Garbage Collection in C# | .NET Framework

Last Updated: 01-05-2019

Automatic memory management is made possible by **Garbage Collection in .NET Framework**. When a class object is created at runtime, certain memory space is allocated to it in the heap memory. However, after all the actions related to the object are completed in the program, the memory space allocated to it is a waste as it cannot be used. In this case, garbage collection is very useful as it automatically releases the memory space after it is no longer required.

Garbage collection will always work on **Managed Heap** and internally it has an Engine which is known as the **Optimization Engine**.

Garbage Collection occurs if at least one of multiple conditions is satisfied. These conditions are given as follows:

- If the system has low physical memory, then garbage collection is necessary.
- If the memory allocated to various objects in the heap memory exceeds a pre-set threshold, then garbage collection occurs.
- If the *GC.Collect* method is called, then garbage collection occurs. However, this method is only called under unusual situations as normally garbage collector runs automatically.

Phases in Garbage Collection

There are mainly **3** phases in garbage collection. Details about these are given as follows:

1. **Marking Phase:** A list of all the live objects is created during the marking phase. This is done by following the references from all the root objects. All of the objects that are not on the list of live objects are potentially deleted from the heap memory.
2. **Relocating Phase:** The references of all the objects that were on the list of all the live objects are updated in the relocating phase so that they point to the new location where the objects will be relocated to in the compacting phase.
3. **Compacting Phase:** The heap gets compacted in the compacting phase as the space occupied by the dead objects is released and the live objects remaining are moved. All the live objects that remain after the garbage collection are moved towards the older end of the heap memory in their original order.

Heap Generations in Garbage Collection

The heap memory is organized into 3 generations so that various objects with different lifetimes can be handled appropriately during garbage collection. The memory to each Generation will be given by the **Common Language Runtime(CLR)** depending on the project size. Internally, Optimization Engine will call the *Collection Means Method* to select which objects will go into Gneration 1 or Generation 2.

- **Generation 0 :** All the short-lived objects such as temporary variables are contained in the generation 0 of the heap memory. All the newly allocated objects are also generation 0 objects implicitly unless they are large objects. In general, the frequency of garbage collection is the highest in generation 0.
- **Generation 1 :** If space occupied by some generation 0 objects that are not released in a garbage collection run, then these objects get moved to generation 1. The objects in this generation are a sort of buffer between the short-lived objects in generation 0 and the long-lived objects in generation 2.
- **Generation 2 :** If space occupied by some generation 1 objects that are not released in the next garbage collection run, then these objects get moved to generation 2. The objects in generation 2 are long lived such as static objects as they remain in the heap memory for the whole process duration.

Note: Garbage collection of a generation implies the garbage collection of all its younger generations. This means that all the objects in that particular generation and its younger generations are released. Because of this reason, the garbage collection of generation 2 is called a full garbage collection as all the objects in the heap memory are.released. Also, the memory allocated to the Generation 2 will be greater than Generation 1's memory and similarly the memory of Generation 1 will be greater than Generation 0's memory(**Generation 2** > **Generation 1** > **Generation 0**).
A program that demonstrates the number of heap generations in garbage collection using the GC.MaxGeneration property of the GC class is given as follows:

filter_none
brightness_4

```
using System;

public class Demo {

    // Main Method
    public static void Main(string[] args)
    {
        Console.WriteLine("The number of generations are: " +
                                           GC.MaxGeneration);
    }
}
```
Output:

```
The number of generations are: 2
```

In the above program, the *GC.MaxGeneration* property is used to find the maximum number of generations that are **supported by the system** i.e. 2. If you will run this program on online compilers then you may get different outputs as it depends on the system.

Methods in GC Class

The GC class controls the garbage collector of the system. Some of the methods in the GC class are given as follows:

GC.GetGeneration() Method : This method returns the generation number of the target object. It requires a single parameter i.e. the target object for which the generation number is required.

A program that demonstrates the *GC.GetGeneration()* method is given as follows:

filter_none

brightness_4

```
using System;

public class Demo {

    public static void Main(string[] args)
    {
        Demo obj = new Demo();
        Console.WriteLine("The generation number of object obj is: "
                                          + GC.GetGeneration(obj));
    }
}
```

Output:

```
The generation number of object obj is: 0
```

GC.GetTotalMemory() Method : This method returns the number of bytes that are allocated in the system. It requires a single boolean parameter where true means that the method waits for the occurrence of garbage collection before returning and false means the opposite.
A program that demonstrates the *GC.GetTotalMemory() method* is given as follows:

filter_none
brightness_4

```
using System;

public class Demo {

    public static void Main(string[] args)
    {
        Console.WriteLine("Total Memory:" + GC.GetTotalMemory(false));

        Demo obj = new Demo();

        Console.WriteLine("The generation number of object obj is: "
                                            + GC.GetGeneration(obj));

        Console.WriteLine("Total Memory:" + GC.GetTotalMemory(false));
    }
}
```

Output:

```
Total Memory:4197120

The generation number of object obj is: 0

Total Memory:4204024
```

Note: The output may vary as it depends on the system.
GC.Collect() Method : Garbage collection can be forced in the system using the *GC.Collect() method*. This method requires a single parameter i.e. number of the oldest generation for which garbage collection occurs.
A program that demonstrates the *GC.Collect() Method* is given as follows:

filter_none
brightness_4

```
using System;

public class Demo {

    public static void Main(string[] args)
    {
        GC.Collect(0);
        Console.WriteLine("Garbage Collection in Generation 0 is: "
                                          + GC.CollectionCount(0));
    }
}
```

Output:

```
Garbage Collection in Generation 0 is: 1
```

Benefits of Garbage Collection

- Garbage Collection succeeds in allocating objects efficiently on the heap memory using the generations of garbage collection.
- Manual freeing of memory is not needed as garbage collection automatically releases the memory space after it is no longer required.
- Garbage collection handles memory allocation safely so that no objects use the contents of another object mistakenly.
- The constructors of newly created objects do not have to initialize all the data fields as garbage collection clears the memory of objects that were previously released.

C# - 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.

Create an ArrayList

The `ArrayList` class included in the `System.Collections` namespace. Create an object of the `ArrayList` using the `new` keyword.

Example: Create an ArrayList

```
using System.Collections;

ArrayList arlist = new ArrayList();
// or
var arlist = new ArrayList(); // recommended
```

Adding Elements in ArrayList

Use the `Add()` method or object initializer syntax to add elements in an `ArrayList`.

