.Net Programming

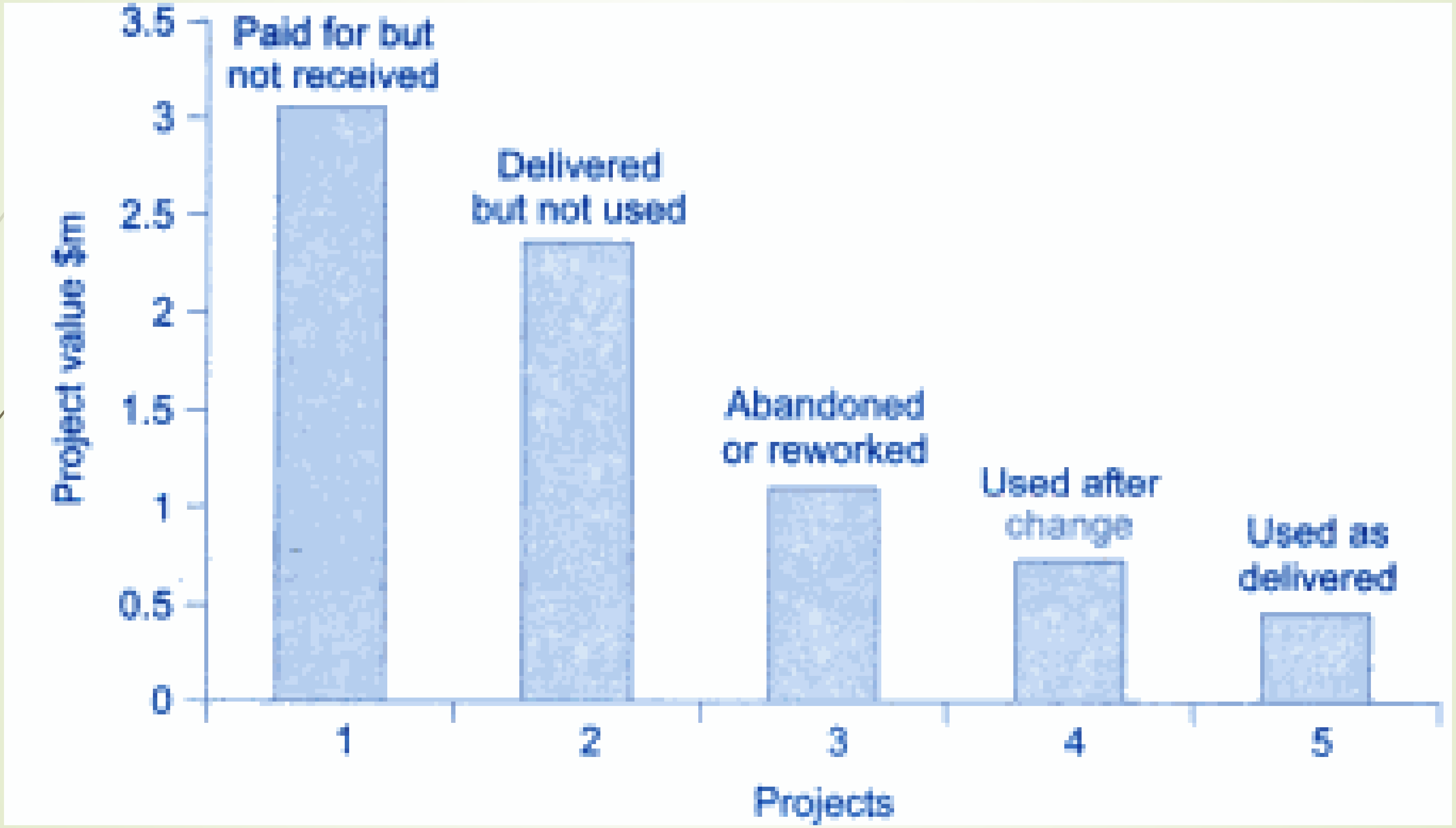
Unit 1 - Introduction to .Net Programming

By – Dr. Jay B. Teraiya

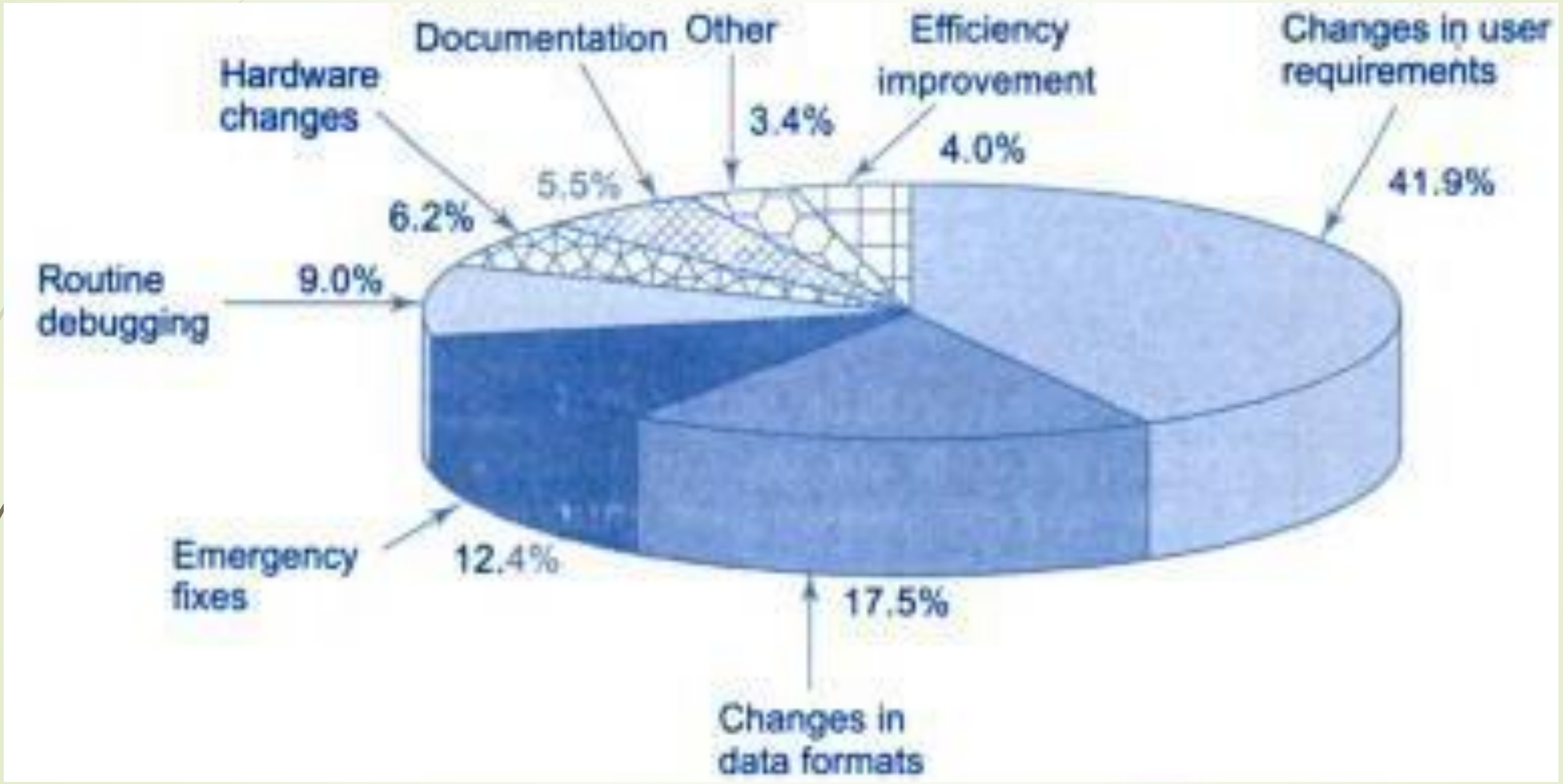
Agenda of Unit 1

- Software Crisis and Software Evolution
- Introduction to .NET Framework
- .NET Framework Components
- Namespace
- .NET Assembly and MetaData
- Garbage Collection
- Boxing and UnBoxing
- Operators in C#
- Array
- Decisions (if types and switch case)
- Loops
- Collection

Software Crisis and Software Evolution



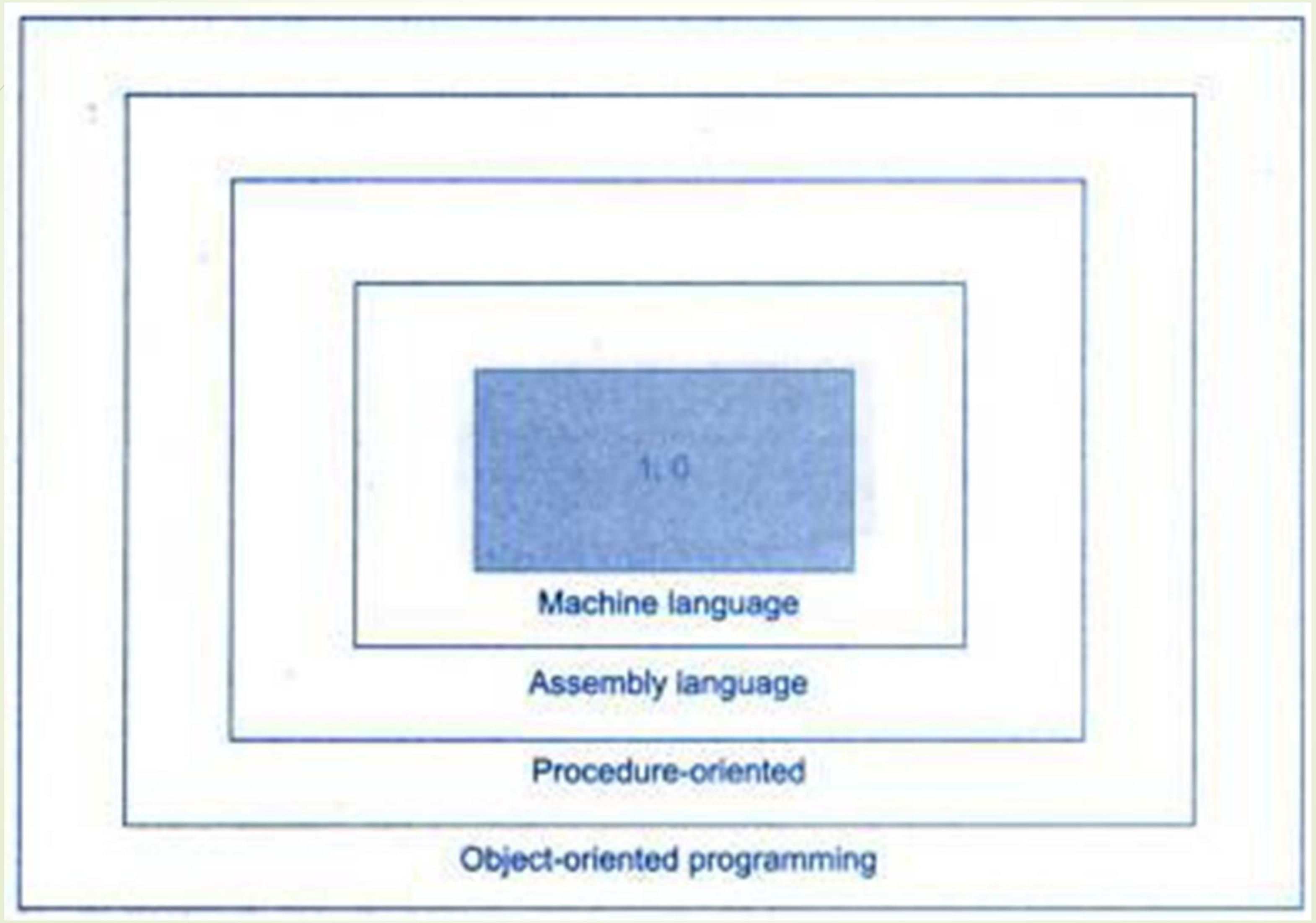
Software Crisis and Software Evolution (Cont.)



Software Crisis and Software Evolution (Cont.)

- Some of the quality issues that must be considered for the software development are bellowed.
  - 1 – Correctness
  - 2 – Maintainability
  - 3 – Reusability
  - 4 – Openness and interoperability
  - 5 – Portability
  - 6 – Security
  - 7 – Integrity
  - 8 – User friendliness

Software Crisis and Software Evolution (Cont.)



