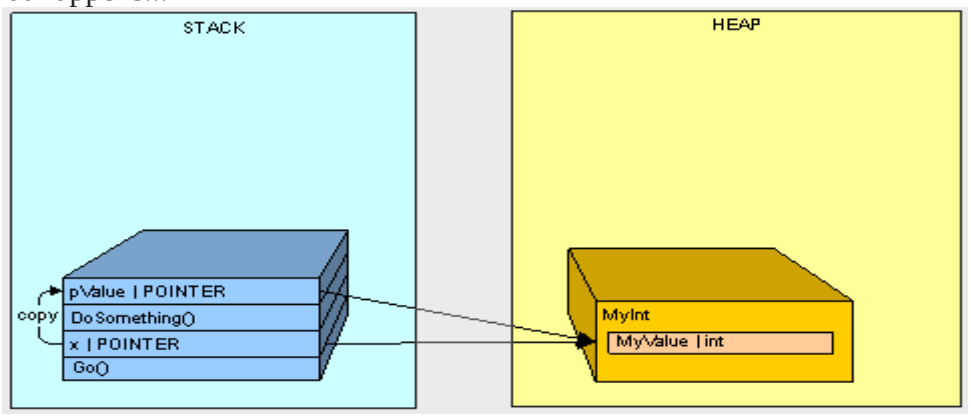


Passing Reference Types

If we execute Go() as in the following code ...

```
1.  public void Go()
2.  {
3.     MyInt x = new MyInt();
4.     x.MyValue = 2;
5.     DoSomething(x);
6.     Console.WriteLine(x.MyValue.ToString());
7.  }
8.   public void DoSomething(MyInt pValue)
9.   {
10.     pValue.MyValue = 12345;
11.  }
```

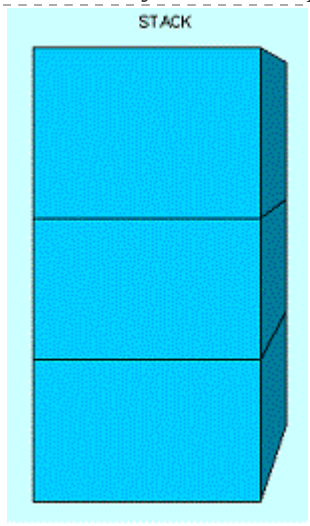
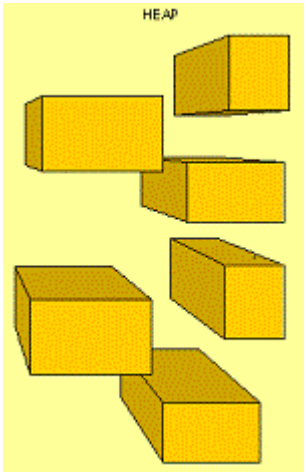
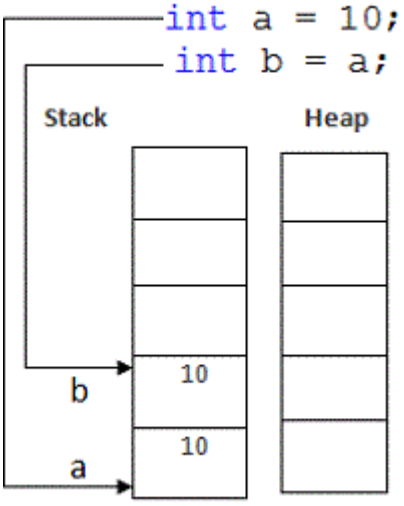
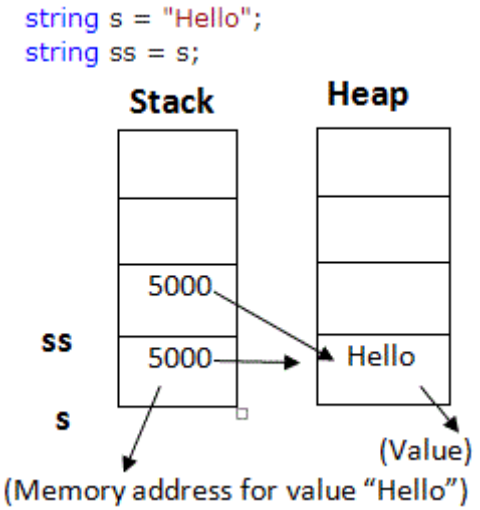
Here's what happens...



1. Starting with the call to Go() the variable x goes on the stack.
2. Starting with the call to DoSomething() the parameter pValue goes on the stack.
3. The value of x (the address of MyInt on the stack) is copied to pValue

So it makes sense that when we change the MyValue property of the MyInt object in the heap using pValue and we later refer to the object on the heap using x, we get the value "12345".

Difference between Stack and Heap Memory in C#

category	Stack Memory	Heap Memory
What is Stack & Heap?	It is an array of memory. It is a LIFO (Last In First Out) data structure. In it data can be added to and deleted only from the top of it.	It is an area of memory where chunks are allocated to store certain kinds of data objects. In it data can be stored and removed in any order.
How Memory is Manages?		
Practical Scenario	 Value of variable storing in stack	 Value of variable storing in heap
What goes on Stack & Heap?	"Things" declared with the following list of type declarations are Value Types (because they are from System.ValueType): bool, byte, char, decimal, double, enum, float, int, long, sbyte, short, struct, uint, ulong, ushort	"Things" declared with following list of type declarations are Reference Types (and inherit from System.Object... except, of course, for object which is the System.Object object): class, interface, delegate, object, string

Memory Allocation	Memory allocation is Static	Memory allocation is Dynamic
How is it Stored?	It is stored Directly	It is stored indirectly
Is Variable Resized?	Variables can't be Resized	Variables can be Resized
Access Speed	Its access is fast	Its access is Slow
How is Block Allocated?	Its block allocation is reserved in LIFO. Most recently reserved block is always the next block to be freed.	Its block allocation is free and done at any time
Visibility or Accessibility	It can be visible/accessible only to the Owner Thread	It can be visible/accessible to all the threads
Used By?	It can be used by one thread of execution	It can be used by all the parts of the application
When wiped off?	Local variables get wiped off once they lose the scope	-
Garbage Collector	-	It is a special thread created by .NET runtime to monitor allocations of heap space. It only collects heap memory since objects are only created in heap

UNIT-III

Object oriented programming and Exception Handling

Classes and objects:

- Everything in C# is associated with classes and objects, along with its attributes and methods.
- For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.
- A Class is like a "blueprint" for creating objects.

Creating class:

To create a class, use the class keyword:

Create a class named "Car" with a variable color:

```
class Car
{
  string color = "red";
}
```

When a variable is declared directly in a class, it is often referred to as a **field** (or attribute).

Creating object:

An object is created from a class. We have already created the class named Car, so now we can use this to create objects.

To create an object of Car, specify the class name, followed by the object name, and use the keyword new:

```
class Car
{
  string color = "red";
  static void Main(string[] args)
  {
    Car myObj1 = new Car();
    Car myObj2 = new Car();
    Console.WriteLine(myObj1.color);
    Console.WriteLine(myObj2.color);
  }
}
```

Using Multiple classes

#Car.cs

```
class Car
{
  public string color = "red";
}
Program.cs
class Program
{
  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.color);
  }
}
```

Constructors:

constructor is a special method that is used to initialize objects. The advantage of a constructor, is that it is called when an object of a class is created. It can be used to set initial values for fields

```
using System;

namespace DefaultConstractor
{
    class addition
    {
        int a, b;
        public addition()   //default contructor
        {
            a = 100;
            b = 175;
        }
        public static void Main()
        {
            addition obj = new addition(); //an object is created , constructor is called
            Console.WriteLine(obj.a);
            Console.WriteLine(obj.b);
            Console.Read();
        }
    }
}
```

Parameterized Constructor in C#

A constructor with at least one parameter is called a parameterized constructor. The advantage of a parameterized constructor is that you can initialize each instance of the class with a different value.

```
using System;

namespace Constructor
{
   class paraconstrctor
   {
     public  int a, b;
     public paraconstrctor(int x, int y)  // decalaring Paremetrized Constructor with ing x,y
parameter
       {
          a = x;
          b = y;
       }
  }
   class MainClass
   {
      static void Main()
      {
         paraconstrctor v = new paraconstrctor(100, 175);   // Creating object of
Parameterized Constructor and ing values
         Console.WriteLine("-----------parameterized constructor example --------“);
         Console.WriteLine("\t");
         Console.WriteLine("value of a=" + v.a );
         Console.WriteLine("value of b=" + v.b);
         Console.Read();
      }
   }
}
```

this keyword:

C# supplies a this keyword that provides access to the current class instance

One possible use of the this keyword is to resolve scope ambiguity, which can arise when an incoming parameter is named identically to a data field of the type.usingSystem.IO;

```
usingSystem;

classStudent{
  publicint id, age;
  publicString name, subject;
```

```
  publicStudent(int id,String name,int age,String subject){
    this.id = id;
    this.name = name;
    this.subject = subject;
    this.age = age;
  }
  publicvoid showInfo(){
    Console.WriteLine(id +" "+ name+" "+age+" "+subject);
  }
}

classStudentDetails{
  publicstaticvoidMain(string[] args){
    Student std1 =newStudent(001,"Jack",23,"Maths");
    Student std2 =newStudent(002,"Harry",27,"Science");
    Student std3 =newStudent(003,"Steve",23,"Programming");
    Student std4 =newStudent(004,"David",27,"English");
    std1.showInfo();
    std2.showInfo();
    std3.showInfo();
    std4.showInfo();
  }
}
```

Constructor chaining:

- Constructor Chaining is an approach where a constructor calls another constructor in the same or base class.
- Using the keyword "this"

```
using System;
using System.Collections.Generic;
using System.Text;

namespace firstsolution
{
      public class mySampleClass
      {
        public mySampleClass() : this(10)
        {
        // No parameter constructor method.// First Constructor
        Console.WriteLine("no parameters");
        }
        public mySampleClass(int Age)
        {
```

```
        Console.WriteLine("one  parameters");// Constructor with one parameter.//
Second Constructor}
      }
      }
      class thiskey
   {
      static void Main()
      {
         mySampleClass m = new mySampleClass();
      }
   }
}
```

Static Class, Methods, Constructors, Fields:

- In C#, static means something which cannot be instantiated.
- we cannot create an object of a static class and cannot access static members using an object.
- C# classes, variables, methods, properties, operators, events, and constructors can be defined as static using the static modifier keyword
- Apply the static modifier before the class name and after the access modifier to make a class static.

Ex: public static class Calculator

You cannot create an object of the static class; therefore the members of the static class can be accessed directly using a class name like ClassName.MemberName

Rules for Static Class:

- Static classes cannot be instantiated.
- All the members of a static class must be static; otherwise the compiler will give an error.
- A static class can contain static variables, static methods, static properties, static operators, static events, and static constructors.
- A static class cannot contain instance members and constructors.
- A static class cannot inherit from other classes.
- Static class members can be accessed using ClassName.MemberName.
- A static class remains in memory for the lifetime of the application domain in which your program resides.

Static Members in Non-static Class:

The normal class (non-static class) can contain one or more static methods, fields, properties, events and other non-static members.

Static fields in a non-static class can be defined using the static keyword.

```
public class StopWatch
{
   public static int InstanceCounter = 0;
   // instance constructor
   public StopWatch()
   {
   }
}
```

Static Methods

- You can define one or more static methods in a non-static class.
- Static methods can be called without creating an object.
- You cannot call static methods using an object of the non-static class.
- The static methods can only call other static methods and access static members.
- You cannot access non-static members of the class in the static methods.

Static Constructors:

- **A non-static class can contain a parameterless static constructor. It can be defined with the static keyword and without access modifiers like public, private, and protected.**

```
using System;
using System.Collections.Generic;
using System.Text;
namespace firstsolution
{
   class demo2
   {
      static demo2()
       {
          Console.WriteLine("in static constructor");
       }
       public demo2()
       {
          Console.WriteLine("in non- static constructor");
       }
   }
   class Class8
   {
      static void Main()
      {
         demo2 d = new demo2();
      }  }
```

OUTPUT:
in static constructor
in non- static constructor

Pillars of object-oriented programming (OOP):

- Encapsulation: How does this language hide an object's internal implementation details and preserve data integrity?
- Inheritance: How does this language promote code reuse?

Polymorphism: How does this language let you treat related objects in a similar way.

The Role of Encapsulation

- The first pillar of OOP is called encapsulation.
- Language's ability to hide unnecessary implementation details from the object user.
- Encapsulation is implemented by using access specifiers.

An access specifier defines the scope and visibility of a class member

C# Access Modifiers

C# Access Modifier	May Be Applied To	Meaning in Life
public	Types or type members	Public items have no access restrictions. A public member can be accessed from an object as well as any derived class. A public type can be accessed from other external assemblies.
private	Type members or nested types	Private items can only be accessed by the class (or structure) that defines the item.
protected	Type members or nested types	Protected items can be used by the class which defines it, and any child class. However, protected items cannot be accessed from the outside world using the C# dot operator.
internal	Types or type members	Internal items are accessible only within the current assembly. Therefore, if you define a set of internal types within a .NET class library, other assemblies are not able to make use of them.
protected internal	Type members or nested types	When the protected and internal keywords are combined on an item, the item is accessible within the defining assembly, the defining class, and by derived classes.

Working with Methods:

```
using System;
namespace firstsolution
{
classdemo1
   {
int x=10;
publicint  getx()
      {
return x;
      }
publicvoid  setx(int a)
      {
         x = a;
      }
   }
classClass7
   {
staticvoid Main()
      {
         demo1 d = new demo1();
         d.setx(3);
int  x=d.getx();
         Console.BackgroundColor = ConsoleColor.DarkBlue;
         Console.WriteLine("hello");
     }
   }
}
```

Encapsulation Using .NET Properties(Worknig with properties)

- A property in C# is a member of a class that provides a flexible mechanism for classes to expose private fields
- Internally, C# properties are special methods called accessors. A C# property have two accessors, get property accessor and set property accessor
- A get accessor returns a property value, and a set accessor assigns a new value.
- The value keyword represents the value of a property.
- Properties in C# and .NET have various access levels that is defined by an access modifier. Properties can be read-write, read-only, or write-only.
- The read-write property implements both, a get and a set accessor.
- A write-only property implements a set accessor, but no get accessor.
- A read-only property implements a get accessor, but no set accessor

[<modifier>] type<Name>
{
[get {stmts}]//get accessor
[set {stmts}]//set accessor
}
Ex: public int xproperty
{
get{ return x;}
set{x=value}
}

WAP to illustrate get and set properties

```
using System;
using System.Collections.Generic;
using System.Text;

namespace firstsolution
{
   public class customer1
   {
      int _custid;
      bool _status;
      string _custname,_state;
      double _balance;
      cities _city;
      public customer1(int custid, bool status, string custname, double balance,cities
city,string state)
      {
         _custid = custid;
         _status = status;
         _custname = custname;
         _balance = balance;
         _city = city;
         _state = state;
      }
      public int custid
      {
         get { return _custid; }
      }
      public bool status
      {
         get { return _status; }
         set { _status = value; }
      }
```

```
        public string custname
        {
           get { return _custname; }
           set { if (_status==true)
              _custname = value; }
        }
        public double balance
        {
           get { return _balance;          }
           set {
              if (_status == true)
              {
                 if (value >= 500)
                    _balance = value;
              }
           }
        }
        public cities city
        {
           get { return _city; }
           set
           {
              if (_status == true)
                 _city = value;
           }
        }
        public string state
        {
           get { return _state; }
            protected set {
              if(_status==true)
                 _state = value; }
        }
        public string country
        {
           get;
        } = "INDIA";
     }
     class Class9
     {
        static void Main()
        {
           customer1 obj = new customer1(101, false, "john",
5000.00,cities.hyderabad,"telangana");
           Console.WriteLine("customer id:" + obj.custid);
           obj.status = true;
```

```
            obj.custname = obj.custname + "smith";
            Console.WriteLine("modified name:" + obj.custname);
            obj.balance = obj.balance - 3000;
            obj.balance = obj.balance - 1600;
            Console.WriteLine("modified balance:" + obj.balance);
            obj.city = cities.mumbai;
            Console.WriteLine("modified  city:" + obj.city);
            Console.WriteLine("state:" + obj.state);
            Console.WriteLine("country:" + obj.country);
        }
    }
}
```

Inheritance:

In C#, it is possible to inherit fields and methods from one class to another. We group the "inheritance concept" into two categories:

- **Derived Class** (child) - the class that inherits from another class
- **Base Class** (parent) - the class being inherited from

To inherit from a class, use the : symbol.

```
using System;
namespace firstsolution
{
   class ABC
   {
     public     void display()
       {
          Console.WriteLine("parent class display");
       }
   }
   class inheritance:ABC
   {
       public  new  void   display()
       {
          Console.WriteLine("child class display");
       }
       static void Main()
       {
          inheritance i = new inheritance();
          ABC a = i;
          a.display();
       }
   }
}
```