An `ArrayList` can contain multiple `null` and duplicate values.

Example: Adding Elements in ArrayList

```
// adding elements using ArrayList.Add() method
var arlist1 = new ArrayList();
arlist1.Add(1);
arlist1.Add("Bill");
arlist1.Add(" ");
arlist1.Add(true);
arlist1.Add(4.5);
arlist1.Add(null);

// adding elements using object initializer syntax
var arlist2 = new ArrayList()
                {
                    2, "Steve", " ", true, 4.5, null
                };
```
Try it

Use the `AddRange(ICollection c)` method to add an entire Array, HashTable, SortedList, `ArrayList`, `BitArray`, Queue, and Stack in the `ArrayList`.

Example: Adding Entire Array/ArrayList into ArrayList

```
var arlist1 = new ArrayList();

var arlist2 = new ArrayList()
                    {
                        1, "Bill", " ", true, 4.5, null
                    };

int[] arr = { 100, 200, 300, 400 };

Queue myQ = new Queue();
myQ.Enqueue("Hello");
myQ.Enqueue("World!");

arlist1.AddRange(arlist2); //adding arraylist in arraylist
arlist1.AddRange(arr); //adding array in arraylist
arlist1.AddRange(myQ); //adding Queue in arraylist
```

Try it

Accessing an ArrayList

The `ArrayList` class implements the `IList` interface. So, elements can be accessed using indexer, in the same way as an array. Index starts from zero and increases by one for each subsequent element.

An explicit casting to the appropriate types is required, or use the var variable.

Example: Accessing Elements of ArrayList

```
var arlist = new ArrayList()
                {
                    1,
                    "Bill",
                    300,
                    4.5f
                };

//Access individual item using indexer
int firstElement = (int) arlist[0]; //returns 1
string secondElement = (string) arlist[1]; //returns "Bill"
//int secondElement = (int) arlist[1]; //Error: cannot cover string to int

//using var keyword without explicit casting
```

```
var firstElement = arlist[0]; //returns 1
var secondElement = arlist[1]; //returns "Bill"
//var fifthElement = arlist[5]; //Error: Index out of range

//update elements
arlist[0] = "Steve";
arlist[1] = 100;
//arlist[5] = 500; //Error: Index out of range
```
Try it

ADVERTISEMENT

Iterate an ArrayList

Insert Elements in ArrayList

Use the `Insert()` method to insert an element at the specified index into an `ArrayList`.

Signature: *`void Insert(int index, Object value)`*

Example: Insert Element in ArrayList

```
ArrayList arlist = new ArrayList()
                {
                    1,
                    "Bill",
                    300,
                    4.5f
                };

arlist.Insert(1, "Second Item");
```

```
foreach (var val in arlist)
    Console.WriteLine(val);
```
Try it

Use the `InsertRange()` method to insert a collection in an `ArrayList` at the specfied index.

Signature: *`Void InsertRange(int index, ICollection c)`*

Example: Insert Collection in ArrayList

```
ArrayList arlist1 = new ArrayList()
                {
                    100, 200, 600
                };

ArrayList arlist2 = new ArrayList()
                {
                    300, 400, 500
                };
arlist1.InsertRange(2, arlist2);

foreach(var item in arlist1)
    Console.Write(item + ", "); //output: 100, 200, 300, 400, 500, 600,
```
Try it

Remove Elements from ArrayList

Use the `Remove()`, `RemoveAt()`, or `RemoveRange` methods to remove elements from an `ArrayList`.

Example: Remove Elements from ArrayList

```
ArrayList arList = new ArrayList()
                {
                    1,
                    null,
                    "Bill",
                    300,
                    " ",
                    4.5f,
                    300,
                };

arList.Remove(null); //Removes first occurance of null
arList.RemoveAt(4); //Removes element at index 4
arList.RemoveRange(0, 2);//Removes two elements starting from 1st item (0 index)
```

Try it

Check Element in ArrayList

Use the `Contains()` method to determine whether the specified element exists in the `ArrayList` or not. It returns true if exists otherwise returns false.

Example: Check for Elements

```
ArrayList arList = new ArrayList()
                {
                    1,
                    "Bill",
                    300,
                    4.5f,
                    300
                };
```

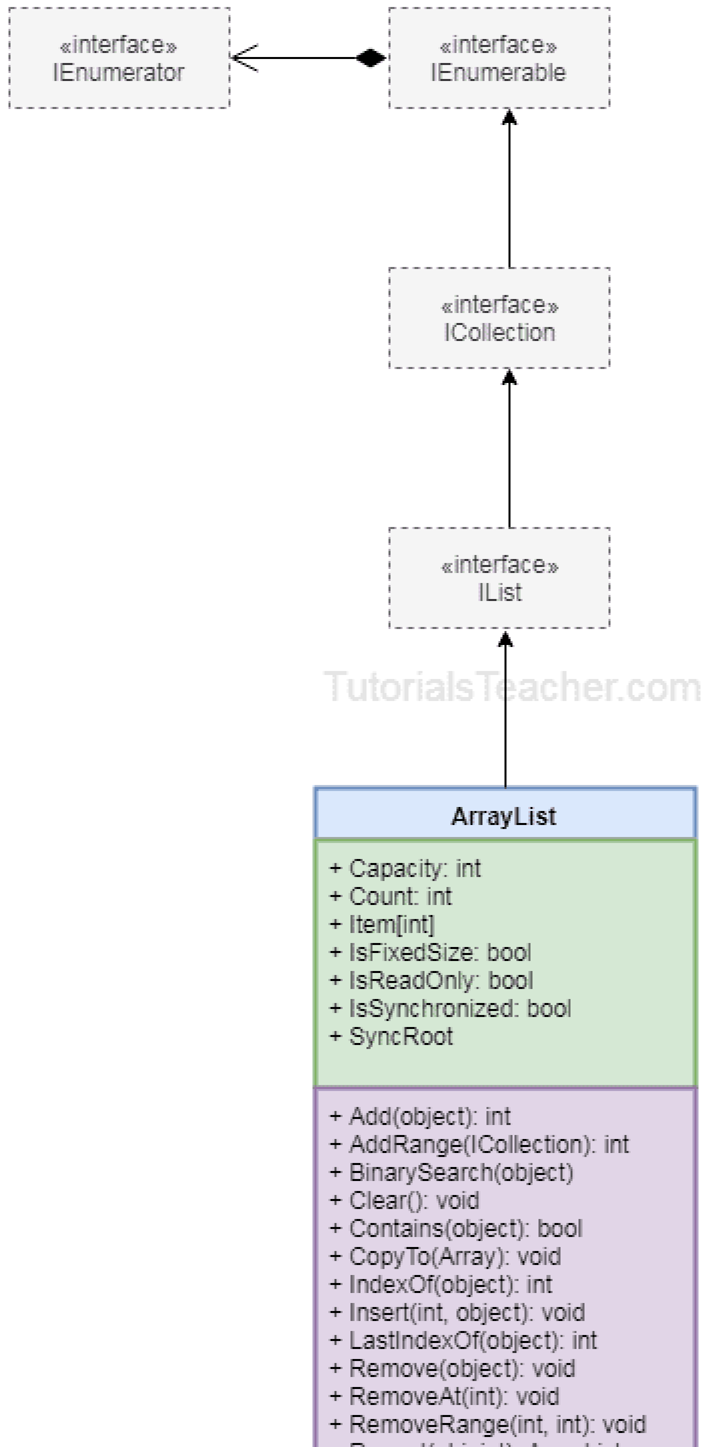
```
Console.WriteLine(arList.Contains(300)); // true
Console.WriteLine(arList.Contains("Bill")); // true
Console.WriteLine(arList.Contains(10)); // false
Console.WriteLine(arList.Contains("Steve")); // false
```
Try it

