Boxing and Unboxing in C#

Boxing and unboxing in C# allows developers to convert .NET data types from value type to reference type and vice versa. Converting a value type to a reference type is called called boxing in C# and converting a reference type to a value type is called unboxing in C#.

C# provides a "unified type system". All types including value types derive from the type object. It is possible to call the object methods on any value, even values of "primitive" types, such as int. The example is shown below.

1. **using** System;
2. **class** Test
3. {
4. **static** **void** Main()
5. {
6. Console.WriteLine(3.ToString());
7. }
8. }

It calls the object-defined ToString method on an integer literal. The example -

1. **class** Test
2. {
3. **static** **void** Main()
4. {
5. **int** i = 1;
6. **object** o = i; // boxing
7. **int** j = (**int**)o; // unboxing
8. }
9. }

An int value can be converted into object and back again into int.

This example shows both, boxing and unboxing. When a variable of a value type needs to be converted into a reference type, an object box is allocated to hold the value, and the value is copied into the box.

Unboxing is just the opposite. When an object box is cast back to its original value type, the value is copied out of the box and into the appropriate storage location.

Boxing conversions

A boxing conversion permits any value-type to be implicitly converted to the type object or to any interface-type implemented by the value-type. Boxing a value of a value-type consists of allocating an object instance and copying the value-type value into that instance.

For example, for any value-type G, the boxing class would be declared as follows:

1. **class** vBox
2. {
3. Gvalue;
4. G\_Box(G g)
5. {
6. value = g;
7. }
8. }

Boxing of a value v of type G now consists of executing the expression new G\_Box(v), and returning the resulting instance as a value of type object. Thus, the statements

1. **int** i = 12;
2. **object** box = i;

conceptually correspond to,

1. **int** i = 12;
2. **object** box = **new** int\_Box(i);

Boxing classes like G\_Box and int\_Box above don't actually exist and the dynamic type of a boxed value isn't actually a class type. Instead, a boxed value of type G has the dynamic type G, and a dynamic type check using the is operator can simply reference type G.

For example -

1. **int** i = 12;
2. **object** box = i;
3. **if** (box **is** **int**)
4. {
5. Console.Write("Box contains an int");
6. }

The above code will output the string "Box contains an int" on the console.

A boxing conversion implies making a copy of the value being boxed. This is different from a conversion of a reference-type to type object, in which the value continues to reference the same instance and simply is regarded as the less derived type object.

For example, given the declaration -

1. **struct** Point
2. {
3. **public** **int** x, y;
4. **public** Point(**int** x, **int** y)
5. {
6. **this**.x = x;
7. **this**.y = y;
8. }
9. }

the following statements -

1. Point p =**new** Point(10, 10);
2. **object** box = p;
3. p.x = 20;
4. Console.Write(((Point)box).x);

These will output the value 10 on the console because the implicit boxing operation that occurs in the assignment of p to box causes the value of p to be copied. Had Point instead been declared a class, the value 20 would be output because p and box would reference the same instance.

Unboxing conversions

An unboxing conversion permits an explicit conversion from type object to any value-type or from any interface-type to any value-type that implements the interface-type. An unboxing operation consists of first checking that the object instance is a boxed value of the given value-type, and then copying the value out of the instance. Unboxing conversion of an object box to a value-type G consists of executing the expression ((G\_Box)box).value.

Thus, the statements,

1. **object** box = 12;
2. **int** i = (**int**)box;

conceptually correspond to,

1. **object** box = **new** int\_Box(12);
2. **int** i = ((int\_Box)box).value;

For an unboxing conversion to a given value-type to succeed at run-time, the value of the source argument must be a reference to an object that was previously created by boxing a value of that value-type. If the source argument is null or a reference to an incompatible object, an InvalidCastException is thrown.

Conclusion

This type-system unification provides value types with the benefits of object-ness without introducing unnecessary overhead. For programs that don't need int values to act like objects, int values are simply 32-bit values. For programs that need int values to behave like objects, this capability is available on demand. This ability to treat value types as objects bridges the gap between value types and reference types that exists in most languages.

Out And Ref

In this article I have tried to show the basic difference between the out and ref keywords.

In C# there are basic two types of data, one is value types and the other one is reference types. When we pass this data to any method via parameters both act differently. By default value types are passed by value and reference types pass their references.