Introduction to .NET Framework

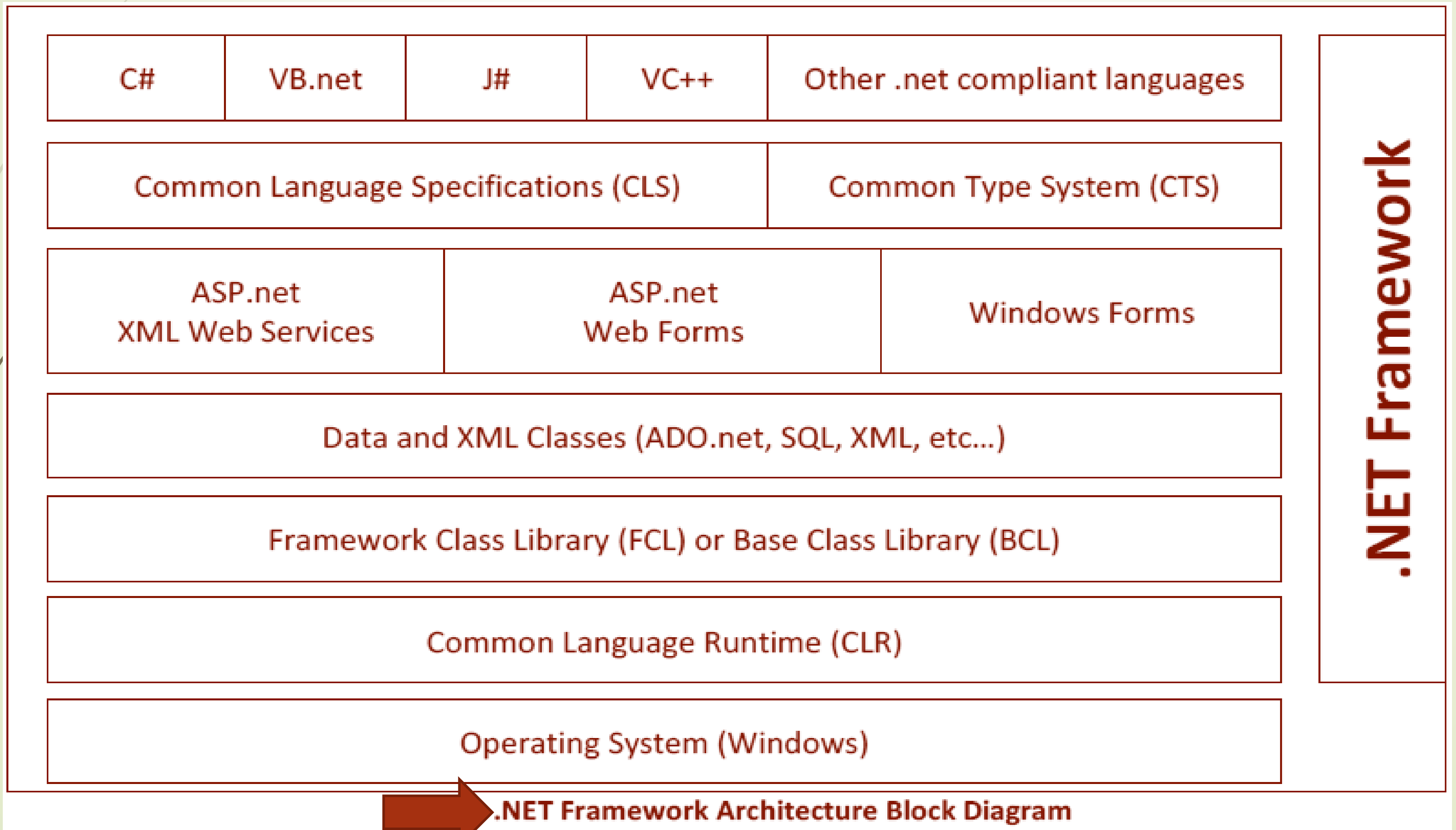
- **Why do we need a framework?**
- Let us take an example
- If I told you to cut a piece of paper with dimensions 5cm by 5cm then surely you would do that.
- But then I ask you to cut 1000 pieces of paper of the same dimensions.
- You won't do the measuring 1000 times, obviously you would make a frame of 5cm by 5cm and then with the help of it you would be able to cut 1000 papers in less time.
- So, what you did is made a framework which would do that type of task and performing the same type of task again and again for the same type of applications.

Introduction to .NET Framework

- **What is .NET framework?**
- .NET (pronounced dot net) is a framework that provides a programming guidelines that can be used to develop a wide range of applications–from web to mobile to Windows-based applications.
- .NET framework runs on different versions of windows operating system, starting from windows 95 to latest versions of windows 7 and windows 8 & 10.
- It is a framework that supports **Multiple Language (40+)** and Cross language integration.
- .NET Framework also includes the .NET Common Language Runtime (CLR), which is responsible for maintaining the execution of all applications developed using the .NET library.

Introduction to .NET Framework

- **.NET Framework Architecture**


C#
VB.net
J#
VC++
Other .net compliant languages
Common Language Specifications (CLS)
Common Type System (CTS)
ASP.net XML Web Services
ASP.net Web Forms
Windows Forms
Data and XML Classes (ADO.net, SQL, XML, etc...)
Framework Class Library (FCL) or Base Class Library (BCL)
Common Language Runtime (CLR)
Operating System (Windows)
.NET Framework

.NET Framework Architecture Block Diagram

.NET Framework Components

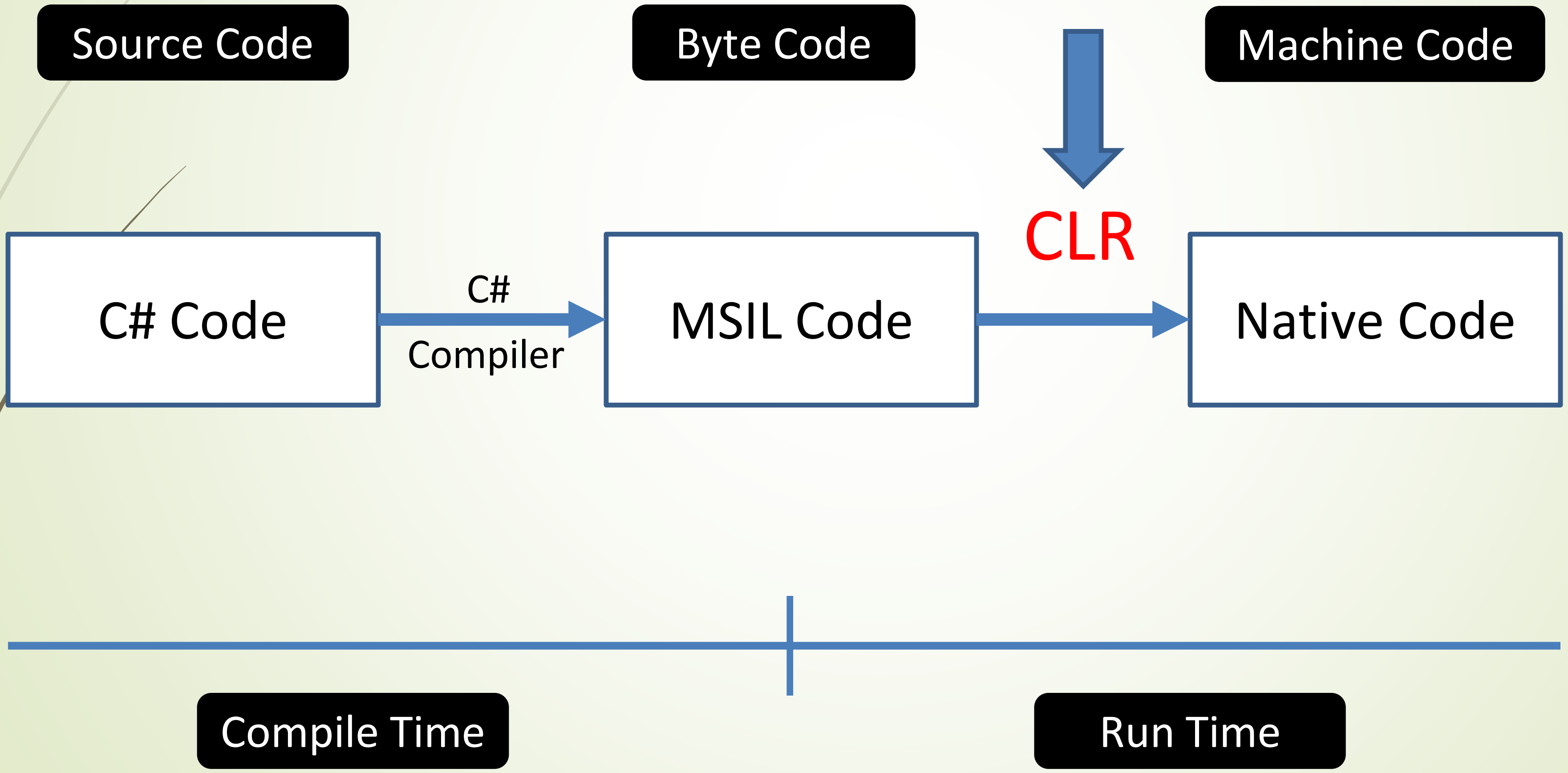
- Common Language Runtime (CLR)
- Framework Class Library (FCL) = Base Class Library (BCL)
- Common Language Specification (CLS)
- Common Type System (CTS)
- Common Intermediate Language (CIL)

.NET Framework Components

- **Common Language Runtime (CLR)**
- The CLR is the execution engine for .NET applications and serves as the interface between .NET applications and the operating system.
- The CLR is somewhat comparable to the Java Virtual Machine (JVM) that Sun Microsystems furnishes for running programs compiled from the Java language.
- Programmers write code in any language like VB.NET, C# and F# etc.
- But when they compile their programs into an intermediate code called MSIL (Microsoft Intermediate Language) that can be managed by the CLR and then the CLR converts it into machine code to be executed by the processor.
- The CLR manages memory, Thread Execution, Garbage Collection (GC), Exception Handling, Common Type System (CTS), code safety verifications and other system services.

.NET Framework Components

- **Common Language Runtime (CLR)**



.NET Framework Components

- **Common Language Runtime (CLR)**
- **Functionality of CLR**
  - Exception Handling
  - Type Safety
  - Memory Management (using the Garbage Collector)
  - Security
  - Improved Performance
  - Language Independency
  - Platform Independency

.NET Framework Components

- **.NET Framework Class Library (FCL)**
- The .NET Framework Class Library (FCL) includes a huge collection of reusable classes, interfaces, and value types that easy, optimize and integrated with the CLR.
- It contains more than 7000+ classes and data types to read and write files, access databases, process XML, display a graphical user interface, draw graphics, create & consume of web services etc.
- The .NET Framework Class Library (FCL) contains code supporting for all the .NET technologies Like Windows Forms, ASP.NET, ADO.NET, Windows Workflow, Windows Communication Foundation & many more.

.NET Framework Components

- **Common Language Specification(CLS)**

- It defines a set of rules and restrictions that every language must follow which runs under .NET framework.
- This is done in such a way, that programs written in any language (.NET compliant) can interoperate with other languages.
- This also can take full advantage of inheritance, polymorphism, exceptions, and other features.
- The languages which follows these set of rules are said to be CLS Compliant Languages.
- In simple words, CLS enables Cross-Language Integration.

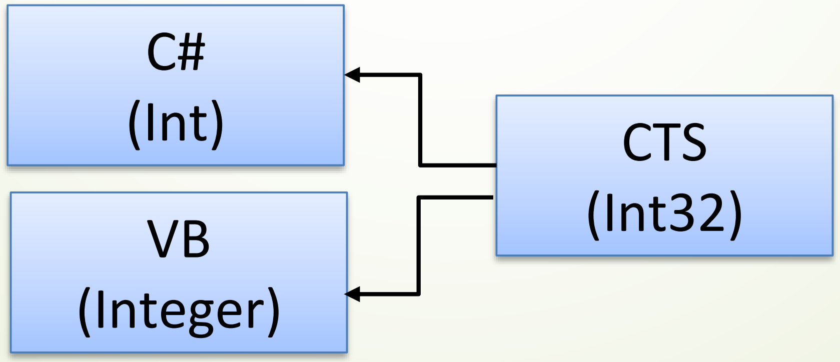
.NET Framework Components

- **Common Type System (CTS)**
- The Language interoperability, and .NET Class Framework, are not possible without all the language sharing the same data types.
- CTS are the mechanism by which, code written in one programming language can talk to code written in a different programming language.
- Suppose we declare an integer variable like “int i;” remains same in VB, VC++, C# and all other .NET compliant languages.
- It defines how types are declared, used and managed at the runtime.
- It facilitates cross-language integration, type safety and high performance code execution.
- It helps developers to develop applications in different languages.