Note:

It is not recommended to use the `ArrayList` class due to performance issue. Instead, use `List<object>` to store heterogeneous objects. To store data of same data type, use Generic List<T>.

ArrayList Class

The following diagram illustrates the `ArrayList` class.


«interface»
IEnumerator
«interface»
IEnumerable
«interface»
ICollection
«interface»
IList
TutorialsTeacher.com
ArrayList
+ Capacity: int
+ Count: int
+ Item[int]
+ IsFixedSize: bool
+ IsReadOnly: bool
+ IsSynchronized: bool
+ SyncRoot
+ Add(object): int
+ AddRange(ICollection): int
+ BinarySearch(object)
+ Clear(): void
+ Contains(object): bool
+ CopyTo(Array): void
+ IndexOf(object): int
+ Insert(int, object): void
+ LastIndexOf(object): int
+ Remove(object): void
+ RemoveAt(int): void
+ RemoveRange(int, int): void

ArrayList Properties

Properties	Description
Capacity	Gets or sets the number of elements that the ArrayList can contain.
Count	Gets the number of elements actually contained in the ArrayList.
IsFixedSize	Gets a value indicating whether the ArrayList has a fixed size.
IsReadOnly	Gets a value indicating whether the ArrayList is read-only.
Item	Gets or sets the element at the specified index.

ArrayList Methods

Methods	Description
Add()/AddRange()	Add() method adds single elements at the end of ArrayList. AddRange() method adds all the elements from the specified collection into ArrayList.
Insert()/InsertRange()	Insert() method insert a single elements at the specified index in ArrayList. InsertRange() method insert all the elements of the specified collection starting from specified index in ArrayList.
Remove()/RemoveRange()	Remove() method removes the specified element from the ArrayList. RemoveRange() method removes a range of elements from the ArrayList.
RemoveAt()	Removes the element at the specified index from the ArrayList.
Sort()	Sorts entire elements of the ArrayList.

Methods	Description
Reverse()	Reverses the order of the elements in the entire ArrayList.
Contains	Checks whether specified element exists in the ArrayList or not. Returns true if exists otherwise false.
Clear	Removes all the elements in ArrayList.
CopyTo	Copies all the elements or range of elements to compitible Array.
GetRange	Returns specified number of elements from specified index from ArrayList.
IndexOf	Search specified element and returns zero based index if found. Returns -1 if element not found.
ToArray	Returns compitible array from an ArrayList.

The following table lists the differences between Array and ArrayList in C#.

Array	ArrayList
Must include **System** namespace to use array.	Must include **System.Collections** namespace to use ArraList.

Array Declaration & Initialization:
`int[] arr = new int[5]`
`int[] arr = new int[5]{1, 2, 3, 4, 5};`
`int[] arr = {1, 2, 3, 4, 5};`
	ArrayList Declaration & Initialization:
	`ArrayList arList = new ArrayList();`
	`arList.Add(1);`
	`arList.Add("Two");`
	`arList.Add(false);`

Array stores a fixed number of elements. The size of an Array must be specified at the time of initialization.	ArrayList grows automatically and you don't need to specify the size.
Array is strongly typed. This means that an array can store only	ArrayList can store any type of items\elements.

Array	ArrayList
specific type of items\elements.	
No need to cast elements of an array while retrieving because it is strongly typed and stores a specific type of items only.	The items of ArrayList need to be cast to an appropriate data type while retrieving. So, boxing and unboxing happens.
Performs faster than ArrayList because it is strongly typed.	Performs slows because of boxging and unboxing.
Use static helper class Array to perform different tasks on the array.	ArrayList itself includes various utility methods for various tasks.

Visit Array or ArrayList in the C# tutorials section for more information.

C# Strings

❮ PreviousNext ❯

C# Strings

Strings are used for storing text.

A `string` variable contains a collection of characters surrounded by double quotes:

Example

Create a variable of type `string` and assign it a value:

```
string greeting = "Hello";
```

Run example »

String Length

A string in C# is actually an object, which contain properties and methods that can perform certain operations on strings. For example, the length of a string can be found with the `Length` property:

Example

```
string txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

Console.WriteLine("The length of the txt string is: " + txt.Length);
```

Run example »

Other Methods

There are many string methods available, for example `ToUpper()` and `ToLower()`, which returns a copy of the string converted to uppercase or lowercase:

Example

```
string txt = "Hello World";

Console.WriteLine(txt.ToUpper());   // Outputs "HELLO WORLD"

Console.WriteLine(txt.ToLower());   // Outputs "hello world"
```

Run example »

String Concatenation

The + operator can be used between strings to combine them. This is called **concatenation**:

Example

```
string firstName = "John ";
string lastName = "Doe";
string name = firstName + lastName;
Console.WriteLine(name);
```

Run example »

Note that we have added a space after "John" to create a space between firstName and lastName on print.