The sealed Keyword:
If you don't want other classes to inherit from a class, use the sealed keyword:
f you try to access a sealed class, C# will generate an error:

```
sealedclassVehicle
{
...
}
classCar:Vehicle
{
...
}
```

The error message will be something like this:
'Car': cannot derive from sealed type 'Vehicle'

Polymorphism:

Static Polymorphism

The mechanism of linking a function with an object during compile time is called early binding. It is also called static binding. C# provides two techniques to implement static polymorphism. They are −

- Function overloading
- Operator overloading

Function Overloading

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

The following example shows using function **print()** to print different data types −

```
using System;
namespace PolymorphismApplication {
  class Printdata {
    void print(int i) {
      Console.WriteLine("Printing int: {0}", i );
    }
    void print(double f) {
      Console.WriteLine("Printing float: {0}" , f);
    }
    void print(string s) {
      Console.WriteLine("Printing string: {0}", s);
    }
    static void Main(string[] args) {
```

```
        Printdata p = new Printdata();

        // Call print to print integer
        p.print(5);
        // Call print to print float
        p.print(500.263);
        // Call print to print string
        p.print("Hello C#");
        Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
Printing int: 5
Printing float: 500.263
Printing string: Hello C#
```

Operator overloading

You can redefine or overload most of the built-in operators available in C#. Thus a programmer can use operators with user-defined types as well. Overloaded operators are functions with special names the keyword **operator** followed by the symbol for the operator being defined. similar to any other function, an overloaded operator has a return type and a parameter list.

```
using System;
using System.Collections.Generic;
using System.Numerics;
using System.Runtime.InteropServices.WindowsRuntime;
using System.Text;
using System.Threading;
namespace firstsolution
{
   class Mat
   {
       public int a, b, c, d;
       public Mat(int a, int b, int c, int d)
       {
          this.a = a;
          this.b = b;
          this.c = c;
          this.d = d;
       }
       public static Mat operator+(Mat obj1,Mat obj2)
       {
```

```
        Mat obj = new Mat(obj1.a + obj2.a, obj1.b + obj2.b, obj1.c + obj2.c, obj1.d + obj2.d);
        return obj;
      }
      public static Mat operator -(Mat obj1, Mat obj2)
      {
        Mat obj = new Mat(obj1.a - obj2.a, obj1.b - obj2.b, obj1.c - obj2.c, obj1.d - obj2.d);
        return obj;
      }
   }
   class Class11
   {
      static void Main()
      {
        Mat m1 = new Mat(20, 18, 16, 14);
        Mat m2 = new Mat(10, 8, 6, 4);
        Mat m3 = m1 + m2;
        Mat m4 = m1 - m2;
        Console.WriteLine("the resultant matrix is\n");
        Console.WriteLine(m3.a + " " + m3.b + "\n" + m3.c + " " + m3.d);
        Console.WriteLine("the resultant matrix is");
        Console.WriteLine("\n"+m4.a + " " + m4.b + "\n" + m4.c + " " + m4.d);
      }
   }
}
```

Dynamic Polymorphism

C# allows you to create abstract classes that are used to provide partial class implementation of an interface. Implementation is completed when a derived class inherits from it. **Abstract** classes contain abstract methods, which are implemented by the derived class. The derived classes have more specialized functionality.

Here are the rules about abstract classes −

- You cannot create an instance of an abstract class
- You cannot declare an abstract method outside an abstract class
- When a class is declared **sealed**, it cannot be inherited, abstract classes cannot be declared sealed.

The following program demonstrates an abstract class −

Live Demo

```
using System;

namespace PolymorphismApplication {
  abstract class Shape {
    public abstract int area();
  }
```

```
  class Rectangle:  Shape {
    private int length;
    private int width;
    public Rectangle( int a = 0, int b = 0) {
      length = a;
      width = b;
    }
    public override int area () {
      Console.WriteLine("Rectangle class area :");
      return (width * length);
    }
  }
  class RectangleTester {
    static void Main(string[] args) {
      Rectangle r = new Rectangle(10, 7);
      double a = r.area();
      Console.WriteLine("Area: {0}",a);
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

```
Rectangle class area :
Area: 70
```

Dynamic polymorphism:

When you have a function defined in a class that you want to be implemented in an inherited class(es), you use virtual functions. The virtual functions could be implemented differently in different inherited class and the call to these functions will be decided at runtime.

Dynamic polymorphism is implemented by abstract classes and virtual functions.

The **Override keyword** is used in the derived class of the base class in order to **override** the base class method.

The following program demonstrates this −

```
using System;
namespace PolymorphismApplication {
  class Shape {
    protected int width, height;
    public Shape( int a = 0, int b = 0) {
      width = a;
      height = b;
```

```
    }
    public virtual int area() {
      Console.WriteLine("Parent class area :");
      return 0;
    }
  }
  class Rectangle: Shape {
    public Rectangle( int a = 0, int b = 0): base(a, b) {
    }
    public override int area () {
      Console.WriteLine("Rectangle class area :");
      return (width * height);
    }
  }
  class Triangle: Shape {
    public Triangle(int a = 0, int b = 0): base(a, b) {
    }
    public override int area() {
      Console.WriteLine("Triangle class area :");
      return (width * height / 2);
    }
  }
  class Caller {
    public void CallArea(Shape sh) {
      int a;
      a = sh.area();
      Console.WriteLine("Area: {0}", a);
    }
  }
  class Tester {
    static void Main(string[] args) {
      Caller c = new Caller();
      Rectangle r = new Rectangle(10, 7);
      Triangle t = new Triangle(10, 5);
      c.CallArea(r);
      c.CallArea(t);
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

```
Rectangle class area:
Area: 70
Triangle class area:
Area: 25
```

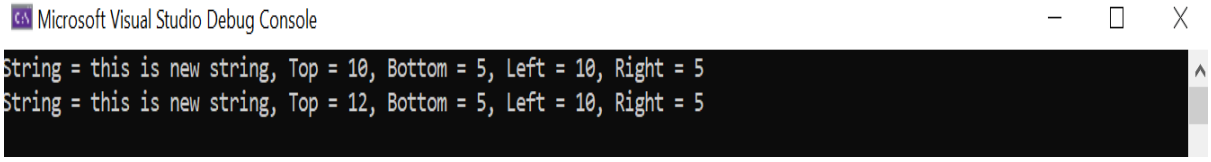
Difference between Object and reference

1. Box b1;//Reference variable,can not create memory
2. b1 **is** called a reference variable (not **object**)
3. now b1 **is null**
4. Box b2 = **new** Box();// object b2 of class BOX is instantiated and create memory on Heap
5. now b2 points box **object**
6. Box b3 = **new** Box();
7. b3.length = 10;
8. b3.width = 20;
9.
10. b1 = b3; //an object is assigned to a reference variable

Copy the reference in another reference variable

```
using System;
using System.Collections.Generic;
using System.Text;

namespace firstsolution
{
   class ShapeInfo
   {
      public string infoString;
      public ShapeInfo(string info)
      {
         infoString = info;
      }
   }
   struct Rectangle
   {
      // The Rectangle structure contains a reference type member.
      public ShapeInfo rectInfo;
      public int rectTop, rectLeft, rectBottom, rectRight;
      public Rectangle(string info, int top, int left, int bottom, int right)
      {
         rectInfo = new ShapeInfo(info);
         rectTop = top; rectBottom = bottom;
         rectLeft = left; rectRight = right;
      }
      public void Display()
      {
         Console.WriteLine("String = {0}, Top = {1}, Bottom = {2}, " +
         "Left = {3}, Right = {4}",
```

```
            rectInfo.infoString, rectTop, rectBottom, rectLeft, rectRight);
        }
    }
class VR2
    {
        static void Main()
        {
            Rectangle r1 = new Rectangle("this is rectangle",10,10,5,5);
            Rectangle r2 = r1;
            r2.rectInfo.infoString = "this is new string";
            r2.rectTop = 12;
            r1.Display();
            r2.Display();
        }
```



Here the assignment of one object reference variable to another didn't create any memory, they will refer to the same object. In other words, any copy of the object is not created, but the copy of the reference is created.

For example, obj1 = obj;

The above line of code just defines that obj1 is referring to the object, obj is referring. So, when you make changes to object using obj1 will also affect the object obj.