.NET Framework Components

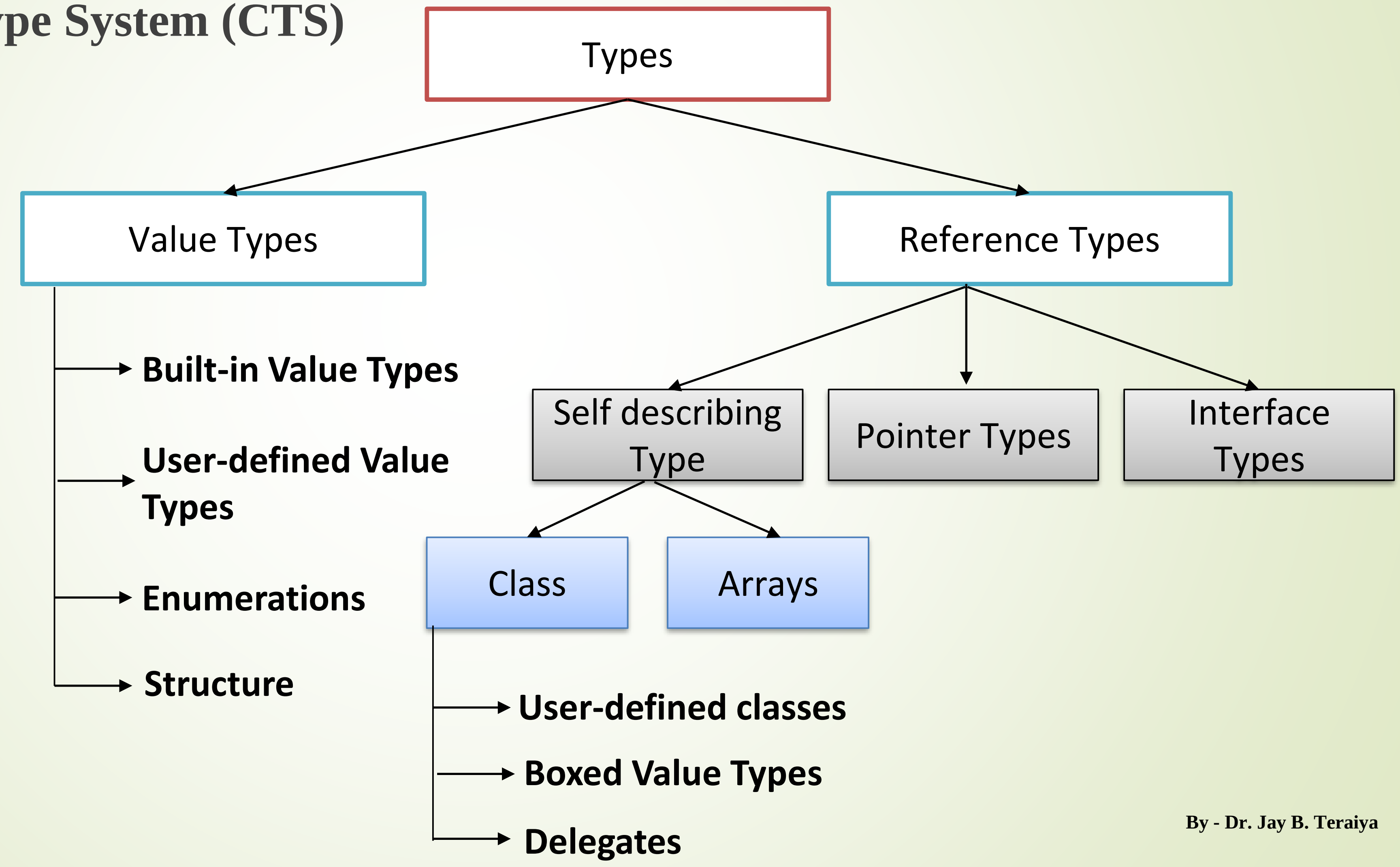
- **Common Type System (CTS)**

- The common type system (CTS) supports two general categories of data types:
  - **Value types :** It can be built-in, user-defined or enumerations types. (Boolean, Date, structs, and enums etc.)
  - **Reference types :** It can be self describing types, pointers types, or interface types. (Object)



.NET Framework Components

- **Common Type System (CTS)**



.NET Framework Components

- **MSIL = CIL = IL**

- Microsoft Intermediate Language (MSIL) is a language used as the output of a number of compilers (C#, VB, .NET, and so forth).
- Microsoft Intermediate Language (MSIL) is a CPU-independent set of instructions that can be efficiently converted to the native code.
- We can also call it as **Intermediate Language (IL)** or **Common Intermediate Language (CIL)**.
- When you compile a .NET Program, the CLR translates code into MSIL code that can be converted into CPU specific code with the help of JIT Compiler.

.NET Framework Components

- **MSIL = CIL = IL**

- The MSIL code includes instruction to load, initialize and invoke methods on objects.
- It also includes the instructions for various operations on program code, such as arithmetic and logical operations, control flow, direct memory access, exception handling etc.
- The program's source code is converted to MSIL code, which is equivalent to **assembly language for CPU**.
- The MSIL code is collected and assembled in the form of byte codes and is converted to a .NET assembly.
- The .NET assembly code is executed by the JIT Compiler to generate native code.
- The native code is executed by the computer's processor.

.NET Framework Components

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

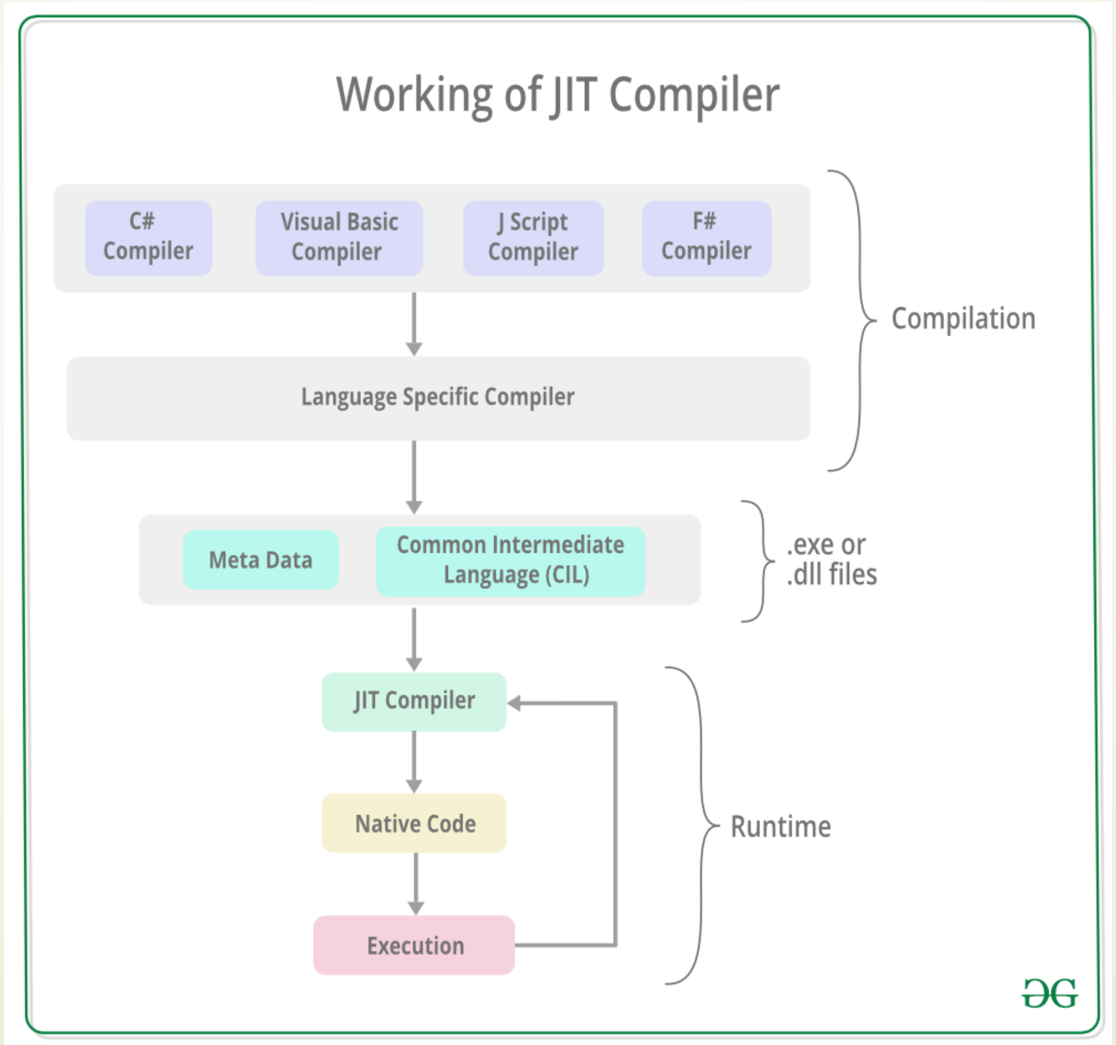
- **Just-In-Time compiler(JIT)** is a part of **Common Language Runtime (CLR)** in .NET which is responsible for managing the execution of .NET programs regardless of any .NET programming language.

- A language-specific compiler converts the source code to the intermediate language. This intermediate language is then converted into the machine code by the Just-In-Time (JIT) compiler. This machine code is specific to the computer environment that the JIT compiler runs on.

- **Working of JIT Compiler:** The JIT compiler is required to speed up the code execution and provide support for multiple platforms. Its working is given as follows:

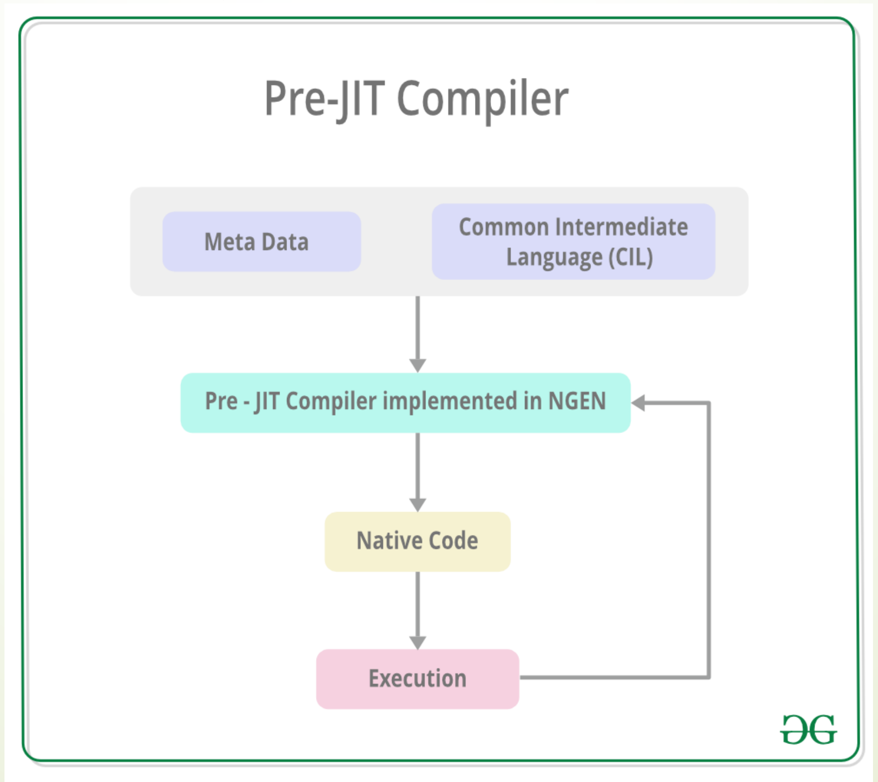
.NET Framework Components

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



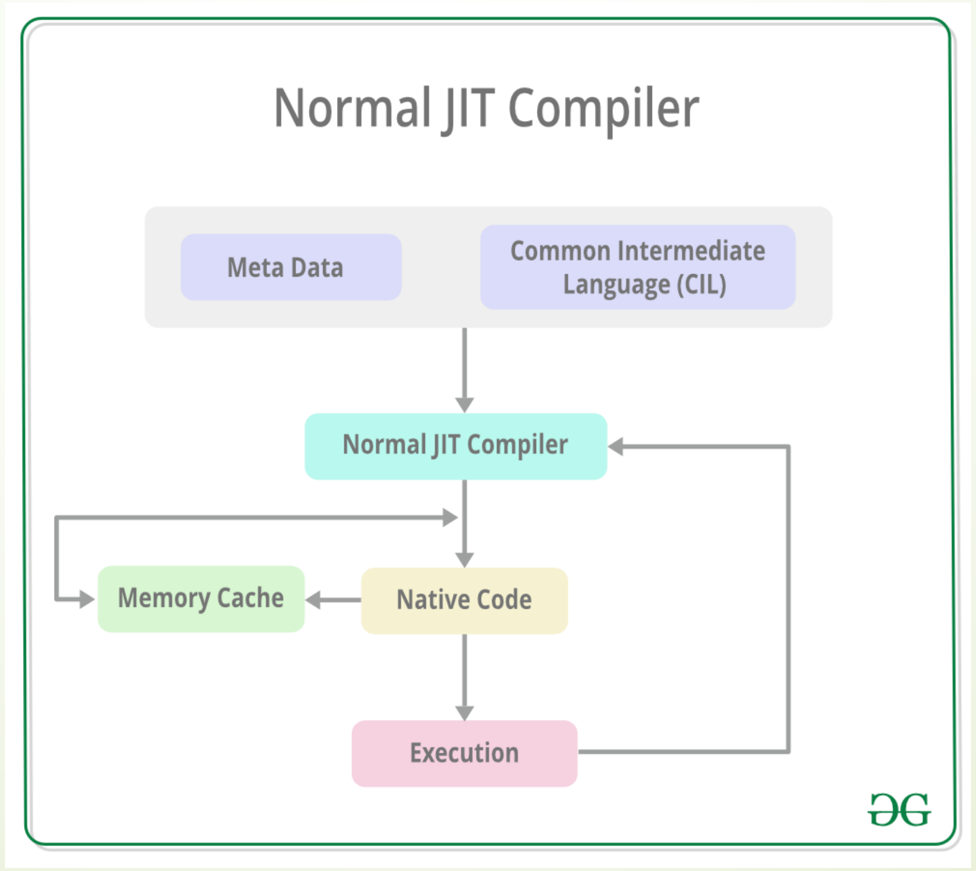
.NET Framework Components

- **Types of Just-In-Time Compiler:** There are 3 types of JIT compilers which are as follows:
- **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. This compilation process is performed at application deployment time. And this compiler is always implemented in the Ngen.exe (Native Image Generator).