You can also use the `string.Concat()` method to concatenate two strings:

Example

```
string firstName = "John ";
string lastName = "Doe";
string name = string.Concat(firstName, lastName);

Console.WriteLine(name);
```

Run example »

ADVERTISEMENT

String Interpolation

Another option of string concatenation, is **string interpolation**, which substitutes values of variables into placeholders in a string. Note that you do not have to worry about spaces, like with concatenation:

Example

```
string firstName = "John";

string lastName = "Doe";

string name = $"My full name is: {firstName} {lastName}";

Console.WriteLine(name);
```

Run example »

Also note that you have to use the dollar sign ($) when using the string interpolation method.

String interpolation was introduced in C# version 6.

Access Strings

You can access the characters in a string by referring to its index number inside square brackets [].

This example prints the **first character** in **myString**:

Example

```
string myString = "Hello";

Console.WriteLine(myString[0]);  // Outputs "H"
```

Run example »

Note: String indexes start with 0: [0] is the first character. [1] is the second character, etc.

This example prints the **second character** (1) in **myString**:

Example

```
string myString = "Hello";

Console.WriteLine(myString[1]);  // Outputs "e"
```

Run example »

You can also find the index position of a specific character in a string, by using the `IndexOf()` method:

Example

```
string myString = "Hello";

Console.WriteLine(myString.IndexOf("e"));  // Outputs "1"
```

Run example »

Another useful method is `Substring()`, which extracts the characters from a string, starting from the specified character position/index, and returns a new string. This method is often used together with `IndexOf()` to get the specific character position:

Example

```
// Full name

string name = "John Doe";


// Location of the letter D

int charPos = name.IndexOf("D");


// Get last name

string lastName = name.Substring(charPos);
```

```
// Print the result

Console.WriteLine(lastName);
```

Run example »

Special Characters

Because strings must be written within quotes, C# will misunderstand this string, and generate an error:

```
string txt = "We are the so-called "Vikings" from the north.";
```

The solution to avoid this problem, is to use the **backslash escape character**.

The backslash (\) escape character turns special characters into string characters:

Escape character	Result	Description
\'	'	Single quote
\"	"	Double quote

\\	\	Backslash

The sequence \" inserts a double quote in a string:

Example

```
string txt = "We are the so-called \"Vikings\" from the north.";
```

Try it Yourself »

The sequence \' inserts a single quote in a string:

Example

```
string txt = "It\'s alright.";
```

Try it Yourself »

The sequence \\ inserts a single backslash in a string:

Example

```
string txt = "The character \\ is called backslash.";
```

Try it Yourself »

Other useful escape characters in C# are:

Code	Result	Try it
\n	New Line	Try it »
\t	Tab	Try it »
\b	Backspace	Try it »

Adding Numbers and Strings

WARNING!

C# uses the + operator for both addition and concatenation.

Remember: Numbers are added. Strings are concatenated.

If you add two numbers, the result will be a number:

Example

```
int x = 10;

int y = 20;

int z = x + y;  // z will be 30 (an integer/number)
```

Run example »

If you add two strings, the result will be a string concatenation:

Example

```
string x = "10";

string y = "20";

string z = x + y;  // z will be 1020 (a string)
```

Run example »

C# Inheritance

❮ PreviousNext ❯

Inheritance (Derived and Base Class)

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.

In the example below, the `Car` class (child) inherits the fields and methods from the `Vehicle` class (parent):

Example

```
class Vehicle  // base class (parent)

{

  public string brand = "Ford";  // Vehicle field

  public void honk()             // Vehicle method

  {

    Console.WriteLine("Tuut, tuut!");

  }

}
```

```
class Car : Vehicle  // derived class (child)

{

  public string modelName = "Mustang";  // Car field

}


class Program

{

  static void Main(string[] args)

  {

    // Create a myCar object

    Car myCar = new Car();


    // Call the honk() method (From the Vehicle class) on the myCar object

    myCar.honk();


    // Display the value of the brand field (from the Vehicle class) and the value of the modelName
from the Car class

    Console.WriteLine(myCar.brand + " " + myCar.modelName);

  }
}
```

Run example »

Why And When To Use "Inheritance"?

- It is useful for code reusability: reuse fields and methods of an existing class when you create a new class.

Tip: Also take a look at the next chapter, Polymorphism, which uses inherited methods to perform different tasks.

The sealed Keyword

If you don't want other classes to inherit from a class, use the `sealed` keyword:

If you try to access a `sealed` class, C# will generate an error:

```
sealed class Vehicle

{

  ...

}



class Car : Vehicle
```

```
{

  ...

}
```

The error message will be something like this:

```
'Car': cannot derive from sealed type 'Vehicle'
```

❮ PreviousNext ❯

Polymorphism means “Many Forms”. In Polymorphism, poly means “Many” and morph means “Forms.” Polymorphism is one of the main pillars in Object Oriented Programming. It allows you to create multiple methods with the same name but different signatures in the same class. The same name methods can also be in derived classes.

There are two types of Polymorphism,

1. Method Overloading
2. Method Overriding

In this article, I will explain method overloading and method overriding concept in C#. I will try to demonstrate step by step differences between these.

Method Overloading

Method Overloading is a type of polymorphism. It has several names like “Compile Time Polymorphism” or “Static Polymorphism” and sometimes it is called “Early Binding”.

Method Overloading means creating multiple methods in a class with same names but different signatures (Parameters). It permits a class, struct, or interface to declare multiple methods with the same name with unique signatures.

Compiler automatically calls required method to check number of parameters and their type which are passed into that method.

```
using System;
namespace DemoCsharp
{
    class Program
    {
```

```
 public int Add(int num1, int num2)
 {
   return (num1 + num2);
 }
    public int Add(int num1, int num2, int num3)
    {
      return (num1 + num2 + num3);
    }
    public float Add(float num1, float num2)
    {
      return (num1 + num2);
    }
    public string Add(string value1, string value2)
    {
      return (value1 + " " + value2);
    }
    static void Main(string[] args)
    {
      Program objProgram = new Program();
      Console.WriteLine("Add with two int parameter :" + objProgram.Add(3, 2));
      Console.WriteLine("Add with three int parameter :" + objProgram.Add(3, 2, 8));
      Console.WriteLine("Add with two float parameter :" + objProgram.Add(3 f, 22 f));
      Console.WriteLine("Add with two string parameter :" + objProgram.Add("hello", "world"));
      Console.ReadLine();
    }
  }
}
```

In the above example, you can see that there are four methods with same name but type of parameters or number of parameters is different. When you call Add(4,5), complier automatically calls the method which has two integer parameters and when you call Add("hello","world"), complier calls the method which has two string parameters. So basically in method overloading complier checks which method should be called at the time of compilation.