Exception Handling:

- An exception is a problem that arises during the execution of a program.
- C# exception handling is built upon four keywords: **try**, **catch**, **finally**, and **throw**.
- **try** − A try block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.
- **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- **finally** − The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
- C# exceptions are represented by classes.
- The exception classes in C# are mainly directly or indirectly derived from the **System.Exception** class

- Some of the exception classes derived from the System.Exception class are the **System.ApplicationException** and **System.SystemException** classes.
- The **System.ApplicationException** class supports exceptions generated by application programs. Hence the exceptions defined by the programmers should derive from this class.
- The **System.SystemException** class is the base class for all predefined system exception.

Sr.No.	Exception Class & Description
1	**System.IO.IOException** Handles I/O errors.
2	**System.IndexOutOfRangeException** Handles errors generated when a method refers to an array index out of range.
3	**System.ArrayTypeMismatchException** Handles errors generated when type is mismatched with the array type.
4	**System.NullReferenceException** Handles errors generated from referencing a null object.
5	**System.DivideByZeroException** Handles errors generated from dividing a dividend with zero.
6	**System.InvalidCastException** Handles errors generated during typecasting.
7	**System.OutOfMemoryException** Handles errors generated from insufficient free memory.
8	**System.StackOverflowException** Handles errors generated from stack overflow.

try..catch..finally

- C# provides three keywords try, catch and finally to implement exception handling.
- The try encloses the statements that might throw an exception whereas catch handles an exception if one exists.
- The finally can be used for any cleanup work that needs to be done**.**

Try..catch..finally block example:

```
01. try
02. {
03. // Statement which can cause an exception.
04. }
05. catch(Type x)
06. {
07. // Statements for handling the exception
08. }
09. finally
10. {
11. //Any cleanup code
12. }
```

- If any exception occurs inside the try block, the control transfers to the appropriate catch block and later to the finally block.
- But in C#, both catch and finally blocks are optional. The try block can exist either with one or more catch blocks or a finally block or with both catch and finally blocks.

```
using System;
class MyClient
{
   public static void Main()
   {
      int x = 0;
      int div = 100/x;
      Console.WriteLine(div);
    }
class MyClient
{
   public static void Main()
   {
      int x = 0;+==
      int div = 0;
      try
      {
         div = 100 / x;
         Console.WriteLine("This linein not executed");
      }
      catch (DivideByZeroException)
      {
         Console.WriteLine("Exception occured");
      }
      Console.WriteLine($"Result is {div}");
   }
}
```

Multiple Catch Blocks :

- A try block can throw multiple exceptions, which can handle by using multiple catch blocks.
- Remember that more specialized catch block should come before a generalized one. Otherwise the compiler will show a compilation error.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;
namespace Exceptions
{
   class multiplecatch
   {
      static void Main(string[] args)
      {
         int a, b, c;
         Console.WriteLine("ENTER ANY TWO NUBERS");
         try
         {
            a = int.Parse(Console.ReadLine());
            b = int.Parse(Console.ReadLine());
            c = a / b;
            Console.WriteLine("C VALUE = " + c);
         }
         catch (DivideByZeroException dbze)
         {
            Console.WriteLine(" second number is zero");
         }
         catch (FormatException fe)
         {
            Console.WriteLine("enter only integer numbers");
         }
         catch
         {
            Console.WriteLine("Exception occured");
         }
         Console.ReadKey();
      }
   }
}
```

throw keyword:

- In c#, the throw is a keyword and it is useful to throw an exception manually during the execution of the program and we can handle those thrown exceptions using try-catch blocks based on our requirements.

- The throw keyword will raise only the exceptions that are derived from the Exception base class.
- Following is the syntax of raising an exception using throw keyword in c#.

```
throw e;
```

- Here, e is an exception that is derived from the Exception class and throw keyword to throw an exception**.**

```
using System;
 namespace  Exceptions
{
   class Program
   {
      static void Main(string[] args)
      {
         try
         {
            GetDetails();
         }
         catch (Exception ex)
         {
            Console.WriteLine(ex.Message);
         }
         Console.ReadLine();
      }
      private static void GetDetails()
      {
         string name = null;
         if (string.IsNullOrEmpty(name))
         {
            throw new NullReferenceException("Name is Empty");
         }
         else
         {
            Console.WriteLine("Name: " + name);
         }
      }
   }
}
```

If you observe the above example, the GetDetails() method will throw NullReferenceException using throw keyword whenever the name variable value is empty or null.

Here, we used a new keyword in throw statement to create an object of valid exception type and we used a try-catch block in method caller to handle thrown exception.

When we execute the above example, we will get the result as shown below.

```
Name is Empty.
```

Exception Properties:

- The StackTrace Property :

The System.Exception.StackTrace property allows you to identify the series of calls that resulted in the exception.

- The HelpLink Property :

The HelpLink property can be set to point the user to a specific URL or standard Windows help file that contains more detailed information.

By default, the value managed by the HelpLink property is an empty string**.**

- The Data Property :

The Data property of System.Exception allows you to fill an exception object with relevant auxiliary information (such as a time stamp)

. The Data property returns an object implementing an interface named IDictionary, defined in the System.Collections namespace**.**

```
using System;
namespace Exceptions
{
   class Exceptionproperties
   {
      static void Main(string[] args)
      {
         int a, b, c;
         Console.WriteLine("ENTER ANY TWO NUBERS");
         try
         {
            a = int.Parse(Console.ReadLine());
            b = int.Parse(Console.ReadLine());
            c = a / b;
            Console.WriteLine("C VALUE = " + c);
         }
         catch (DivideByZeroException dbze)
         {
            Console.WriteLine(dbze.Message);
            Console.WriteLine(dbze.TargetSite);
            Console.WriteLine(dbze.TargetSite.DeclaringType);
            Console.WriteLine(dbze.TargetSite.MemberType);
            Console.WriteLine(dbze.StackTrace);
            Console.WriteLine(dbze.HelpLink = "https://docs.microsoft.com/");
                  }
         catch (FormatException fe)
         {
            fe.Data.Add(" Exception Cause", "Number is not integer");
```

```
            fe.Data.Add("time of Exception", DateTime.Now);
            Console.WriteLine("\n custom data");
            if (fe.Data != null)
            {
               foreach (DictionaryEntry de in fe.Data)
                  Console.WriteLine("{0}:{1}", de.Key, de.Value);
            }
         }
         Console.ReadKey();
      }
   }}
```

Custom Exceptions:

When creating custom exception classes, they should inherit from the System.Exception class (or any of your other custom exception classes from the previous section). The class name should end with the word **Exception.**

```
using System;
namespace Exceptions
{
   public  class newcustomexception:Exception
   {
      public newcustomexception():base()
      {
      }
      public newcustomexception(string msg):base(msg)
      {
      }
   }
   class CustomException
   {
      static void Main(String[] args)
      {
         try
         {
            throw  new  newcustomexception("custom exception is thrown" );
         }
         catch(Exception e)
         {
            Console.WriteLine(e.Message);
         }
      }
   }
}
```

UNIT-IV

DELEGATES AND ASSEMBLIES

UNDERSTANDING .NET DELEGATE TYPE:

- C# delegates are similar to pointers to functions, in C or C++.
- A delegate is a reference type variable that holds the reference to a method. The reference can be changed at runtime.
- Delegates are especially used for implementing events and the call-back methods.
- All delegates are implicitly derived from the System. Delegate class.

Declaring Delegates

- Delegate declaration determines the methods that can be referenced by the delegate.
- A delegate can refer to a method, which has the same signature as that of the delegate.
- Syntax for delegate declaration is −
- <access specifier>delegate <return type><delegate-name><parameter list>

For example, consider a delegate −

public delegate int MyDelegate (string s);

The preceding delegate can be used to reference any method that has a single string parameter and returns an int type variable.

Instantiating Delegates:

MyDelegate del = new MyDelegate(MethodA);

// or

MyDelegate del = MethodA;

Invoke a Delegate

A delegate can be invoked using the Invoke() method Or using the () operator.

Example: Invoke a Delegate

del.Invoke("Hello World!");

// or

del("Hello World!");

Write a program to implement Delegate

```
using System;
namespace Delegates
{
   public delegate void adddelegate(int a, int b);
   public  delegate  string saydelegate(string name);

   class Program
   {
      public void addnums(int a, int b)
      {
         Console.WriteLine(a + b);
      }
      public static string sayhello(string name)
      {
         return "HELLO" + name;
      }
      static void Main(string[] args)
      {
         Program p = new Program();
         adddelegate ad = new adddelegate(p.addnums);
         //adddelegate ad1 = p.addnums;
         saydelegate sd = new saydelegate(sayhello);
         //ad(100, 50);
         ad.Invoke(100, 50);
         //String str =sd("MRCET");
         String str=sd.Invoke("MRCET");
         Console.WriteLine(str);
      }
   }
}
```

Output:
150
HELLOMRCET

MULTICAST DELEGATES:

A Multicast Delegate is a delegate that holds the references of more than one function.

When we invoke the multicast delegate, then all the functions which are referenced by the delegate are going to be invoked.

If you want to call multiple methods using a delegate then all the method signature should be the same.