Pre-JIT Compiler
Meta Data
Common Intermediate Language (CIL)
Pre - JIT Compiler implemented in NGEN
Native Code
Execution

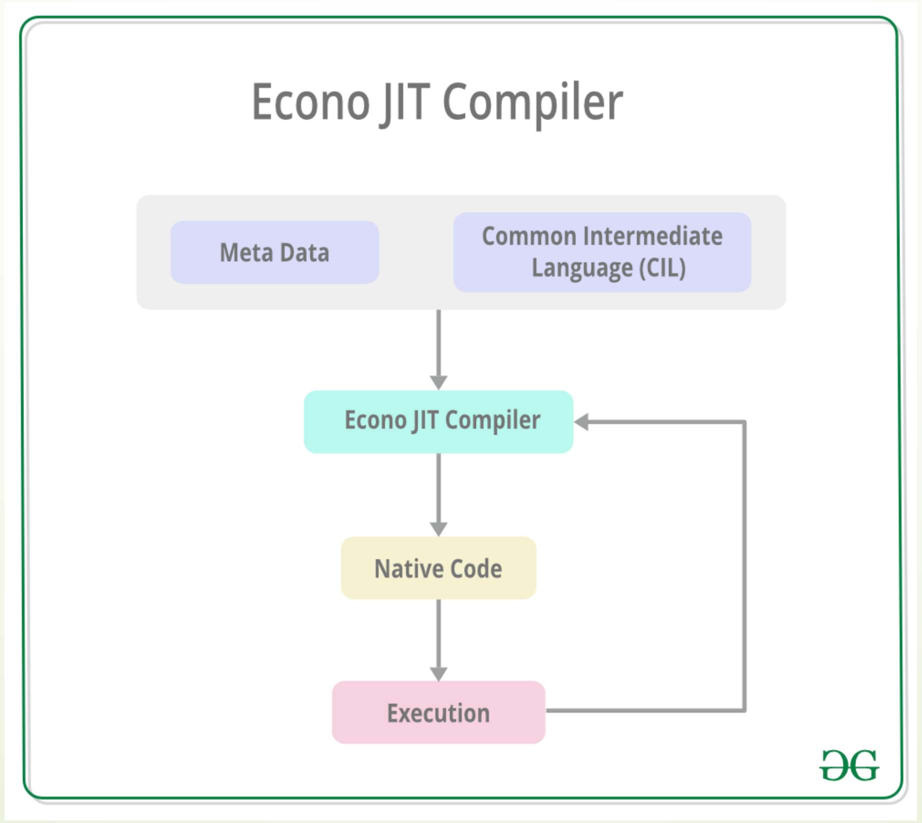
.NET Framework Components

- **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.



.NET Framework Components

- **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. This JIT compiler is obsolete starting from dotnet 2.0



.NET Framework Components

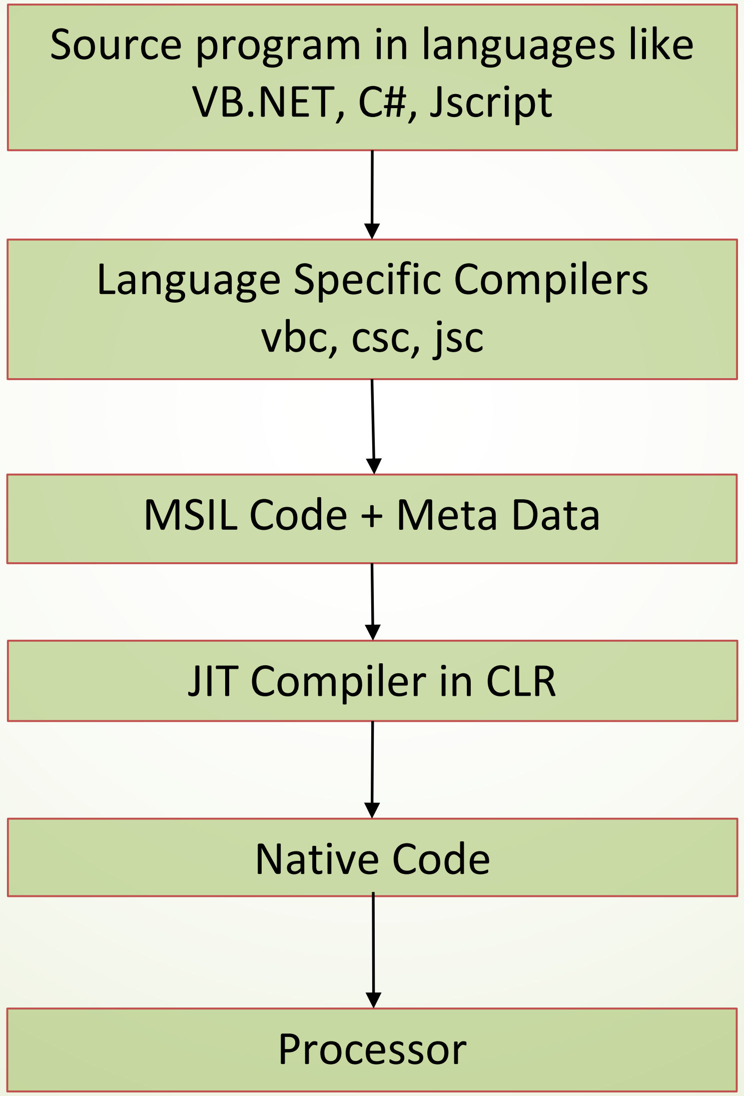
- **Advantages of JIT Compiler:**
  - The JIT compiler requires less memory usage as only the methods that are required at run-time are compiled into machine code by the JIT Compiler.
  - Page faults are reduced by using the JIT compiler as the methods required together are most probably in the same memory page.
  - Code optimization based on statistical analysis can be performed by the JIT compiler while the code is running.
- **Disadvantages of JIT compiler:**
  - The JIT compiler requires more startup time while the application is executed initially.
  - The cache memory is heavily used by the JIT compiler to store the source code methods that are required at run-time.

Managed Code and Unmanaged Code

- **Managed Code**
- Managed code is the code that is executed directly by the CLR.
- The applications that are created using managed code automatically have CLR services, such as type checking, security and automatic garbage collection.
- These CLR services help to provide platform and language independence to managed code applications.
- The CLR compiles the applications to Microsoft Intermediate Language (MSIL) and not the machine code.
- This MSIL along with the metadata that describes the attributes, classes, and methods of the code reside in assembly.
- The compilation takes place in managed execution environment, which assures the working of the code.

Managed Code and Unmanaged Code

- **Managed Code**



Source program in languages like VB.NET, C#, Jscript
Language Specific Compilers vbc, csc, jsc
MSIL Code + Meta Data
JIT Compiler in CLR
Native Code
Processor

Managed Code and Unmanaged Code

- **Managed Code**

- When you compile the code into managed environment, the compiler converts the source code into MSIL, which is CPU-independent.
- Compilation of source code into MSIL, generates metadata.
- MSIL must be converted into CPU – specific code by the JIT compiler, before the execution of the code.
- The runtime locates and extracts the metadata from the file during execution, while executing the application, a JIT compiler translates the MSIL into native code.
- After compiling the code is passed with the MSIL and metadata to check whether the code is safe, such as it should be able to access only those memory & locations which it is authorized to access.

Managed Code and Unmanaged Code

- **Unmanaged Code**

- Unmanaged code directly compiles to the machine code and runs on the machine where it has been compiled.
- It does not have services, such as security or memory management, which are provided by the runtime.
- If code is not security-prone, it can be directly interpreted by any user, which is harmful.
- Applications that do not run under the control of the CLR are said to be unmanaged.

Managed Code and Unmanaged Code

- **Managed v/s Unmanaged Code**

Managed Code	Unmanaged Code
Code Executed by CLR Instead of Operating System	Code which executed by operating system directly
Runtime provide services like GC, Type checking, Exception Handling.	Does not provide the services like GC, Type checking, Exception Handling taken care by the programmer
The code compiled by the language compiler into MSIL code	Code will be compiled into native code.

Namespace

- Namespace is a grouping of logically related identifiers, classes, types etc.
- Namespace is used to avoid conflicts with the elements of an unrelated code which have the same names.
- A namespace acts as a container—like a disk folder—for classes organized into groups usually based on functionality.
- All classes and types of .NET FCL are organized in namespaces.
- Implementing Namespaces in your own code is a good habit because it is likely to save you from problems later when you want to reuse some of your code.
- **Namespaces do not correspond to file or directory names.**

Namespace

- **How to Create Namespace?**

Syntax: namespace namespace_name {   }

```
namespace Demo
{
        class Student
        {
            public void Student_Details()
            {
                Console.WriteLine("This is Student class");
            }
        }
        class Subject
        {
            public void Subject_Details()
            {
                Console.WriteLine("This is Subject class");
            }
        }
}
```

Namespace

- **How to Use Namespace?**

```
using Demo;
using System; // Inbuilt Namespace
class Result
{
    Subject s;
        //Only class name, no need of namespace because we have used
        in the beginning of code
    System.Console.WriteLine("Hello");
        //Full qualifier name because we have not included namespace.
     Console.WriteLine("Hello");
        // Console class belongs to System Namespace.
}
```

Namespace

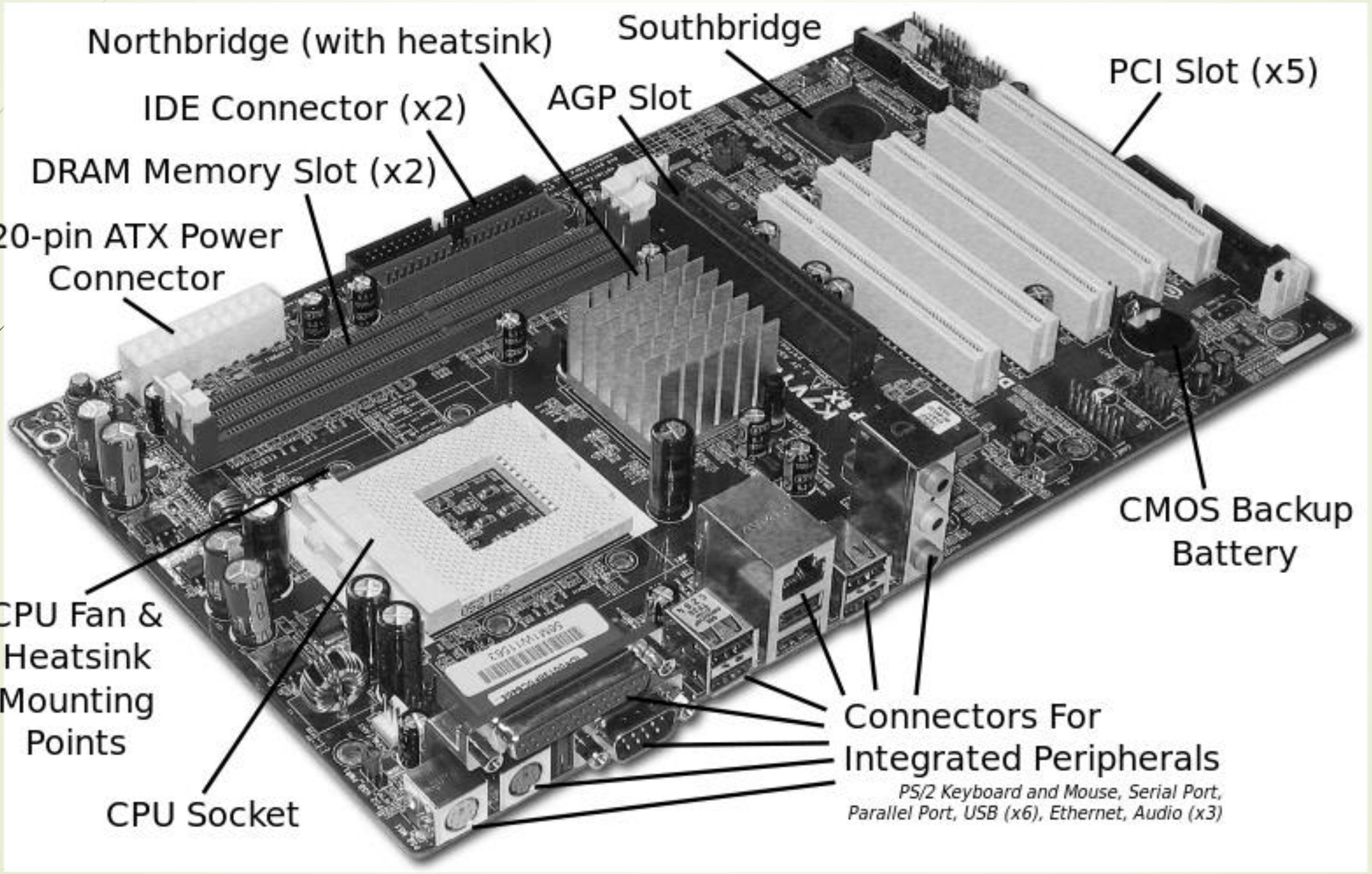
- **Alias for Namespace** Syntax: using alias-name = namespace;
-

```
using Sys = System; // Alias Namespace

namespace Demo
{
    class Student
        {
           public static void Main()
           {
              Sys.Console.WriteLine("This is Student class");
              Sys.Console.WriteLine("Hello");
                // Console class belongs to System Namespace.
           }
        }
}
```

Namespace

Namespace	Description
System.Data	Includes classes which lets us handle data from data sources.
System.Drawing	Provides access to drawing methods.
System.IO	Includes classes for data access with Files.
System.NET	Provides interface to protocols used on the internet.
System.Security	Includes classes to support the structure of common language runtime security system.
System.Web	Includes classes and interfaces that support browser-server communication.
System.XML	Includes classes for XML support.