Note: Changing the return type of method does not make the method overloaded. You cannot create method overloaded vary only by return type.

Method Overriding

Method Overriding is a type of polymorphism. It has several names like “Run Time Polymorphism” or “Dynamic Polymorphism” and sometime it is called “Late Binding”.

Method Overriding means having two methods with same name and same signatures [parameters], one should be in the base class and other method should be in a derived class [child class]. You can override the functionality of a base class method to create a same name method with same signature in a derived class. You can achieve method overriding using inheritance. Virtual and Override keywords are used to achieve method overriding.

```
using System;
namespace DemoCsharp
{
    class BaseClass
    {
        public virtual int Add(int num1, int num2)
        {
            return (num1 + num2);
        }
      }
      class ChildClass: BaseClass
      {
          public override int Add(int num1, int num2)
          {
              if (num1 <= 0 || num2 <= 0)
              {
                  Console.WriteLine("Values could not be less than zero or equals to zero");
                  Console.WriteLine("Enter First value : ");
                  num1 = Convert.ToInt32(Console.ReadLine());
                  Console.WriteLine("Enter First value : ");
                  num2 = Convert.ToInt32(Console.ReadLine());
              }
              return (num1 + num2);
          }
      }
      class Program
      {
          static void Main(string[] args)
          {
              BaseClass baseClassObj;
              baseClassObj = new BaseClass();
              Console.WriteLine("Base class method Add :" + baseClassObj.Add(-3, 8));
              baseClassObj = new ChildClass();
              Console.WriteLine("Child class method Add :" + baseClassObj.Add(-2, 2));
              Console.ReadLine();
          }
      }
  }
```

In the above example, I have created two same name methods in the BaseClass as well as in the ChildClass. When you call the BaseClass Add method with less than zero value as parameters then it adds successfully. But when you call the ChildClass Add method with less than zero value then it checks for negative value. And the passing values are negative then it asks for new value.

So, here it is clear that we can modify the base class methods in derived classes.

Points to be remembered,

1. Method cannot be private.
2. Only abstract or virtual method can be overridden.
3. Which method should be called is decided at run time.

Conclusion

So, today we learned what Polymorphism is in OOP and what are the differences between method overloading and method overriding.

Prerequisite : Data Types in C#
Boxing and unboxing is an important concept in C#. C# Type System contains three data types: **Value Types (int, char, etc)**, **Reference Types (object)** and **Pointer Types**. Basically it convert a Value Type to a Reference Type, and vice versa. Boxing and Unboxing enables 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 **(char, int etc.) to a** Reference Type**(object)** is called **Boxing**.
- Boxing is implicit conversion process in which object type (super type) is used.
- The Value type is always stored in Stack. The Referenced Type is stored in Heap.
- **Example :**
- `int num = 23; // 23 will assigned to num`
- `Object Obj = num; // Boxing`
- **Description :** First declare a value type variable (num), which is integer type and assigned it with value 23. Now create a references object type (obj) and applied Explicit operation which results in num value type to be copied and stored in object reference type obj as shown in below figure :

```
// C# implementation to demonstrate

// the Boxing

using System;

class GFG {



    // Main Method
    static public void Main()
    {

        // assigned int value
        // 2020 to num
        int num = 2020;

        // boxing
        object obj = num;

        // value of num to be change
        num = 100;

        System.Console.WriteLine
        ("Value - type value of num is : {0}", num);
        System.Console.WriteLine
        ("Object - type value of obj is : {0}", obj);
    }
}
```

Output:

```
Value - type value of num is : 100
Object - type value of obj is : 2020
```

Unboxing In C#

- The process of converting **reference type into the value type** is known as **Unboxing**.
- It is explicit conversion process.
- **Example :**

```
int num = 23;         // value type is int and assigned value 23
Object Obj = num;    // Boxing
int i = (int)Obj;    // Unboxing
```

- **Description :** Declaration a value type variable (num), which is integer type and assigned with integer value 23. Now, create a reference object type (obj).The explicit operation for boxing create an value type integer i and applied casting method. Then the referenced type residing on Heap is copy to stack as shown in below figure :

- Let's understand **Unboxing** with a C# programming code :

filter_none

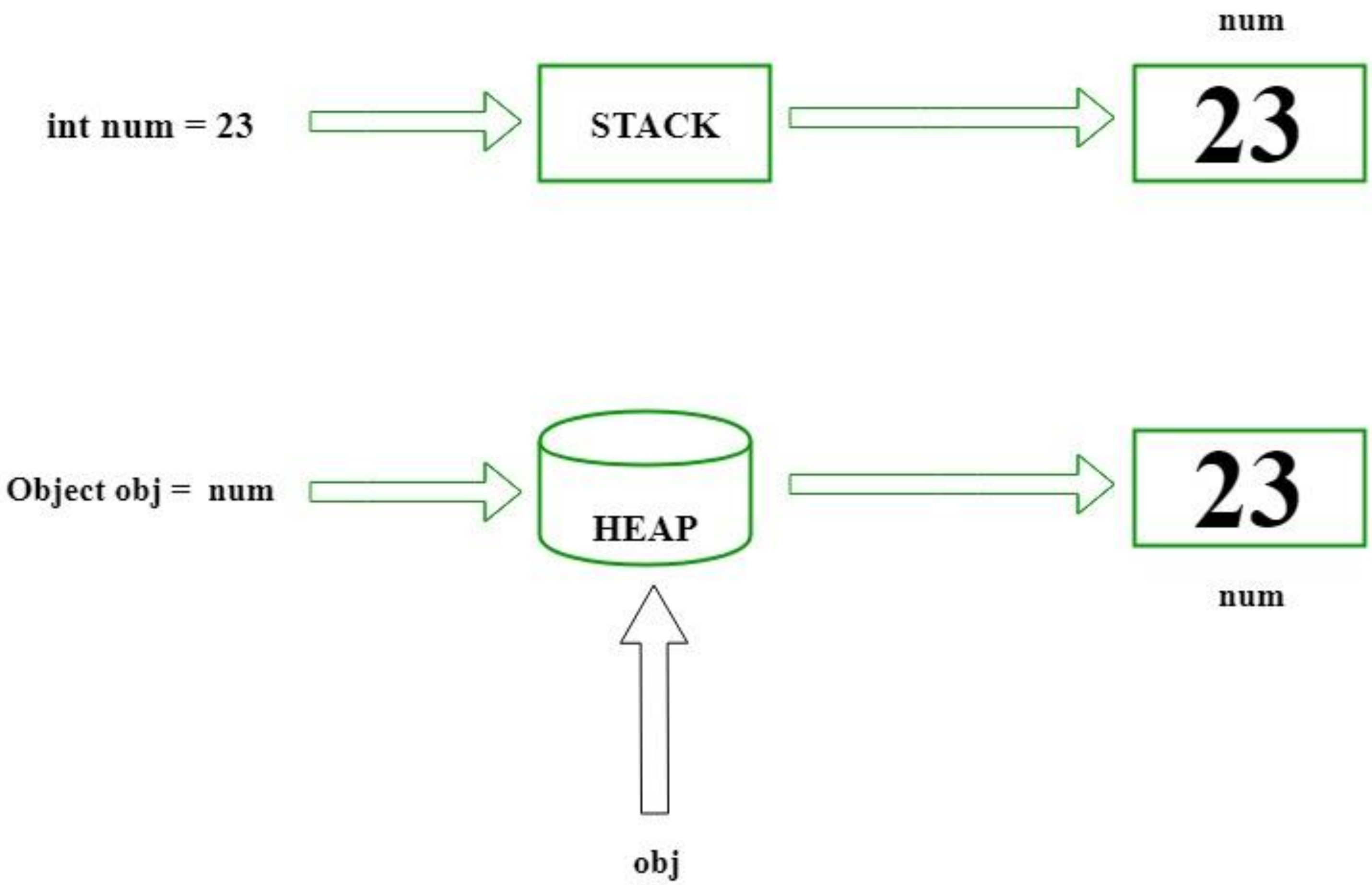
edit
play_arrow
brightness_4

```
// C# implementation to demonstrate
// the Unboxing
using System;
```

```
class GFG {

    // Main Method
    static public void Main()
    {

        // assigned int value
        // 23 to num
        int num = 23;

        // boxing
        object obj = num;

        // unboxing
        int i = (int)obj;

        // Display result
        Console.WriteLine("Value of ob object is : " + obj);
        Console.WriteLine("Value of i is : " + i);
        }
}
```