Pass by value means, if we make changes on data in the calling method, no changes will reflect in the caller method.

**Example**

1. **static** **void** Main(**string**[] args)
2. {
3. **int** Counter = 10;
4. //display value before parsing
5. Console.WriteLine($"Original Value: {Counter}");
6. doSomething(Counter);
7. //display value after parsing
8. Console.WriteLine($"Value After Changing: {Counter}");
9. Console.ReadLine();
10. }
11. **static** **void** doSomething(**int** number)
12. {
13. number = 80;
14. }

**OUTPUT**

Text

Description automatically generated

We can see here changes made in the ‘doSomething’ method are not reflected in the original data. That is because it is passed by value. But what if we want to pass this value type as reference?

Out and ref keywords are used to pass value types as reference. So if we make changes on data in the calling method, all changes will reflect in the caller method also (original data).

To pass value types as reference we simply put (ref/out) keyword before the parameter in the definition of method, and while calling we have to put (ref/out) keyword before the variable name accordingly.

**Example**

1. **static** **void** Main(**string**[] args)
2. {
3. **int** Counter = 10;
4. //display value before parsing
5. Console.WriteLine($"Original Value: {Counter}");
6. doSomething(**ref** Counter);
7. //display value after parsing
8. Console.WriteLine($"Value After Changing: {Counter}");
9. Console.ReadLine();
10. }
11. **static** **void** doSomething(**ref** **int** number)
12. {
13. number = 80;
14. }

**OUTPUT**

A screenshot of a computer

Description automatically generated with medium confidence

Here changes made in the calling method are reflected since reference of the variable is passed to the method. We can have similar output by using out keyword also. The only difference between these two is, when we use ref keyword the original value of the variable is also passed to calling method, whereas in case of out keyword we have to assign new value to the variable in the calling method.

Let's understand this with an example,

Graphical user interface, text, application, email

Description automatically generated

We can clearly see that we have to initialize the variable if we use out keyword. And it is ok with ref keyword.

In this article we have learned the difference between out and ref, if you have any doubts feel free to comment below.

# C# Function

Function is a block of code that has a signature. Function is used to execute statements specified in the code block. A function consists of the following components:

**Function name:** It is a unique name that is used to make Function call.

**Return type:** It is used to specify the data type of function return value.

**Body:** It is a block that contains executable statements.

7.3M

157

HTML Tutorial

**Access specifier:** It is used to specify function accessibility in the application.

**Parameters:** It is a list of arguments that we can pass to the function during call.

### **C# Function Syntax**

1. <access-specifier><**return**-type>FunctionName(<parameters>)
2. {
3. // function body
4. // return statement
5. }

Access-specifier, parameters and return statement are optional.

Let's see an example in which we have created a function that returns a string value and takes a string parameter.

### **C# Function: using no parameter and return type**

A function that does not return any value specifies **void** type as a return type. In the following example, a function is created without return type.

1. **using** System;
2. **namespace** FunctionExample
3. {
4. **class** Program
5. {
6. // User defined function without return type
7. **public** **void** Show() // No Parameter
8. {
9. Console.WriteLine("This is non parameterized function");
10. // No return statement
11. }
12. // Main function, execution entry point of the program
13. **static** **void** Main(**string**[] args)
14. {
15. Program program = **new** Program(); // Creating Object
16. program.Show(); // Calling Function
17. }
18. }
19. }

**Output:**

This is non parameterized function

### **C# Function: using parameter but no return type**

1. **using** System;
2. **namespace** FunctionExample
3. {
4. **class** Program
5. {
6. // User defined function without return type
7. **public** **void** Show(**string** message)
8. {
9. Console.WriteLine("Hello " + message);
10. // No return statement
11. }
12. // Main function, execution entry point of the program
13. **static** **void** Main(**string**[] args)
14. {
15. Program program = **new** Program(); // Creating Object
16. program.Show("Rahul Kumar"); // Calling Function
17. }
18. }
19. }

**Output:**

Hello Rahul Kumar

A function can have zero or any number of parameters to get data. In the following example, a function is created without parameters. A function without parameter is also known as **non-parameterized** function.