.NET Assembly


Northbridge (with heatsink)
Southbridge
PCI Slot (x5)
IDE Connector (x2)
AGP Slot
DRAM Memory Slot (x2)
20-pin ATX Power Connector
CMOS Backup Battery
CPU Fan & Heatsink Mounting Points
Connectors For Integrated Peripherals
PS/2 Keyboard and Mouse, Serial Port, Parallel Port, USB (x6), Ethernet, Audio (x3)
CPU Socket

.NET Assembly and MetaData

- Assembly is the smallest unit of deployment of a .NET applications. It can be a dll or an exe.
- There are mainly two types to it:
- **Private Assembly:**
- The dll or exe which is sole property of one application only. It is generally stored in application root folder.
- **Public/Shared Assembly:**
- It is a dll which can be used by multiple applications at a time.
- A shared assembly is stored in **GAC i.e Global Assembly Cache**.
- GAC is simply C:\Windows\Assembly folder where you can find the public assemblies/dlls of all the softwares installed in your PC.
- There is also a third and least known type of an assembly**: Satellite Assembly.**
- A Satellite Assembly contains only static objects like **images, text files and other non-executable** files required by the application.

.NET Assembly and MetaData

- **Assembly contains four major parts.**
- **Manifest:**
- Every assembly file contains information about itself. This information is called as Assembly Manifest.
- The information includes version information, list of files packed, and definition of types, security permissions, version control and metadata.
- **Metadata:**
- Metadata is binary information describing about your program that is stored either in a **CLR portable executable (PE) file or in memory.**
- **MSIL code:**
- Containing business logics and also an Intermediate version of program.
- **Set of Resource:**
- Resources of an assembly like icons, text files, image files etc.

Garbage Collection

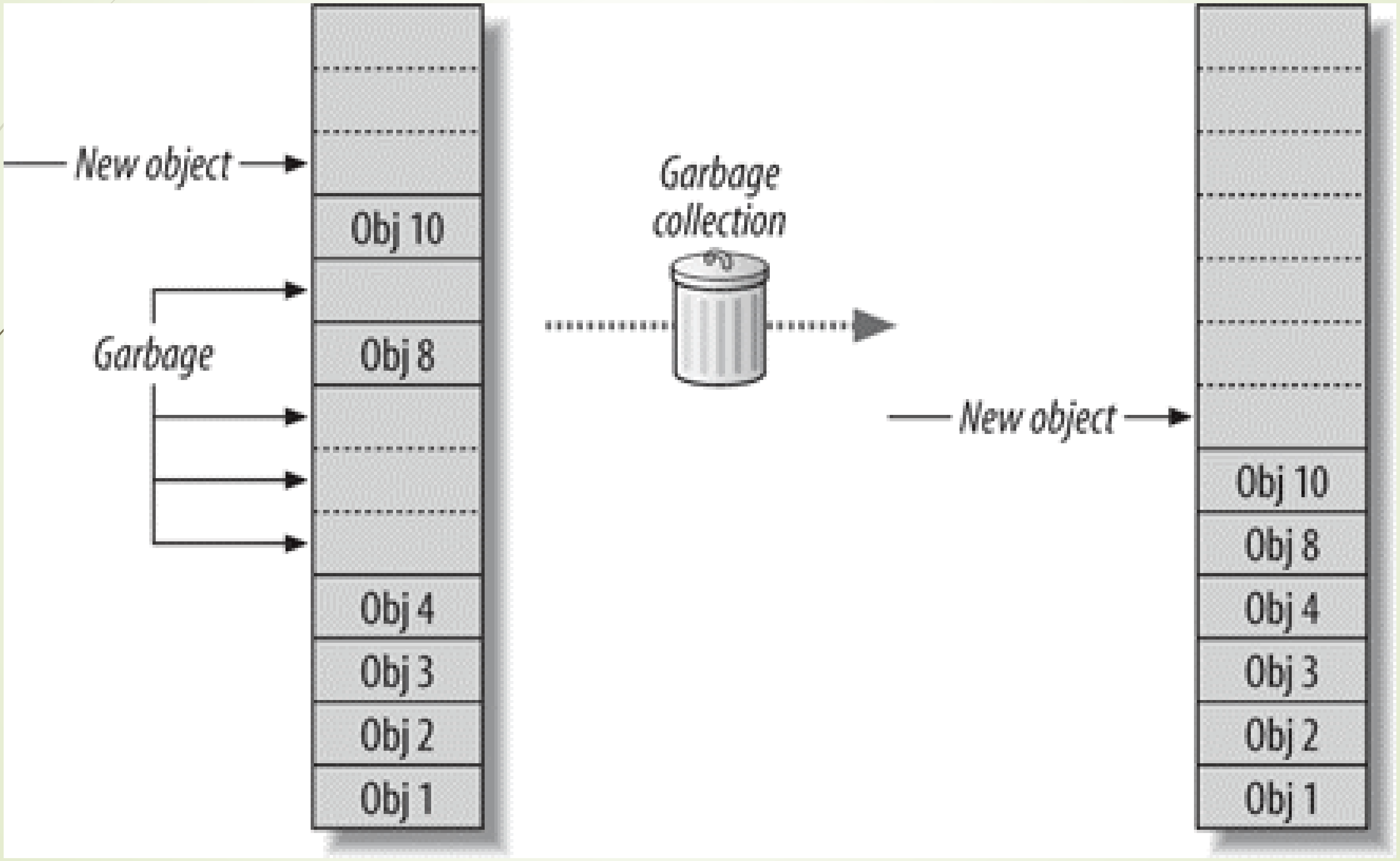
- The .NET Framework provides a new mechanism for releasing unreferenced objects from the memory (that is no longer needed objects in the program), is called Garbage Collection (GC).

- When a program creates an object, the object takes up the memory.

- Later when the program has no more references to that object, the object's memory becomes unreachable, but it is not immediately freed.

- The Garbage Collection checks to see if there are any objects in the heap that are no longer being used by the application.

Garbage Collection



Garbage Collection

- If such objects exist, then the memory used by these objects can be reclaimed.
- So these unreferenced objects should be removed from memory, then the other new objects you create can find a place in the Heap.
- This releasing of unreferenced objects is happening automatically in .NET languages by the Garbage Collector (GC).
- In .NET languages there is a facility that we can call Garbage Collector (GC) explicitly in the program by calling **System.GC.Collect()**.

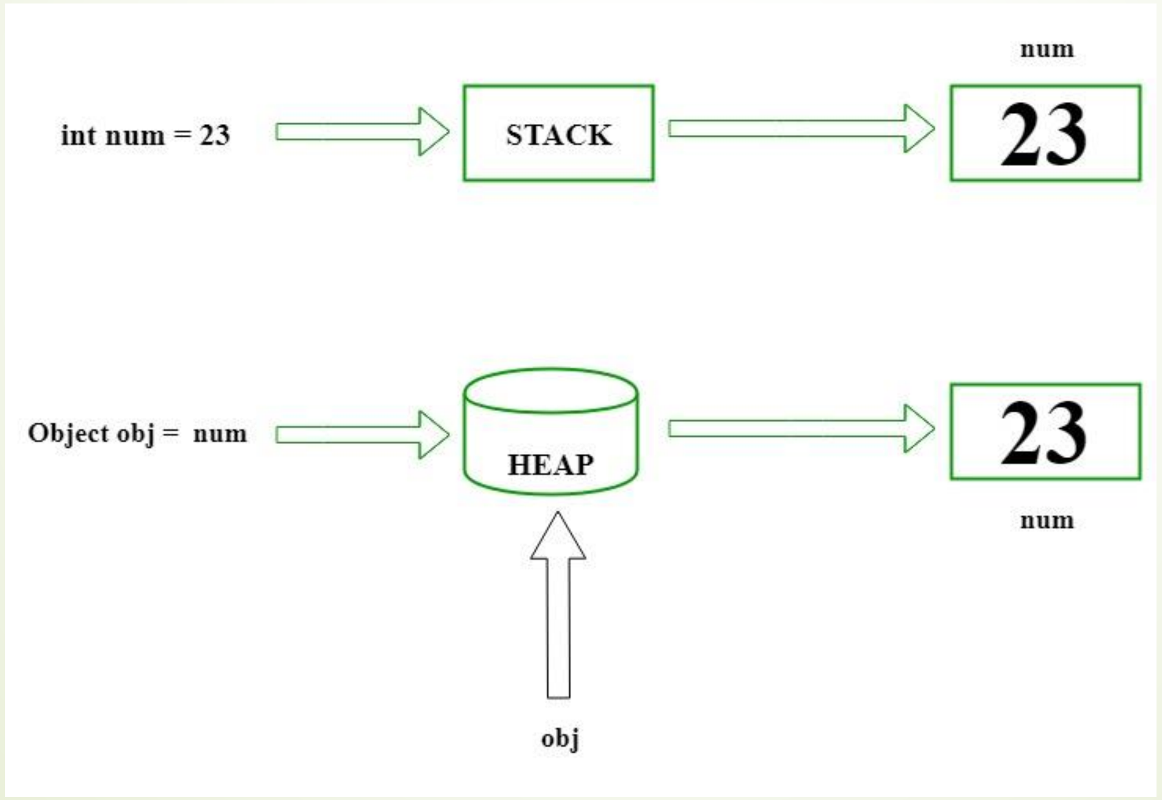
- **Advantage of using Garbage Collector**
  - Allow us to develop an application without having worry to free memory.
  - Allocates memory for objects efficiently on the managed heap.
  - Reclaims the memory for no longer used objects and keeps the free memory for future allocations.
  - Provides memory safety by making sure that an object cannot use the content of another object.

Boxing and UnBoxing

- Boxing and unboxing are important concepts in C#. The C# Type System contains three data types: Value Types (int, char, etc), Reference Types (object) and Pointer Types.
- Basically, Boxing converts a **Value Type variable into a Reference Type variable, and Unboxing achieves the vice-versa**. Boxing and Unboxing enable a unified view of the type system in which a value of any type can be treated as an object.
- **Boxing In C#**
  - The process of converting a Value Type variable (char, int etc.) to a Reference Type variable (object) is called Boxing.
  - Boxing is an implicit conversion process in which object type (super type) is used.
  - Value Type variables are always stored in **Stack memory**, while Reference Type variables are stored in **Heap memory**.

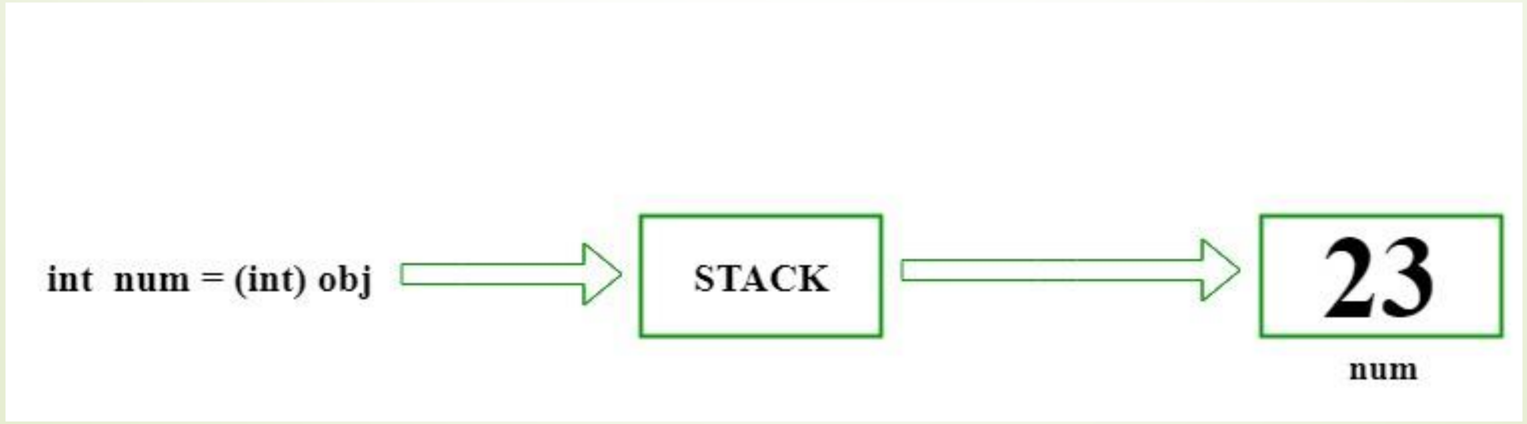
Boxing and UnBoxing

- int num = 23; // 23 will assigned to num
- Object Obj = num; // Boxing



Boxing and UnBoxing

- Unboxing In C#
- The process of converting a Reference Type variable into a Value Type variable is known as Unboxing. It is an explicit conversion process.
- Example :
  - int num = 23;   // value type is int and assigned value 23
  - Object Obj = num;   // Boxing
  - int i = (int)Obj;   // Unboxing



Operators in C#

- Operators are symbols that are used to perform operations on operands. Operands may be variables and/or constants.