Output:

```
Value of ob object is : 23

Value of i is : 23
```


num
int num = 23
STACK
23
Object obj = num
HEAP
23
num
obj

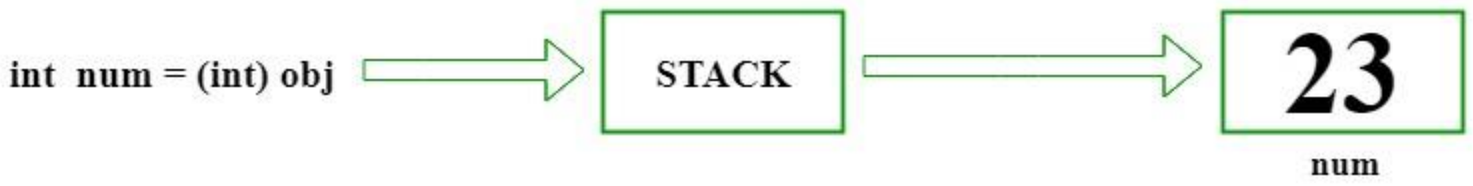


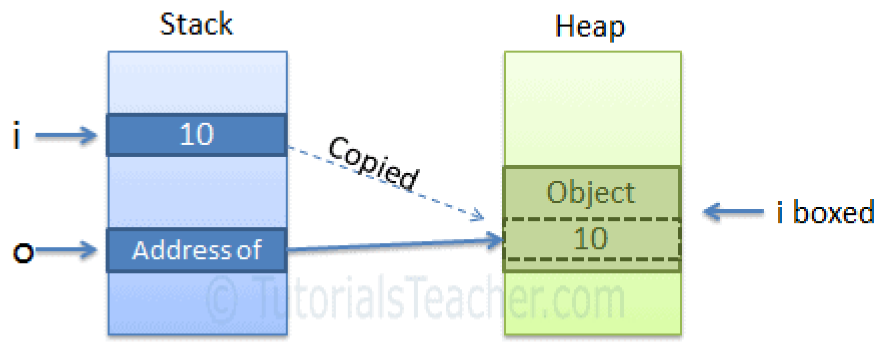
Figure: unboxing

Why named boxing?

You may be wondering, why is it named as boxing?

As you know, all the reference types stored on heap where it contains the address of the value and value type is just an actual value stored on the stack. Now, as shown in the first example, int `i` is assigned to object `o`. Object `o` must be an address and not a value itself. So, the CLR boxes the value type by creating a new System.Object on the heap and wraps the value of `i` in it and then assigns an address of that object to `o`. So, because the CLR creates a box on the heap that stores the value, the whole process is called 'Boxing'.

The following figure illustrates the boxing process.



Boxing

What is Unboxing?

Unboxing is the reverse of boxing. It is the process of converting a reference type to value type. Unboxing extract the value from the reference type and assign it to a value type.

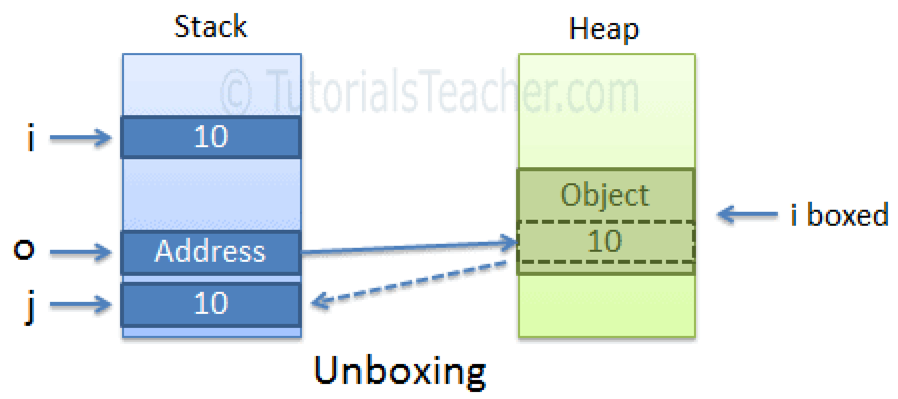
Unboxing is explicit. It means we have to cast explicitly.

Example: Unboxing

Copy

```
object o = 10;
int i = (int)o; //performs unboxing
```

The following figure illustrates the unboxing process.