### **C# Function: using parameter and return type**

1. **using** System;
2. **namespace** FunctionExample
3. {
4. **class** Program
5. {
6. // User defined function
7. **public** **string** Show(**string** message)
8. {
9. Console.WriteLine("Inside Show Function");
10. **return** message;
11. }
12. // Main function, execution entry point of the program
13. **static** **void** Main(**string**[] args)
14. {
15. Program program = **new** Program();
16. **string** message = program.Show("Rahul Kumar");
17. Console.WriteLine("Hello "+message);
18. }
19. }
20. }

**Output:**

Inside Show Function

Hello Rahul Kumar

# C# Call By Value

In C#, value-type parameters are that pass a copy of original value to the function rather than reference. It does not modify the original value. A change made in passed value does not alter the actual value. In the following example, we have pass value during function call.

### **C# Call By Value Example**

1. **using** System;
2. **namespace** CallByValue
3. {
4. **class** Program
5. {
6. // User defined function
7. **public** **void** Show(**int** val)
8. {
9. val \*= val; // Manipulating value
10. Console.WriteLine("Value inside the show function "+val);
11. // No return statement
12. }
13. // Main function, execution entry point of the program
14. **static** **void** Main(**string**[] args)
15. {
16. **int** val = 50;
17. Program program = **new** Program(); // Creating Object
18. Console.WriteLine("Value before calling the function "+val);
19. program.Show(val); // Calling Function by passing value
20. Console.WriteLine("Value after calling the function " + val);
21. }
22. }
23. }

**Output:**

Value before calling the function 50

Value inside the show function 2500

Value after calling the function 50

# C# Call By Reference

C# provides a **ref** keyword to pass argument as reference-type. It passes reference of arguments to the function rather than copy of original value. The changes in passed values are permanent and **modify** the original variable value.

### **C# Call By Reference Example**

1. **using** System;
2. **namespace** CallByReference
3. {
4. **class** Program
5. {
6. // User defined function
7. **public** **void** Show(**ref** **int** val)
8. {
9. val \*= val; // Manipulating value
10. Console.WriteLine("Value inside the show function "+val);
11. // No return statement
12. }
13. // Main function, execution entry point of the program
14. **static** **void** Main(**string**[] args)
15. {
16. **int** val = 50;
17. Program program = **new** Program(); // Creating Object
18. Console.WriteLine("Value before calling the function "+val);
19. program.Show(**ref** val); // Calling Function by passing reference
20. Console.WriteLine("Value after calling the function " + val);
21. }
22. }
23. }

**Output:**

Value before calling the function 50

Value inside the show function 2500

Value after calling the function 2500

# C# Params

In C#, **params** is a keyword which is used to specify a parameter that takes variable number of arguments. It is useful when we don't know the number of arguments prior. Only one params keyword is allowed and no additional parameter is permitted after params keyword in a function declaration.

### **C# Params Example 1**

1. **using** System;
2. **namespace** AccessSpecifiers
3. {
4. **class** Program
5. {
6. // User defined function
7. **public** **void** Show(**params** **int**[] val) // Params Paramater
8. {
9. **for** (**int** i=0; i<val.Length; i++)
10. {
11. Console.WriteLine(val[i]);
12. }
13. }
14. // Main function, execution entry point of the program
15. **static** **void** Main(**string**[] args)
16. {
17. Program program = **new** Program(); // Creating Object
18. program.Show(2,4,6,8,10,12,14); // Passing arguments of variable length
19. }
20. }
21. }

**Output:**

2

4

6

8

10

12

14

### **C# Params Example 2**

In this example, we are using object type params that allow entering any number of inputs of any type.

1. **using** System;
2. **namespace** AccessSpecifiers
3. {
4. **class** Program
5. {
6. // User defined function
7. **public** **void** Show(**params** **object**[] items) // Params Paramater
8. {
9. **for** (**int** i = 0; i < items.Length; i++)
10. {
11. Console.WriteLine(items[i]);
12. }
13. }
14. // Main function, execution entry point of the program
15. **static** **void** Main(**string**[] args)
16. {
17. Program program = **new** Program(); // Creating Object
18. program.Show("Ramakrishnan Ayyer","Ramesh",101, 20.50,"Peter", 'A'); // Passing arguments of variable length
19. }
20. }
21. }

**Output:**

Ramakrishnan Ayyer

Ramesh

101

20.5

Peter

A