- For example, in 2+3, + is an operator that is used to carry out addition operation, while 2 and 3 are operands.

- Operators are used to manipulate variables and values in a program. C# supports a number of operators that are classified based on the type of operations they perform.
- Basic Assignment Operator

- **Basic assignment operator (=)** is used to assign values to variables. For example,

```
double x;
x = 50.05;
```

Operators in C#

- **Arithmetic Operators** - Arithmetic operators are used to perform arithmetic operations such as addition, subtraction, multiplication, division, etc.

C# Arithmetic Operators		
Operator	Operator Name	Example
+	Addition Operator	6 + 3 evaluates to 9
-	Subtraction Operator	10 - 6 evaluates to 4
*	Multiplication Operator	4 * 2 evaluates to 8
/	Division Operator	10 / 5 evaluates to 2
%	Modulo Operator (Remainder)	16 % 3 evaluates to 1

Operators in C#

- **Relational Operators** - Relational operators are used to check the relationship between two operands. If the relationship is **true** the result will be **true**, otherwise it will result in **false.** Relational operators are used in decision making and loops.

C# Relational Operators		
Operator	**Operator Name**	**Example**
==	Equal to	6 == 4 evaluates to false
>	Greater than	3 > -1 evaluates to true
<	Less than	5 < 3 evaluates to false
>=	Greater than or equal to	4 >= 4 evaluates to true
<=	Less than or equal to	5 <= 3 evaluates to false
!=	Not equal to	10 != 2 evaluates to true

Operators in C#

- **Logical Operators** - Logical operators are used to perform logical operation such as **AND & OR**. Logical operators operates on Boolean expressions (true and false) and returns Boolean values. Logical operators are used in decision making and loops.

C# Logical operators			
Operand 1	**Operand 2**	**OR (||)**	**AND (&&)**
true	true	true	true
true	false	true	false
false	true	true	false
false	false	false	false

Operators in C#

- **Unary Operators** - Unlike other operators, the unary operators operates on a single operand.

C# unary operators		
Operator	**Operator Name**	**Description**
+	Unary Plus	Leaves the sign of operand as it is
-	Unary Minus	Inverts the sign of operand
++	Increment	Increment value by 1
--	Decrement	Decrement value by 1
!	Logical Negation (Not)	Inverts the value of a boolean

Operators in C#

- **Unary Operators** –
- The increment (++) and decrement (--) operators can be used as prefix and postfix. If used as prefix, the change in value of variable is seen on the same line and if used as postfix, the change in value of variable is seen on the next line. This will be clear by the example below.

```
int number = 10;
Console.WriteLine((number++));
Console.WriteLine((number));
Console.WriteLine((++number));
Console.WriteLine((number));
```

- When we run the above program output will be
  - 10
  - 11
  - 12
  - 12

Operators in C#

- **Ternary Operator**– The ternary operator **“? :”** operates on three operands. It is a shorthand for if-then-else statement. Ternary operator can be used as follows:

```
variable = Condition? Expression1 : Expression2;
```

- The ternary operator works as follows:
  - If the expression stated by Condition is true, the result of Expression1 is assigned to variable.
  - If it is false, the result of Expression2 is assigned to variable.

- Example of Ternary Operator

```
int number = 10;
string result;
result = (number % 2 == 0)? "Even Number" : "Odd Number";
Console.WriteLine("{0} is {1}", number, result);
```

Operators in C#

- **Bitwise and Bit Shift Operators**– Bitwise and bit shift operators are used to perform bit manipulation operations.

C# Bitwise and Bit Shift operators	
Operator	**Operator Name**
~	Bitwise Complement
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive OR
<<	Bitwise Left Shift
>>	Bitwise Right Shift

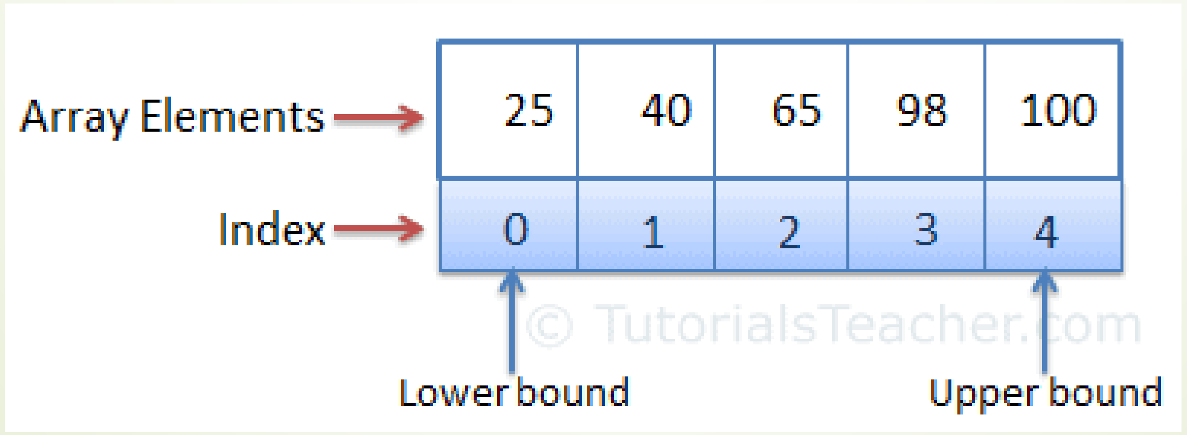
Operators in C#

- **Compound Assignment Operators**–

C# Compound Assignment Operators			
Operator	**Operator Name**	**Example**	**Equivalent To**
+=	Addition Assignment	x += 5	x = x + 5
-=	Subtraction Assignment	x -= 5	x = x - 5
*=	Multiplication Assignment	x *= 5	x = x * 5
/=	Division Assignment	x /= 5	x = x / 5
%=	Modulo Assignment	x %= 5	x = x % 5
&=	Bitwise AND Assignment	x &= 5	x = x & 5
|=	Bitwise OR Assignment	x |= 5	x = x | 5
^=	Bitwise XOR Assignment	x ^= 5	x = x ^ 5
<<=	Left Shift Assignment	x <<= 5	x = x << 5
>>=	Right Shift Assignment	x >>= 5	x = x >> 5

Array

- A variable is used to store a literal value, whereas an array is used to store multiple literal values.
- An array is the data structure that stores a fixed number of literal values (elements) of the same data type. Array elements are stored contiguously in the memory.
- In C#, an array can be of three types: **single-dimensional, multidimensional, and jagged array**.
- The following figure illustrates an array representation.



Array

- **One Dimensional Array**
- An array can be declared using by specifying the type of its elements with square brackets.

```
int[] evenNums;  // integer array
string[] cities; // string array
```

- The following declares and adds values into an array in a single statement.

```
int[] evenNums = new int[5]{ 2, 4, 6, 8, 10 };
string[] cities = new string[3]{ "Mumbai", "London", "New York" };
```

- Above, evenNums array can store up to five integers. The number 5 in the square brackets new int[5] specifies the size of an array. In the same way, the size of cities array is three. Array elements are added in a comma-separated list inside curly braces { }.

Array

- **One Dimensional Array**
- If you are adding array elements at the time of declaration, then size is optional. The compiler will infer its size based on the number of elements inside curly braces, as shown below.
- Example: Short Syntax of Array Declaration
  - int[] evenNums = { 2, 4, 6, 8, 10};
  - string[] cities = { "Mumbai", "London", "New York" }

Array

- **One Dimensional Array**

- The following example demonstrate invalid array declarations.
- Example: Invalid Array Creation

```
//must specify the size
int[] evenNums = new int[];

//number of elements must be equal to the specified size
int[] evenNums = new int[5] { 2, 4 };

//cannot use var with array initializer
var evenNums = { 2, 4, 6, 8, 10};
```

Array

- **One Dimensional Array**

- It is not necessary to declare and initialize an array in a single statement. You can first declare an array then initialize it later on using the new operator.

- Example: Late Initialization

```
int[] evenNums;
evenNums = new int[5];
// or
evenNums = new int[]{ 2, 4, 6, 8, 10 };
```

Array

- **One Dimensional Array**
- Array elements can be accessed using an index. An index is a number associated with each array element, starting with index 0 and ending with array size - 1.
- The following example add/update and retrieve array elements using indexes.
- Example: Access Array Elements using Indexes

```
int[] evenNums = new int[5];
evenNums[0] = 2;
evenNums[1] = 4;
//evenNums[6] = 12;  //Throws run-time exception IndexOutOfRange

Console.WriteLine(evenNums[0]);  //prints 2
Console.WriteLine(evenNums[1]);  //prints 4
```

- Note that trying to add more elements than its specified size will result in IndexOutOfRangeException.

Array

- **One Dimensional Array**

- Accessing Array using for Loop
- Use the for loop to access array elements. Use the length property of an array in conditional expression of the for loop.

- Example: Accessing Array Elements using for Loop

```
int[] evenNums = { 2, 4, 6, 8, 10 };
for(int i = 0; i < evenNums.Length; i++)
Console.WriteLine(evenNums[i]);

for(int i = 0; i < evenNums.Length; i++)
evenNums[i] = evenNums[i] + 10;  // update the value of each element by 10
```

Array

- **One Dimensional Array**

- Accessing Array using foreach Loop

- Use foreach loop to read values of an array elements without using index.
- Example: Accessing Array using foreach Loop

```
int[] evenNums = { 2, 4, 6, 8, 10};
string[] cities = { "Mumbai", "London", "New York" };

foreach(int item in evenNums)
Console.WriteLine(item);

foreach(string city in cities)
Console.WriteLine(city);
```

Array

- **One Dimensional Array**
- Passing Array as Argument

```
public static void Main()
{
     int[] nums = { 1, 2, 3, 4, 5 };
    UpdateArray(nums);
    foreach(var item in nums)
    Console.WriteLine(item);
}
public static void UpdateArray(int[] arr)
{
     for(int i = 0; i < arr.Length; i++)
         arr[i] = arr[i] + 10;
}
```

Array

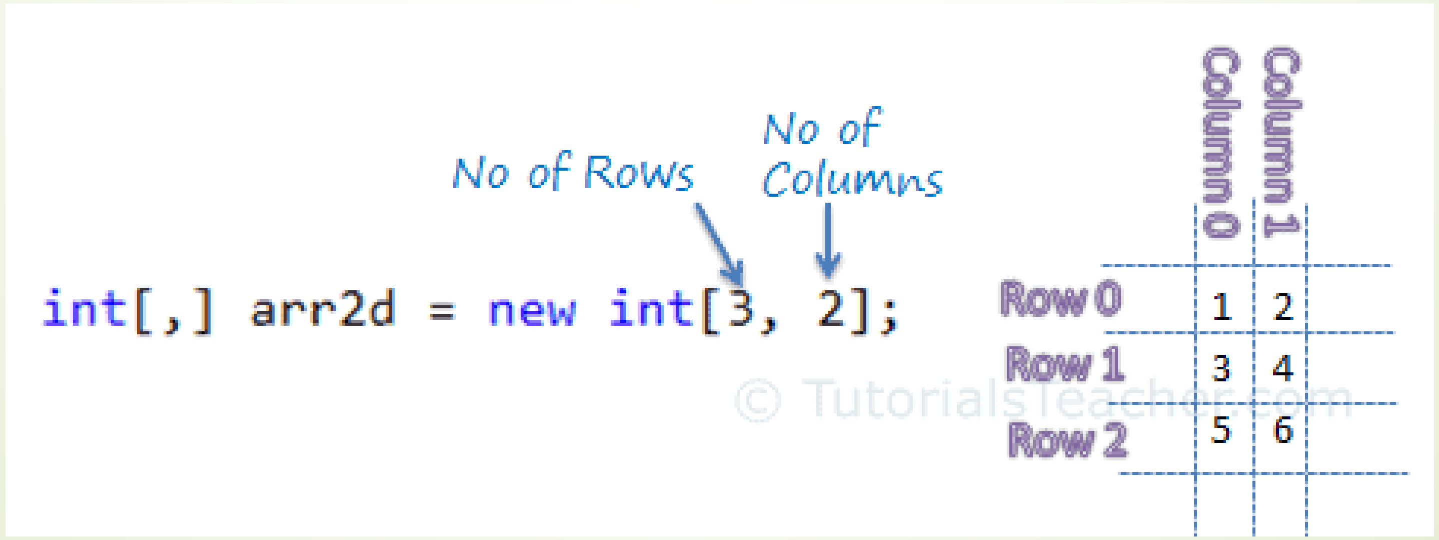
- **Multidimensional Dimensional Array**
- C# supports multidimensional arrays up to **32 dimensions**. The multidimensional array can be declared by adding commas in the square brackets. For example, [,] declares two-dimensional array, [, ,] declares three-dimensional array, [, , ,] declares four-dimensional array, and so on. So, in a multidimensional array, no of commas = No of Dimensions - 1.
- The following declares multidimensional arrays.

```
int[,] arr2d; // two-dimensional array
int[, ,] arr3d; // three-dimensional array
int[, , ,] arr4d ; // four-dimensional array
int[, , , ,] arr5d; // five-dimensional array
```

Array

- **Multidimensional Dimensional Array**
- Let's understand the two-dimensional array. The following initializes the two-dimensional array.

```
        int[,] arr2d = new int[3,2]{{1, 2},{3, 4}, {5, 6} };
  // or
        int[,] arr2d = {{1, 2}, {3, 4}, {5, 6} };
```



© TutorialsTeacher.com

Array

- **C# Jagged Arrays: An Array of Array**

- A jagged array is an array of array. Jagged arrays store arrays instead of literal values.
- A jagged array is initialized with two square brackets [][]. The first bracket specifies the size of an array, and the second bracket specifies the dimensions of the array which is going to be stored.
- The following example declares jagged arrays.