```
using System;
namespace Delegates
{
   public delegate void  rectdelegate(double width, double height);
   class   Rectangle
   {
      public void   getarea(double width,double height)
      {
         Console.WriteLine("Area of rectangle" + (width * height));
      }
      public void  getperimeter(double width, double height)
         {
         Console.WriteLine("perimeter of rectangle" + (2*(width+height)));
      }
      static void Main()
      {
         Rectangle rect = new Rectangle();
         rectdelegate obj = rect.getarea;
        // obj = obj + rect.getperimeter;
        obj+= rect.getperimeter;

         obj.Invoke(12.34, 56.38);
         //obj = obj - rect.getperimeter;
         obj -= rect.getperimeter;
         obj.Invoke(12.34, 56.38);
      }
   }
}
```

Output:
Area of rectangle695.7292
perimeter of rectangle137.44
Area of rectangle695.7292

Creating Custom namespaces:

Namespaces in C# are used to organize too many classes so that it can be easy to handle the application.

In a simple C# program, we use System.Console where System is the namespace and Console is the class. To access the class of a namespace, we need to use namespacename.classname. We can use **using** keyword so that we don't have to use complete name all the time.

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows

```
namespace namespace_name {
   // code declarations
}
```

To call the namespace-enabled version of either function or variable, prepend the namespace name as follows −

```
namespace_name.item_name;
```

The following program demonstrates use of namespaces −

```
usingSystem;

namespace first_space {
class namespace_cl {
publicvoid func(){
Console.WriteLine("Inside first_space");
}
}
}
namespace second_space {
class namespace_cl {
publicvoid func(){
Console.WriteLine("Inside second_space");
}
}
}
classTestClass{
staticvoidMain(string[] args){
      first_space.namespace_cl fc =new first_space.namespace_cl();
      second_space.namespace_cl sc =new second_space.namespace_cl();
      fc.func();
      sc.func();
Console.ReadKey();
}
}
```

When the above code is compiled and executed, it produces the following result −

```
Inside first_space
Inside second_space
```

The *using* Keyword

The **using** keyword states that the program is using the names in the given namespace. For example, we are using the **System** namespace in our programs. The class Console is defined there. We just write −

```
Console.WriteLine ("Hello there");
```

We could have written the fully qualified name as

System.Console.WriteLine("Hello there");

You can also avoid prepending of namespaces with the **using** namespace directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace

.NET Assembly:

The .NET assembly is the standard for components developed with the Microsoft.NET.

. NET assemblies may or may not be executable, i.e., they might exist as the executable (.exe) file or dynamic link library (DLL) file.

All the standard library files are saved as .dll and userdefined files are saved with .exe extension.

We cannot execute .dll file unlike .exe rather we have to consume them in some other file to execute them.

- .NET supports two kinds of assemblies:
- private
- Shared

Private Assembly

Private assembly requires us to copy separately in all application folders where we want to use that assembly's functionalities; without copying, we cannot access the private assembly features and power.

Private assembly means every time we have one, we exclusively copy into the BIN folder of each application folder.

Private assembly requires us to copy separately in all application folders where we want to use that assembly's functionalities; without copying, we cannot access the private assembly features and power. Private assembly means every time we have one, we exclusively copy into the BIN folder of each application folder.

Public Assembly:

Public assembly is not required to copy separately into all application folders.

Public assembly is also called Shared Assembly.

Only one copy is required in system level, there is no need to copy the assembly into the application folder.

Public assembly should install in GAC.

GAC (Global Assembly Cache):

- When the assembly is required for more than one project or application, we need to make the assembly with a strong name and keep it in GAC or in the Assembly folder by installing the assembly with the gacutil -i command.

Configuring assemblies:

- **Creating private assembly**:
- Open visualstudio->create a new project-> select class library->next->give some name(mydll)->create some method under a class without main->build->build solution
- Open visualstudio->create a new project-> select console application->next->give some name(mydll)-> right click solution->add->project reference>browse->c:->windows->Microsoft .net->assembly->GAC_MSIL->select mydll->mydll.dll
- Now, we can use the method declared in .dll file

Configuring assemblies-creating public assemblies

- Open visualstudio->create a new project-> select class library->next->give some name(mydll)->create some method under a class without main->project tab-> projectname properties->signing->sign the assembly->choose a strong name key file->new->give some file name->ok->build

For Eg:

```
1. public string sayhello()
2. {
3.     return "hello from shared assembly";
4. }
```

Now goto developer command prompt(signin as administrator)->

C:\ cd (give the path where the .dll file is present)\gacutil -i mydll.dll

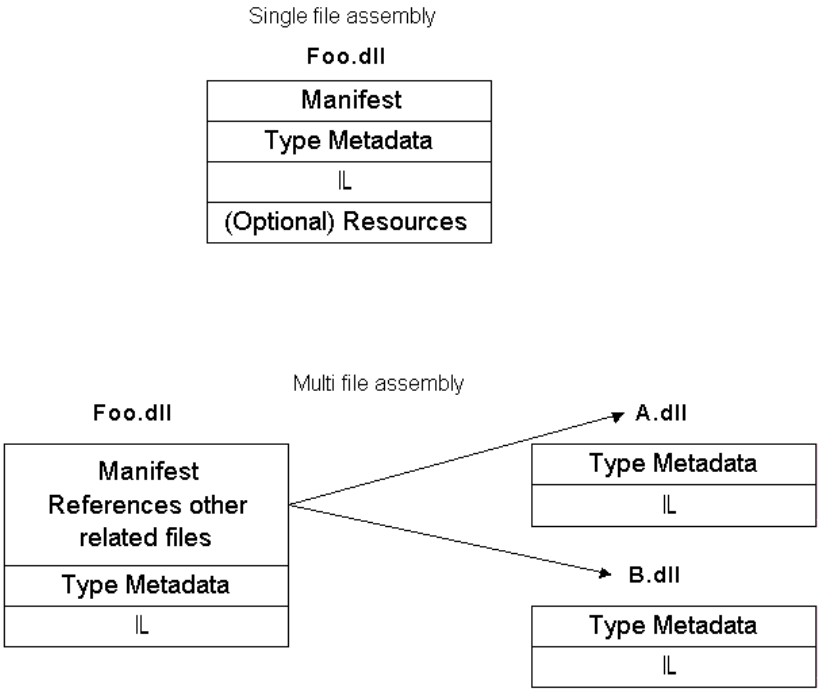
- Open visualstudio->create a new project-> select console application->next->give some name(mydll)-> right click solution->add->project reference>browse->c:->windows->Microsoft .net->assembly->GAC_MSIL->select mydll->mydll.dll
- Now, we can use the method declared in .dll file

Single File assembly and Multifile Assemblies:

When the components(manifest,metadata) of an assembly is grouped in a single physical file, it is known as Single-file assembly.

When the components of an assembly is contained in several files, it is known as Multifile assembly.

These files can be modules of compiled code (.netmodule), resources (such as .bmp or .jpg files), or other files required by the application.


Single file assembly
Foo.dll
Manifest
Type Metadata
IL
(Optional) Resources
Multi file assembly
Foo.dll
Manifest
References other related files
Type Metadata
IL
A.dll
Type Metadata
IL
B.dll
Type Metadata
IL

UNIT-V

ADO.NET

ADO.NET stands for ActiveX Data Object is a database access technology created by Microsoft as part of its .NET framework that can access any kind of data source.
It's a set of object-oriented classes that provides a rich set of data components to create database applications for client- server applications as well as distributed environments over the Internet and intranets.

A High-Level Definition of ADO.NET:

Unlike classic ADO, which was primarily designed for tightly coupled client/server systems, ADO.NET was built with the disconnected world in mind, using DataSets. This type represents a local copy of any number of related data tables, each of which contains a collection of rows and column. Using the DataSet, the calling assembly (such as a web page or desktop executable) is able to manipulate and update a DataSet's contents while disconnected from the data source and send any modified data back for processing using a related data adapter.

The Faces of ADO.NET:

You can use the ADO.NET libraries in three conceptually unique manners: connected, disconnected, or through the Entity Framework. When you use the connected layer, your code base explicitly connects to and disconnects from the underlying data store. When you use ADO.NET in this manner, you typically interact with the data store using connection objects, command objects, and data reader objects.

The disconnected layer allows you to manipulate a set of DataTable objects (contained within a DataSet) that functions as a client-side copy of the external data. When you obtain a DataSet using a related data adapter object, the connection is automatically opened and closed on your behalf.

Once a caller receives a DataSet, it is able to traverse and manipulate the contents without incurring the cost of network traffic. Also, if the caller wishes to submit the changes back to the data store, the data adapter (in conjunction with a set of SQL statements) is used to update the data source; at this point the connection is reopened for the database updates to occur, and then closed again immediately.

ADO.NET data providers:

A data provider is a set of types defined in a given namespace that understand how to communicate with a specific type of data source. Regardless of which data provider you use, each defines a set of class types that provide core functionality.

Table 21-1 documents some of the core common types, their base class (all defined in the System.Data.Common namespace), and the key interfaces (each is defined in the System.Data namespace) they implement.

Table 21-1. *The Core Objects of an ADO.NET Data Provider*

Type of Object	Base Class	Relevant Interfaces	Meaning in Life
Connection	DbConnection	IDbConnection	Provides the ability to connect to and disconnect from the data store. Connection objects also provide access to a related transaction object.
Command	DbCommand	IDbCommand	Represents a SQL query or a stored procedure. Command objects also provide access to the provider's data reader object.
DataReader	DbDataReader	IDataReader, IDataRecord	Provides forward-only, read-only access to data using a server-side cursor.
DataAdapter	DbDataAdapter	IDataAdapter, IDbDataAdapter	Transfers DataSets between the caller and the data store. Data adapters contain a connection and a set of four internal command objects used to select, insert, update, and delete information from the data store.
Parameter	DbParameter	IDataParameter, IDbDataParameter	Represents a named parameter within a parameterized query.
Transaction	DbTransaction	IDbTransaction	Encapsulates a database transaction.

Although the specific names of these core classes will differ among data providers (e.g., SqlConnection vs. OracleConnection vs. OdbcConnection vs. MySqlConnection), each class derives from the same base class (DbConnection, in the case of connection objects) that implements identical interfaces (e.g., IDbConnection)

Figure 21-2 shows the big picture behind ADO.NET data providers. Note how the diagram illustrates that the Client Assembly can literally be any type of .NET application: console program, Windows Forms application, WPF application, ASP.NET web page, WCF service, a .NET code library, and so on.

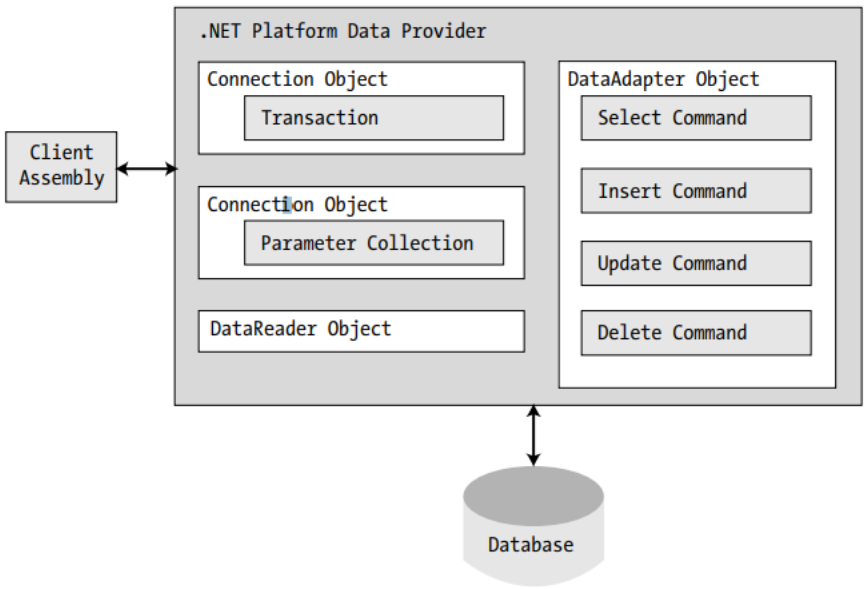


Figure 21-2. *ADO.NET data providers provide access to a given DBMS*

A data provider will supply you with other types beyond the objects shown in Figure 21-2; however, these core objects define a common baseline across all data providers.

The Microsoft-Supplied ADO.NET Data Providers

Table 21-2. *Microsoft ADO.NET Data Providers*

Data Provider	Namespace	Assembly
OLE DB	System.Data.OleDb	System.Data.dll
Microsoft SQL Server	System.Data.SqlClient	System.Data.dll
Microsoft SQL Server Mobile	System.Data.SqlServerCe	System.Data.SqlServerCe.dll
ODBC	System.Data.Odbc	System.Data.dll

Additional ADO.NET Namespaces:

In addition to the .NET namespaces that define the types of a specific data provider, the .NET base class libraries provide a number of additional ADO.NET-centric namespaces, some of which you can see in Table 21-3

Namespace	Meaning in Life
Microsoft.SqlServer.Server	This namespace provides types that facilitate CLR and SQL Server 2005 and later integration services.
System.Data	This namespace defines the core ADO.NET types used by all data providers, including common interfaces and numerous types that represent the disconnected layer (e.g., DataSet and DataTable).
System.Data.Common	This namespace contains types shared between all ADO.NET data providers, including the common abstract base classes.
System.Data.Sql	This namespace contains types that allow you to discover Microsoft SQL Server instances installed on the current local network.
System.Data.SqlTypes	This namespace contains native data types used by Microsoft SQL Server. You can always use the corresponding CLR data types, but the SqlTypes are optimized to work with SQL Server (e.g., if your SQL Server database contains an integer value, you can represent it using either int or SqlTypes.SqlInt32).

System. Data Namespace

Of all the ADO.NET namespaces, System.Data is the lowest common denominator. You cannot build ADO.NET applications without specifying this namespace in your data access applications.

This namespace contains types that are shared among all ADO.NET data providers, regardless of the underlying data store. In addition to a number of database-centric exceptions (e.g.,NoNullAllowedException,RowNotInTableException,andMissingPrimaryKeyException) , System.Data contains types that represent various database primitives (e.g., tables, rows, columns, and constraints), as well as the common interfaces implemented by data provider objects. Table 21-4 lists some of the core types you should be aware of.

Type	Meaning in Life
Constraint	Represents a constraint for a given DataColumn object.
DataColumn	Represents a single column within a DataTable object.
DataRelation	Represents a parent/child relationship between two DataTable objects.
DataRow	Represents a single row within a DataTable object.
DataSet	Represents an in-memory cache of data consisting of any number of interrelated DataTable objects.
DataTable	Represents a tabular block of in-memory data.
DataTableReader	Allows you to treat a DataTable as a fire-hose cursor (forward only, read-only data access).
DataView	Represents a customized view of a DataTable for sorting, filtering, searching, editing, and navigation.
IDataAdapter	Defines the core behavior of a data adapter object.
IDataParameter	Defines the core behavior of a parameter object.
IDataReader	Defines the core behavior of a data reader object.
IDbCommand	Defines the core behavior of a command object.

Role ofIDbConnection Interface:

- The IDbConnection type is implemented by a data provider's connection object. This interface defines a set of members used to configure a connection to a specific data store. It also allows you to obtain the data provider's transaction object.

Role ofIDbTransaction Interface:

- The overloaded BeginTransaction() method defined by IDbConnection provides access to the provider's transaction object. You can use the members defined by IDbTransaction to interact programmatically with a transactional session and the underlying data store

Role ofIDbCommand Interface:

- The IDbCommand interface, which will be implemented by a data provider's command object. Like other data access object models, command objects allow programmatic manipulation of SQL statements, stored procedures, and parameterized queries. Command objects also provide access to the data provider's data reader type through the overloaded ExecuteReader() method

Role of IDbDataAdapter and IDataAdapter Interfaces:

- You use data adapters to push and pull DataSets to and from a given data store.
- The IDbDataAdapter interface defines a set of properties that you can use to maintain the SQL statements for the related select, insert, update, and delete operations:

Role of IDataReader and IDataRecord Interfaces:

- The next key interface to be aware of is IDataReader, which represents the common behaviors supported by a given data reader object.
- When you obtain an IDataReader-compatible type from an ADO.NET data provider, you can iterate over the result set in a forw
- IDataReader extends IDataRecord, which defines many members that allow you to extract a strongly typed value from the streamard-only, read-only manner

Abstracting Data Providers Using Interfaces:

even though the exact names of the implementing types will differ among data providers, you can program against these types in a similar manner —that's the beauty of interfacebased polymorphism.

For example, if you define a method that takes an IDbConnection parameter, you can pass in any ADO.NET connection object:

```
public static void OpenConnection(IDbConnection cn)
{ // Open the incoming connection for the caller.
cn.Open();
 }
```

For example, consider the following simple C# Console Application project (named MyConnectionFactory), which allows you to obtain a specific connection object based on the value of a custom enumeration

```
namespace MyConnectionFactory
{
// A list of possible providers.
 enum DataProvider
 { SqlServer, OleDb, Odbc, None }
 class Program
 {
 static void Main(string[] args)
 {
 Console.WriteLine("**** Very Simple Connection Factory *****\n");
 // Get a specific connection.
 IDbConnection myCn = GetConnection(DataProvider.SqlServer);
 Console.WriteLine("Your connection is a {0}", myCn.GetType().Name);
 // Open, use and close connection...
 Console.ReadLine();
 }
 // This method returns a specific connection object
 // based on the value of a DataProvider enum.
 static IDbConnection GetConnection(DataProvider dp)
 {
 IDbConnection conn = null;
 switch (dp)
 {
 case DataProvider.SqlServer:
 conn = new SqlConnection();
 break;
 case DataProvider.OleDb:
 conn = new OleDbConnection();
 break;
 case DataProvider.Odbc:
 conn = new OdbcConnection();
 break;
 }
 return conn;
 }
 }
}
```

Creating the AutoLot Database:
Creating the Inventory Table

To begin building your testing database, launch Visual Studio 2010 and open the Server Explorer using the View menu of the IDE. Next, right-click the Data Connections node and select the Create New SQL Server Database menu option. In the resulting dialog box, connect to the SQL Server installation on your local machine (with the (local) token) and specify AutoLot as the database name (Windows Authentication should be fine; see Figure 21-3).

Note If you use SQL Server Express, you will need to enter (local)\SQLEXPRESS in the Server name text box.

Figure 21-3. *Creating a new SQL Server 2008 Express database with Visual Studio 2010*

At this point, the AutoLot database is empty of any database objects (e.g., tables and stored procedures). To insert a new database table, right-click the Tables node and select Add New Table (see Figure 21-4)

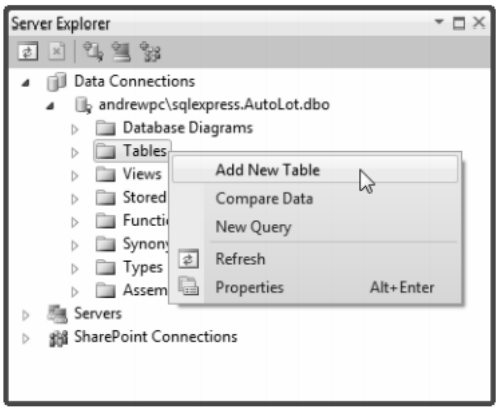
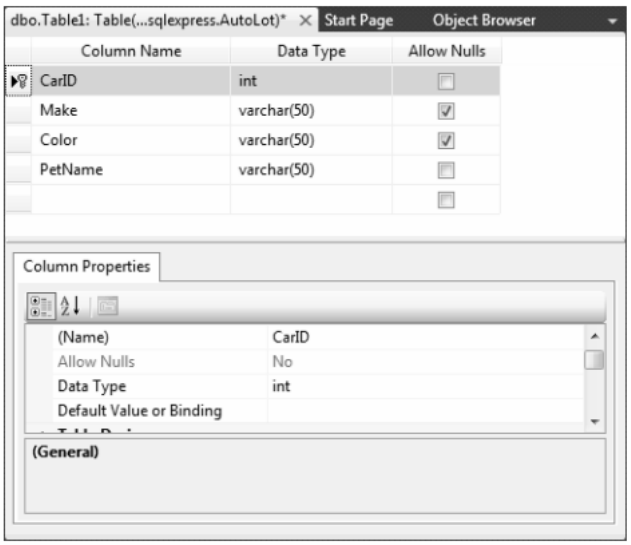


Figure 21-4. *Adding the Inventory table*

Use the table editor to add four columns (CarID, Make, Color, and PetName). Ensure that the CarID column has been set to the Primary Key (do this by right-clicking the CarID row and selecting Set Primary Key). Figure 21-5 shows the final table settings (you don't need to change anything in the Column Properties editor, but you should notice the data types for each column).



Save (and then close) your new table; also, be sure you name this new database object Inventory. At this point, you should see the Inventory table under the Tables node of the Server Explorer. Right-click the Inventory table icon and select Show Table Data

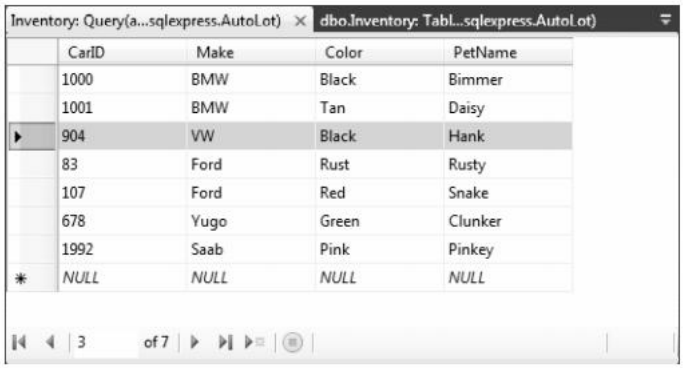


Figure 21-6. *Populating the Inventory table*

Creating the Customers and Orders Tables:

Your testing database will have two additional tables: Customers and Orders. The Customers table (as the name suggests) will contain a list of customers; these which will be represented by three columns: CustID (which should be set as the primary key), FirstName, and LastName. You can create the Customers table by following the same steps you used to create the Inventory table; be sure to create the Customers table using the schema shown in Figure 21-8.

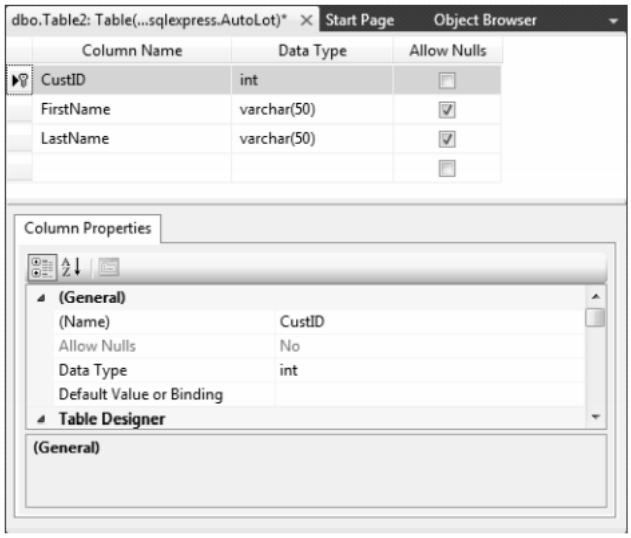


Figure 21-8. *Designing the Customers table*

After you save your table, add a handful of customer records (see Figure 21-9).

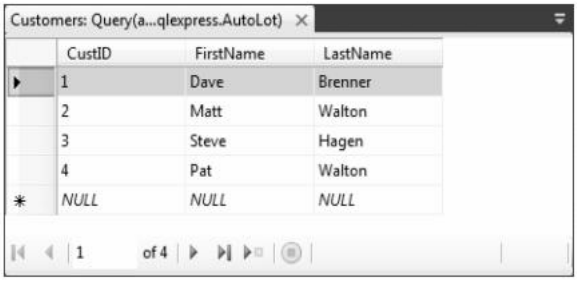
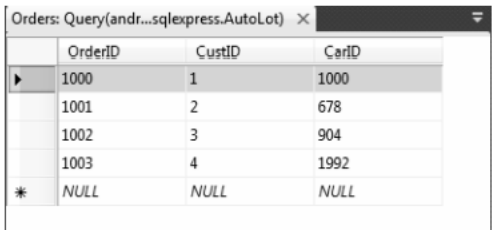


Figure 21-9. *Populating the Customers table*

Figure 21-10. *Designing the Orders table*

Now add data to your `Orders` table. Assuming that the `OrderID` value begins at `1000`, select a unique `CarID` for each `CustID` value (see Figure 21-11).



To Establish a Connection with the database in connection oriented architechture

name space to be used:

system.data.sqlclient

Connection: Use to establish the connection with the database.

Steps:

1. Declare connection object

syntax:

sqlconnection con;

2. Define connection object

con= new sqlconnection("path to the database");

3. Open the connection:

objectname.open();

con.open();

4. close the connection

con.close();

To send a command to a database

Command :

step1:

Declare command object

sqlcommand cmd;

step 2:

Define command object

cmd=new sqlcommand("insert/delete/update",con);

step 3:

mention command type

object.commandtype= commandtype.value;

Step 4:

con.open();

step 5 : define execution method

commandobject cmd;

cmd.executionmethod()

sqlDatareader sd;

Execution methods:

1. ExecuteNonquery()-action queries (insert,delete,update,create)

2.ExecuteReader()-Non action queries-select

3.Executescalar()-Non action queries-(select along with aggregate functions(min,max avg....)