A boxing conversion makes a copy of the value. So, changing the value of one variable will not impact others.

```
int i = 10;
object o = i; // boxing
o = 20;
Console.WriteLine(i); // output: 10
```

The casting of a boxed value is not permitted. The following will throw an exception.

Example: Invalid Conversion

Copy

```
int i = 10;
object o = i; // boxing
double d = (double)o; // runtime exception
```

First do unboxing and then do casting, as shown below.

Example: Valid Conversion

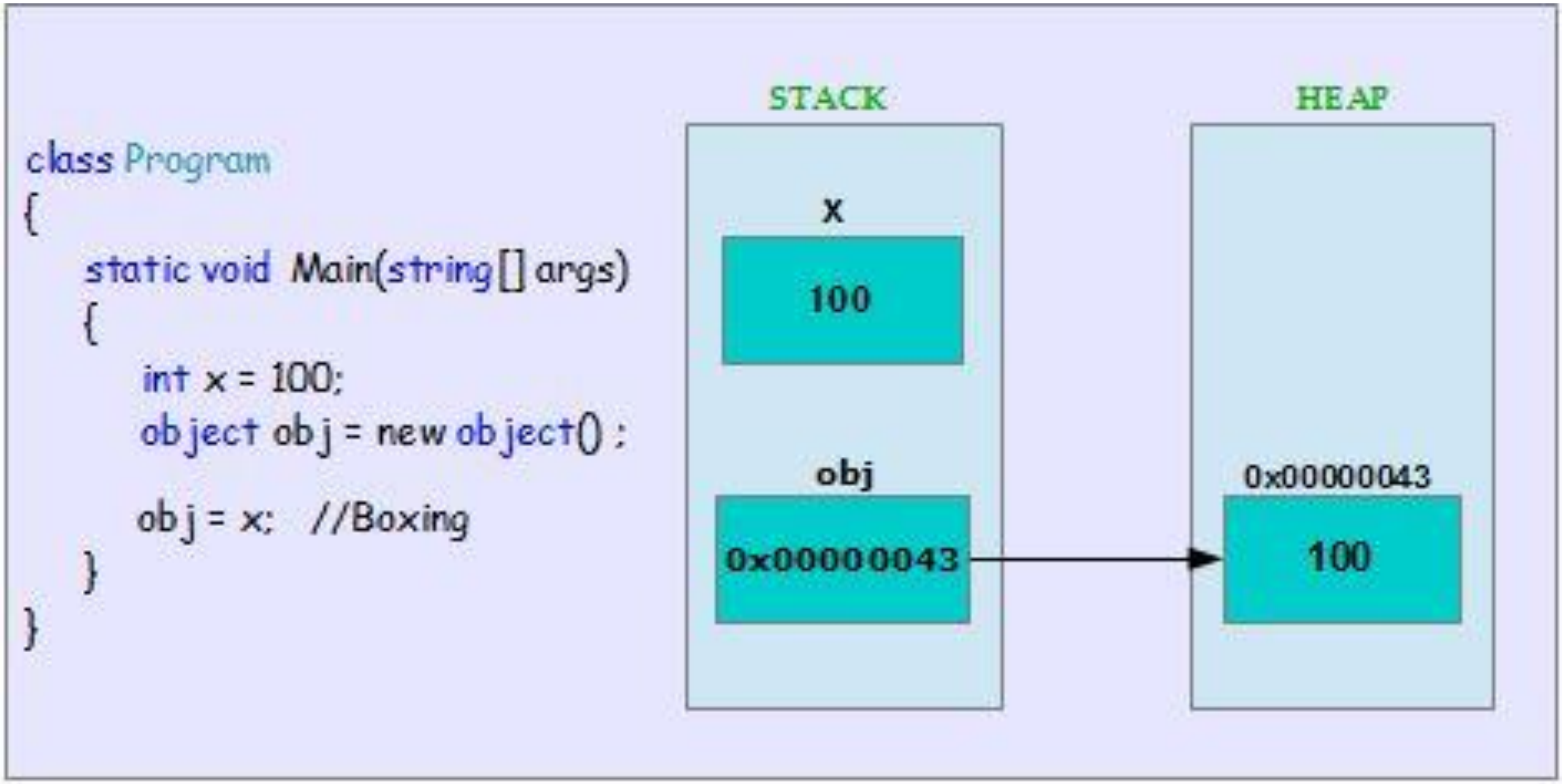
Copy

```
int i = 10;
object o = i; // boxing
double d = (double)(int)o; // valid
```

Note:

Boxing and unboxing degrade the performance. So, avoid using it. Use generics to avoid boxing and unboxing. For example, use List instead of ArrayList.

Read this interesting discussion on stackoverflow: Why do we need boxing and unboxing in C#?



```
class Program
{
    static void Main(string[] args)
    {
        int x = 100;
        object obj = new object() ;

        obj = x;   //Boxing
    }
}
```

Metadata and Self-Describing Components

- 03/30/2017
- 8 minutes to read
  -

    -
    -
    -
    -
    -
    - +5

In the past, a software component (.exe or .dll) that was written in one language could not easily use a software component that was written in another language. COM provided a step towards solving this problem. .NET makes component interoperation even easier by allowing compilers to emit additional declarative information into all modules and assemblies. This information, called metadata, helps components to interact seamlessly.

Metadata is binary information describing your program that is stored either in a common language runtime portable executable (PE) file or in memory. When you compile your code into a PE file, metadata is inserted into one portion of the file, and your code is converted to Microsoft intermediate language (MSIL) and inserted into another portion of the file. Every type and member that is defined and referenced in a module or assembly is described within metadata. When code is executed, the runtime loads metadata into memory and references it to discover information about your code's classes, members, inheritance, and so on.

Metadata describes every type and member defined in your code in a language-neutral manner. Metadata stores the following information:

- Description of the assembly.
  - Identity (name, version, culture, public key).

    - The types that are exported.
    - Other assemblies that this assembly depends on.
    - Security permissions needed to run.
- Description of types.
    - Name, visibility, base class, and interfaces implemented.
    - Members (methods, fields, properties, events, nested types).
- Attributes.
    - Additional descriptive elements that modify types and members.

Benefits of Metadata

Metadata is the key to a simpler programming model, and eliminates the need for Interface Definition Language (IDL) files, header files, or any external method of component reference. Metadata enables .NET languages to describe themselves automatically in a language-neutral manner, unseen by both the developer and the user. Additionally, metadata is extensible through the use of attributes. Metadata provides the following major benefits:

- Self-describing files.