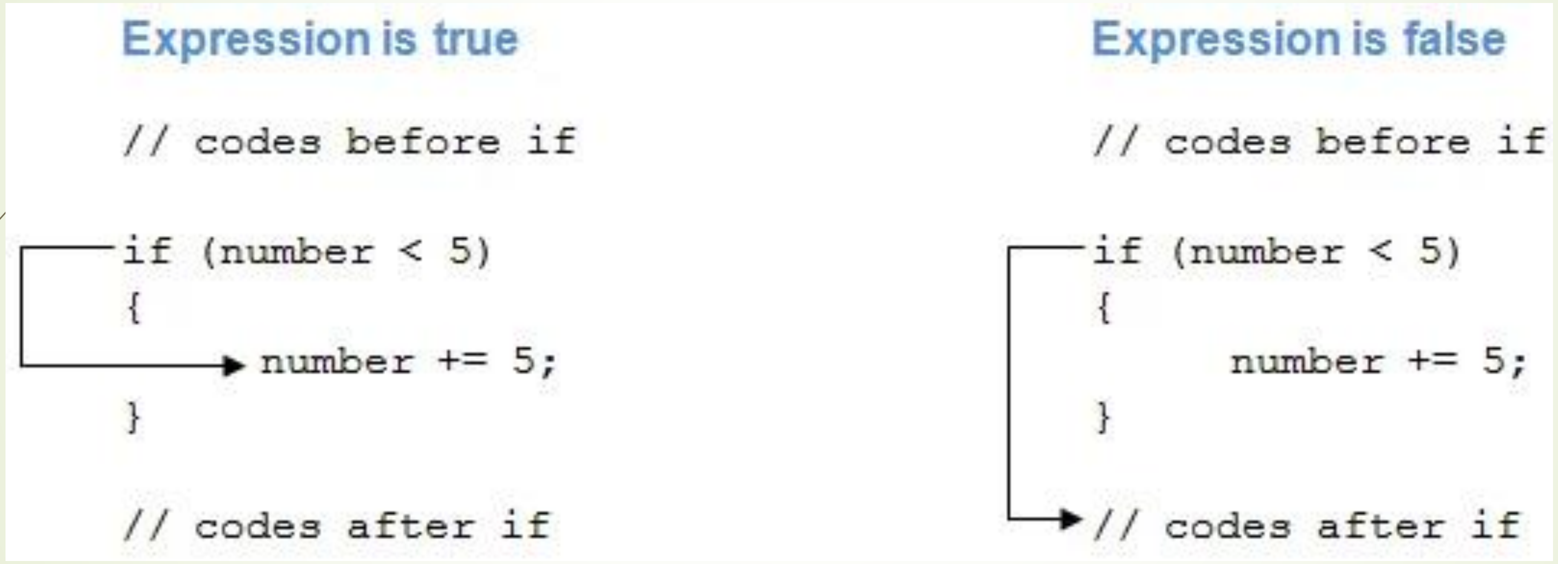
```
int[][] jArray1 = new int[2][]; // can include two single-dimensional arrays
int[][,] jArray2 = new int[3][,]; // can include three two-dimensional arrays
```

- Example: Jagged Array

```
int[][] jArray = new int[2][];
jArray[0] = new int[3]{1, 2, 3};
jArray[1] = new int[4]{4, 5, 6, 7 };
```

Decisions (if types and switch case)

- **C# if statement** will execute a block of code if the given condition is true. The syntax of if statement in C# is:

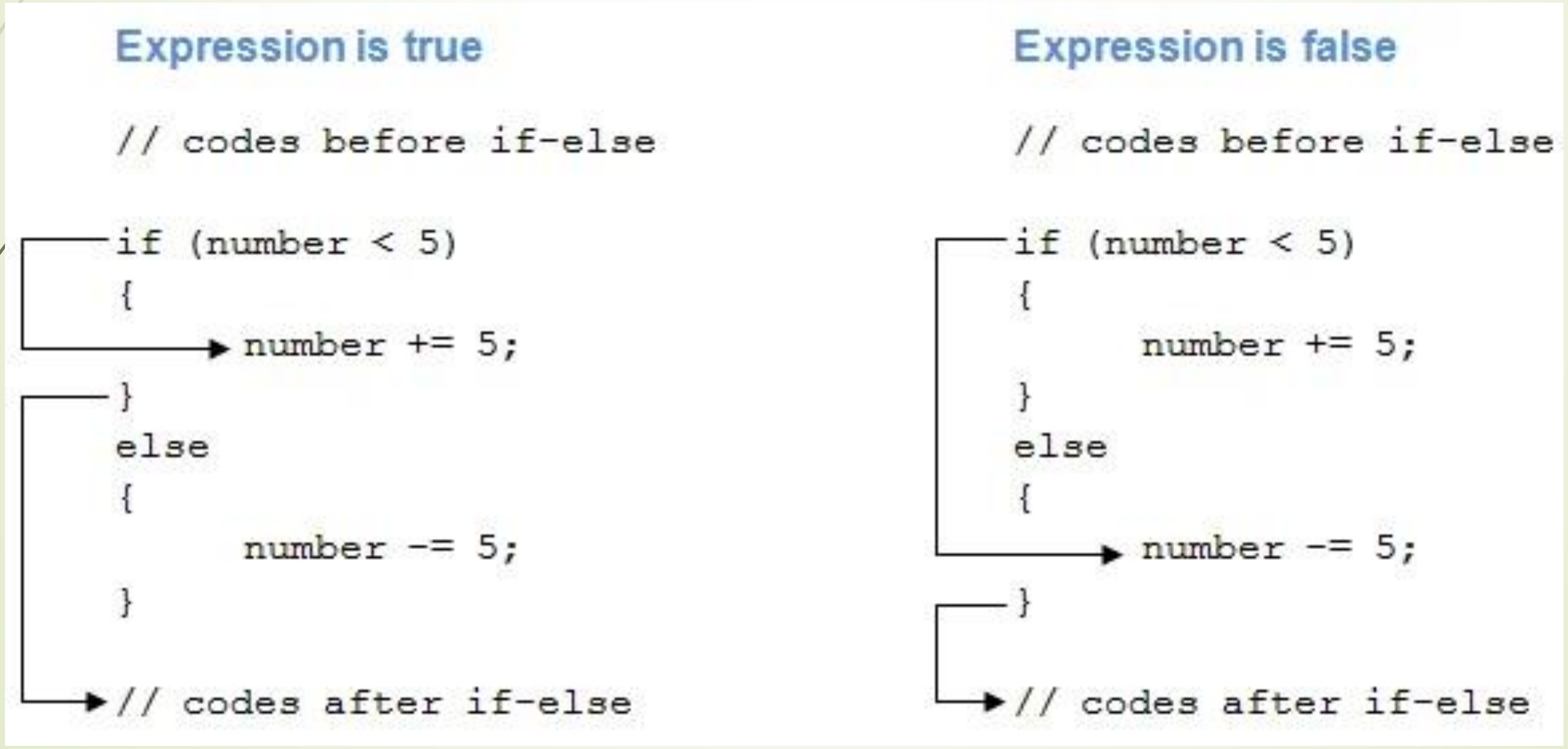


```
// codes before if

if (number < 5)
{
    number += 5;
}

// codes after if
```

- The boolean-expression will return either true or false.
- If the boolean-expression returns true, the statements inside the body of if (inside {...}) will be executed.
- If the boolean-expression returns false, the statements inside the body of if will be ignored.

Decisions (if types and switch case)

- **C# if...else -** The if statement in C# may have an optional else statement. The block of code inside the else statement will be executed if the expression is evaluated to false.



Expression is true

```
// codes before if-else

if (number < 5)
{
      number += 5;
}
else
{
      number -= 5;
}

// codes after if-else
```

Expression is false

```
// codes before if-else

if (number < 5)
{
      number += 5;
}
else
{
      number -= 5;
}

// codes after if-else
```

Decisions (if types and switch case)

- **C# if...else if Statement -** When we have only one condition to test, if and if-else statement works fine. But what if we have a multiple condition to test and execute one of the many block of code. For such case, we can use if..else if statement in C#. The syntax for if...else if statement is:

```
if (boolean-expression-1)
{       // statements executed if boolean-expression-1 is true
}
else if (boolean-expression-2)
{       // statements executed if boolean-expression-2 is true
}
else if (boolean-expression-3)
{       // statements executed if boolean-expression-3 is true
}
        ……………………..
else
{       // statements executed if all above expressions are false
}
```

Decisions (if types and switch case)

- **Nested if...else Statement** - An if...else statement can exist within another if...else statement. Such statements are called nested if...else statement. The general structure of nested if…else statement is:

```
if (boolean-expression)
{
        if (nested-expression-1)
        {
                // code to be executed
        }
        else
        {
                // code to be executed
        }
}
else
{       -----------
}
```

Decisions (if types and switch case)

- **C# - Switch Statement**
- The switch statement can be used instead of if else statement when you want to test a variable against three or more conditions.
- The following is the general syntax of the switch statement.

```
switch(match expression/variable)
{
        case constant-value:  statement(s) to be executed;
                              break;
        default:    statement(s) to be executed;
                    break;
}
```

Decisions (if types and switch case)

```
int x = 10;
switch (x)
{
        case 5:
             Console.WriteLine("Value of x is 5");
             break;
        case 10:
             Console.WriteLine("Value of x is 10");
             break;
         case 15:
              Console.WriteLine("Value of x is 15");
               break;
        default:
                Console.WriteLine("Unknown value");
                break;
}
```

Decisions (if types and switch case)

- The switch statement is an alternative to if else statement.
- The switch statement tests a match expression/variable against a set of constants specified as cases.
- The switch case must include break keyword to exit a case.
- The switch can include one optional default label, which will be executed when no case executed.
- The switch statement can include any non-null expression that returns a value of type**: char, string, bool, int, or enum.**

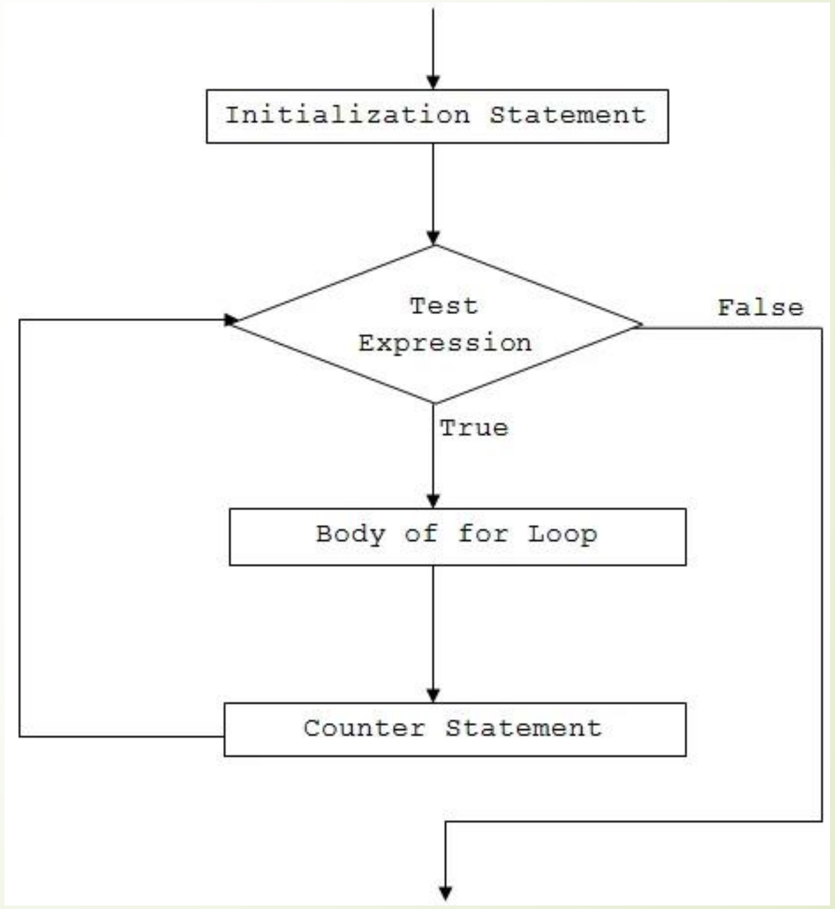
Loops (for)

- The for keyword indicates a loop in C#. The for loop executes a block of statements repeatedly until the specified condition returns false.