```
class program
{
static void Main()
{
  sqlconnection con;
con= new sqlconnection("path to the database");
con.open();
 string starsql="select * from Employee";
sqlcommand mycommand=new sqlcommand(starsql,"con");
sqldatareader mydatareader;
mydatareader=mycommand.Exeutrereader();
while(mydatareader.read())
{
----
----
}
myreader.close();
}
```

Understanding the Connected Layer of ADO.NET

You need to perform the following steps when you wish to connect to a database and read the records using a data reader object:

1. Allocate, configure, and open your connection object.

2. Allocate and configure a command object, specifying the connection object as a constructor argument or with the Connection property.

3. Call ExecuteReader() on the configured command class.

4. Process each record using the Read() method of the data reader.

create a new Console Application named AutoLotDataReader and import the System.Data and System.Data.SqlClient namespaces. Here is the complete code within Main(); an analysis will follow

```
class Program
{
 static void Main(string[] args)
 {
 Console.WriteLine("***** Fun with Data Readers *****\n");
 // Create an open a connection.
 using(SqlConnection cn = new SqlConnection())
 {
 cn.ConnectionString =
 @"Data Source=(local)\SQLEXPRESS;Integrated Security=SSPI;" +
 "Initial Catalog=AutoLot";
```

```
cn.Open();
// Create a SQL command object.
string strSQL = "Select * From Inventory";
SqlCommand myCommand = new SqlCommand(strSQL, cn);
// Obtain a data reader a la ExecuteReader().
using(SqlDataReader myDataReader = myCommand.ExecuteReader())
{
// Loop over the results.
while (myDataReader.Read())
{
Console.WriteLine("-> Make: {0}, PetName: {1}, Color: {2}.",
myDataReader["Make"].ToString(),
myDataReader["PetName"].ToString(),
myDataReader["Color"].ToString());
}
}
}
Console.ReadLine();
}
}
```

Working with Connection Objects:

The first step to take when working with a data provider is to establish a session with the data source using the connection object (which, as you recall, derives from DbConnection).

.NET connection objects are provided with a formatted connection string; this string contains a number of name/value pairs, separated by semicolons.

You use this information to identify the name of the machine you wish to connect to, required security settings, the name of the database on that machine, and other data provider–specific information.

Once you establish your construction string, you can use a call to Open() to establish a connection with the DBMS.

Working with Command Objects

When you create a command object, you can establish the SQL query as a constructor parameter or directly by using the CommandText property.

Also when you create a command object, you need to specify the connection you want to use.

string strSQL = "Select * From Inventory";

SqlCommand myCommand = new SqlCommand(strSQL, cn);

Working with Data Readers

Once you establish the active connection and SQL command, the next step is to submit the query to the data source.

Data readers are useful when you need to iterate over large amounts of data quickly and you do not need to maintain an in-memory representation

You obtain data reader objects from the command object using a call to ExecuteReader().

The data reader represents the current record it has read from the database. The data reader has an indexer method (e.g, [] syntax in C#) that allows you to access a column in the current record.

You can access the column either by name or by zero-based integer.

The following use of the data reader leverages the Read() method to determine when you have

reached the end of your records (using a false return value). For each incoming record that you read

from the database, you use the type indexer to print out the make, pet name, and color of each

automobile. Also note that you call Close() as soon as you finish processing the records; this frees up theconnection object:

```
static void Main(string[] args)
{
...
 // Obtain a data reader via ExecuteReader().
 using(SqlDataReader myDataReader = myCommand.ExecuteReader())
 {
 // Loop over the results.
 while (myDataReader.Read())
{
 Console.WriteLine("-> Make: {0}, PetName: {1}, Color: {2}.",
 myDataReader["Make"].ToString(),
 myDataReader["PetName"].ToString(),
 myDataReader["Color"].ToString());
 }
 }
 Console.ReadLine();
}
```

Adding the Insertion Logic:

Inserting a new record into the Inventory table is as simple as formatting the SQL Insert statement (based on user input) and calling the ExecuteNonQuery() using your command object

```
public void InsertAuto(int id, string color, string make, string petName)
{
 // Format and execute SQL statement.
 string sql = string.Format("Insert Into Inventory" +
 "(CarID, Make, Color, PetName) Values" +
 "('{0}', '{1}', '{2}', '{3}')", id, make, color, petName);
 // Execute using our connection.
 using(SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
 {
 cmd.ExecuteNonQuery();
 }
}
```

Adding the Deletion Logic:

Deleting an existing record is as simple as inserting a new record. Unlike when you created the code listing for InsertAuto(), this time you will learn about an important try/catch scope that handles the possibility of attempting to delete a car that is currently on order for an individual in the Customers table.

```
string sql = string.Format("Delete from Inventory where CarID = '{0}'", id);
using(SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
 {
 try
 {
 cmd.ExecuteNonQuery();
 }
 catch(SqlException ex)
 {
 Exception error = new Exception("Sorry! That car is on order!", ex);
 throw error;
 }
 }
}
```

Adding the Update Logic:

```
// Get ID of car to modify and new pet name.
string sql = string.Format("Update Inventory Set PetName = '{0}' Where CarID = '{1}'",
newPetName, id);
using(SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
{
cmd.ExecuteNonQuery();
}
}
```

Adding the Selection Logic:

```
string sql = "Select * From Inventory";
using (SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
{
SqlDataReader dr = cmd.ExecuteReader();
while (dr.Read())
{
inv.Add(new NewCar
{
CarID = (int)dr["CarID"],
Color = (string)dr["Color"],
Make = (string)dr["Make"],
PetName = (string)dr["PetName"]
});
}
dr.Close();
```

Understanding the Disconnected Layer of ADO.NET:

Connectionless oriented architecture contains:

- Connection
- DataAdapter
- DataSet

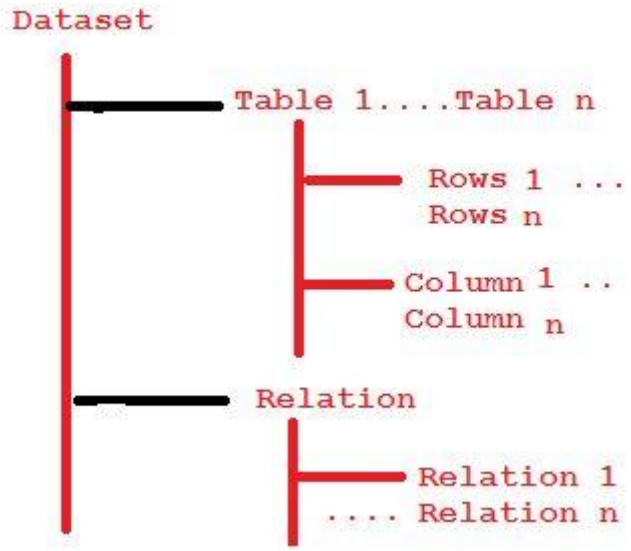
Connection: Connection class uses to establish the connection between the front end and back end.

1. SqlConnection con=**new** SqlConnection(“integratd security=**true**;initial catalog=Table Name;data source=.”);

DataAdapter: DataAdapter behaves as a mediator between data source and table.

1. SqlDataAdapter da=**new** SqlDataAdapter(“Query which has to excute”,Connection **object**);

DataSet: DataSet contains the tables and relationships as in the following figure:



DataAdapter does not have a feature of containing data so there is a dataset that contains table and relation after generating result set.

Syntax for DataSet is:

DataSet ds=new DataSet();

Da.Fill(ds,"X");

The architecture of connectionless is as in the following figure:

Client-side interacts with server-side through ADO.NET because the front end application will not understand the syntax of back end application so there is ADO.NET which is a group of classes that contain Connection, Command, DataAdapter, and DataSet.