  Common language runtime modules and assemblies are self-describing. A module's metadata contains everything needed to interact with another module. Metadata automatically provides the functionality of IDL in COM, so you can use one file for both definition and implementation. Runtime modules and assemblies do not even require registration with the operating system. As a result, the descriptions used by the runtime always reflect the actual code in your compiled file, which increases application reliability.

- Language interoperability and easier component-based design.

  Metadata provides all the information required about compiled code for you to inherit a class from a PE file written in a different language. You can create an instance of any class written in any managed language (any language that targets the common language runtime) without worrying about explicit marshaling or using custom interoperability code.

- Attributes.

  .NET lets you declare specific kinds of metadata, called attributes, in your compiled file. Attributes can be found throughout .NET and are used to control in more detail how your program behaves at run time. Additionally, you can emit your own custom metadata into .NET files through user-defined custom attributes. For more information, see Attributes.

Metadata and the PE File Structure

Metadata is stored in one section of a .NET portable executable (PE) file, while Microsoft intermediate language (MSIL) is stored in another section of the PE file. The metadata portion of the file contains a series of table and heap data structures. The MSIL portion contains MSIL and metadata tokens that reference the metadata portion of the PE file. You might encounter metadata tokens when you use tools such as the MSIL Disassembler (Ildasm.exe) to view your code's MSIL, for example.

Metadata Tables and Heaps

Each metadata table holds information about the elements of your program. For example, one metadata table describes the classes in your code, another table describes the fields, and so on. If you have ten classes in your code, the class table will have tens rows, one for each class. Metadata tables reference other tables and heaps. For example, the metadata table for classes references the table for methods.

Metadata also stores information in four heap structures: string, blob, user string, and GUID. All the strings used to name types and members are stored in the string heap. For example, a method table does not directly store the name of a particular method, but points to the method's name stored in the string heap.

Metadata Tokens

Each row of each metadata table is uniquely identified in the MSIL portion of the PE file by a metadata token. Metadata tokens are conceptually similar to pointers, persisted in MSIL, that reference a particular metadata table.

A metadata token is a four-byte number. The top byte denotes the metadata table to which a particular token refers (method, type, and so on). The remaining three bytes specify the row in the metadata table that corresponds to the programming element being described. If you define a method in C# and compile it into a PE file, the following metadata token might exist in the MSIL portion of the PE file:

```
0x06000004
```

The top byte (`0x06`) indicates that this is a **MethodDef** token. The lower three bytes (`000004`) tells the common language runtime to look in the fourth row of the **MethodDef** table for the information that describes this method definition.

Metadata within a PE File

When a program is compiled for the common language runtime, it is converted to a PE file that consists of three parts. The following table describes the contents of each part.

METADATA WITHIN A PE FILE

PE section	**Contents of PE section**
PE header	The index of the PE file's main sections and the address of the entry point. The runtime uses this information to identify the file as a PE file and to determine where execution starts when loading the program into memory.
MSIL	The Microsoft intermediate language instructions (MSIL) that make up your code. Many MSIL

PE section	Contents of PE section
instructions	instructions are accompanied by metadata tokens.
Metadata	Metadata tables and heaps. The runtime uses this section to record information about every type and member in your code. This section also includes custom attributes and security information.

Run-Time Use of Metadata

To better understand metadata and its role in the common language runtime, it might be helpful to construct a simple program and illustrate how metadata affects its run-time life. The following code example shows two methods inside a class called `MyApp`. The `Main` method is the program entry point, while the `Add` method simply returns the sum of two integer arguments.

C#Copy

```
using System;
public class MyApp
{
   public static int Main()
   {
      int ValueOne = 10;
      int ValueTwo = 20;
      Console.WriteLine("The Value is: {0}", Add(ValueOne, ValueTwo));
      return 0;
   }
   public static int Add(int One, int Two)
   {
      return (One + Two);
```

```
    }
}
```

When the code runs, the runtime loads the module into memory and consults the metadata for this class. Once loaded, the runtime performs extensive analysis of the method's Microsoft intermediate language (MSIL) stream to convert it to fast native machine instructions. The runtime uses a just-in-time (JIT) compiler to convert the MSIL instructions to native machine code one method at a time as needed.

Uses of Metadata

Given below are the uses of Metadata:

- It provides description about assembly data types like name, visibility, base class and interfaces etc.
- It provides data members like methods, fields, properties, events and nested types.
- It also provides additional description of the elements that modify types and members.
- It have identity like name, version, public key etc.
- It is a key to simple programming model and it will eliminate the necessity for IDL (Interface Definition Language) files, header files.

Given below are the examples of Metadata in C#:

Example #1

Multiplication of 3 Numbers

Code: Multiplication.cs

```
using System; //Used for declaring the package or used for importing existed

packege

public class Multiplication//declaring the class

{

public static int Main ()// main method for displaying the output

{

//declaring and defining the varaiables

int x = 50;

int y = 20;

int z=30;

//Printing the output of the multiplication of 2 numbers
Console.WriteLine ("Multiplication of {0},{1} and {2} is

{3}",x,y,z,multiplication(x,y,z));

return 0;

}

public static int multiplication(int x, int y, int z)// multiplication() method

implemention

{

return (x * y*z);// return multiplication of 3 numbers

}

}
```

Output:

Explanation:

- As you can see in the about you can see the actual data, if we want metadata or binary data we can see the compiler inside machine generated code, that is always encrypted humans can't understand it.

Example #2

Area of Square

Code: SquareOfArea.cs

```
using System; //Used for declaring the package or used for importing existed
packege
public class SquareArea//declaring the class
{
public static int Main ()// main method for displaying the output
{
//declaring and defining the varaiables
int x = 50;
//Printing the output of the areaOfSquare

Console.WriteLine ("Area of Square is {0}",areaOfSquare(x));

return 0;

}

public static int areaOfSquare(int x)// multiplication() method implemention

{

return (x*x);// return area Of Square

}

}
```

Output:

Explanation:

- As you can see in the about you can see the actual data, if we want metadata or binary data we can see the compiler inside machine generated code, that is always encrypted humans can’t understand it.

The metadata contains the following information:

- Description of assembly
    - Identity (name, version, culture, public key).
    - Types that are exported.
    - Other assemblies where this assembly depends on.
    - The security permissions required to run.
- Description of types
    - Name, visibility, base class and interfaces implemented
    - Members (methods, fields, properties, events and nested types)
- Attributes
    - The additional description of the elements that modify types and members.