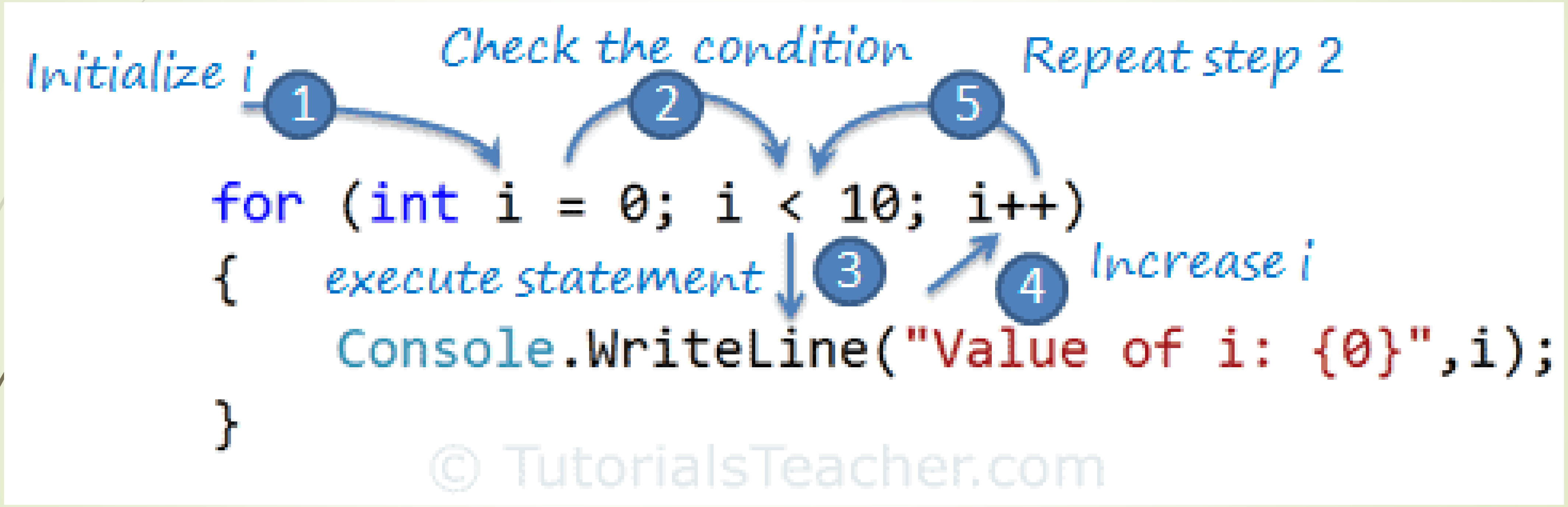
```
for (initializer; condition; iterator)
{
 //code block
}
```

- Example of for loop

```
for(int i = 0; i < 10; i++)
{
    Console.WriteLine("Value of i: {0}", i);
}
```



Loops (for)



```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine("Value of i: {0}",i);
}
```

© TutorialsTeacher.com

Loops (for)

```
int i = 0;
for(;;)
{
        if (i < 10)
        {
                Console.WriteLine("Value of i: {0}", i);
                i++;
        }
        else
        {
                break;
        }
}
```

```
for(int i = 10; i > 0; i--)
{
    Console.WriteLine("Value of i: {0}", i);
}
```

```
for (int i = 0, j = 0; i+j < 5; i++, j++)
{
    Console.WriteLine("Value of i: {0}, J: {1} ", i,j);
}
```

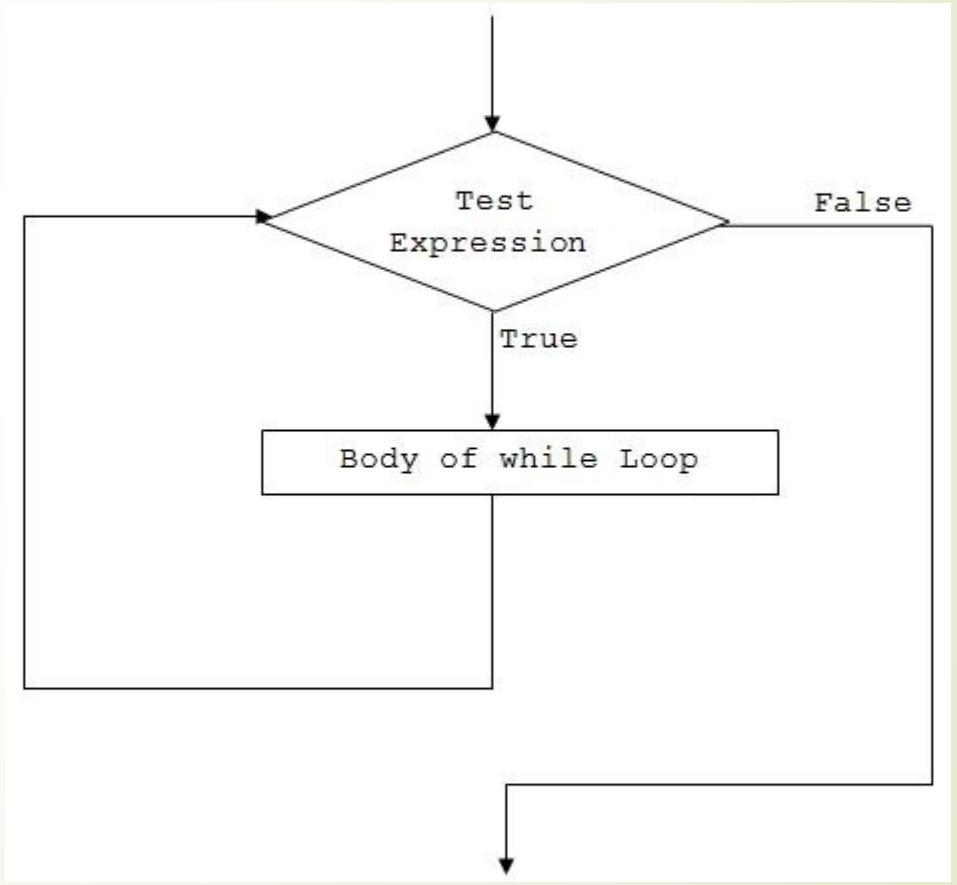
Loops (while)

- C# provides the while loop to repeatedly execute a block of code as long as the specified condition returns true.

```
while(condition)
{
  //code block
}
```

- Example: C# while Loop

```
int i = 0; // initialization
while (i < 10) // condition
{
        Console.WriteLine("i = {0}", i);
        i++; // increment
}
```



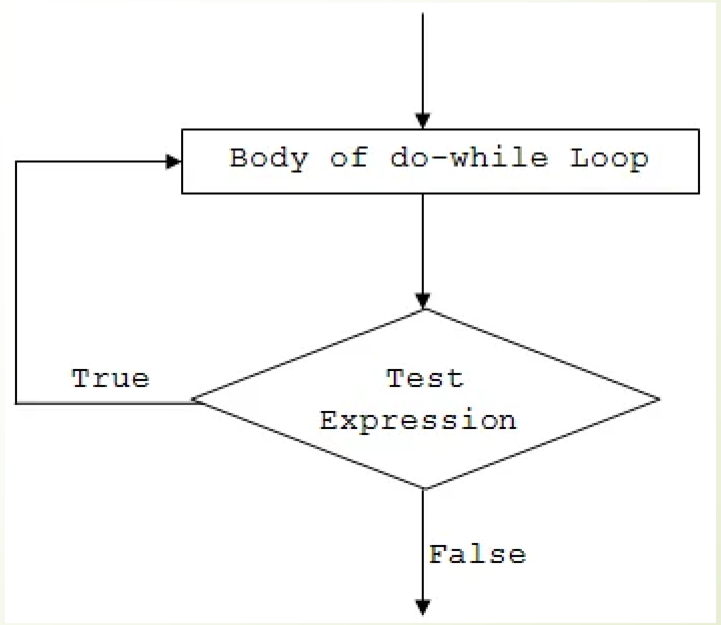
Loops (do…while)

- The do and while keyword is used to create a do...while loop. It is similar to a while loop, however there is a major difference between them.

- In while loop, the condition is checked before the body is executed. It is the exact opposite in do...while loop, i.e. condition is checked after the body is executed.

- The syntax for do...while loop is:

```
do
{
        // body of do while loop
} while (test-expression);
```



Loops (do…while)

- **Infinite while loop**

```
while (true)
{
        // body of while loop
}
```

- **Infinite do...while loop**

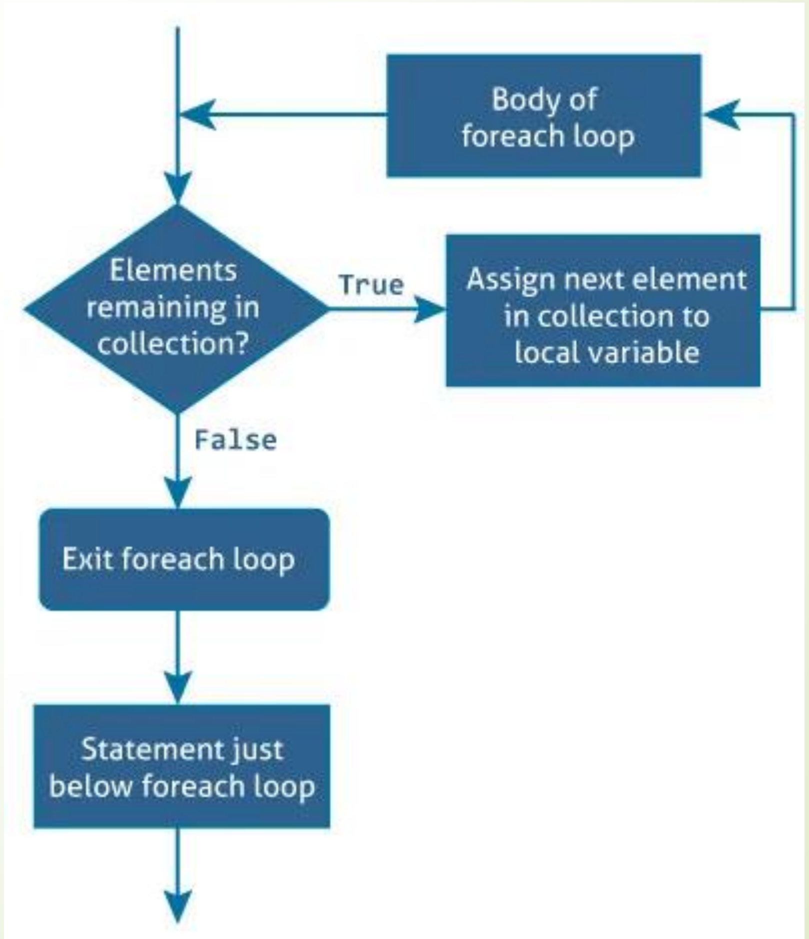
```
do
{
        // body of while loop
} while (true);
```

Loops (foreach)

- C# provides an easy to use and more readable alternative to for loop, the foreach loop when working with arrays and collections to iterate through the items of arrays/collections. The foreach loop iterates through each item, hence called foreach loop.

- Syntax of foreach loop

```
foreach (datatype var in iterable-item)
{
        // body of foreach loop
}
```


Body of foreach loop
Elements remaining in collection?
True
Assign next element in collection to local variable
False
Exit foreach loop
Statement just below foreach loop

Loops (foreach)

```
using System;
class ForEachLoop
{
        public static void Main(string[] args)
        {
            char[] myArray = {'H','e','l','l','o'};

            foreach(char ch in myArray)
            {
                Console.WriteLine(ch);
            }
        }
    }
```

Collection

- **ArrayList -** In C#, the ArrayList is a non-generic collection of objects whose size increases dynamically. It is the same **as Array except that its size increases dynamically**. An ArrayList can be used to add unknown data where you don't know the types and the size of the data.
- **Hashtable** - The Hashtable is a non-generic collection that stores key-value pairs, similar to generic collection. It optimizes lookups by computing the hash code of each key and stores it in a different bucket internally and then matches the hash code of the specified key at the time of accessing values.
- **Stack** - is a special type of collection that stores elements in LIFO style (Last In First Out). Stack is useful to store temporary data in LIFO style, and you might want to delete an element after retrieving its value.
- **Queue** - is a special type of collection that stores the elements in FIFO style (First In First Out), exactly opposite of the Stack<T> collection. It contains the elements in the order they were added.
- **SortedList** - are collection classes that can store key-value pairs that are sorted by the keys based on the associated IComparer implementation.

QUESTIONS?
COMMENTS?
CONCERNS?