Lambda Expressions

The Lambda Expression in C# is the shorthand for writing the anonymous function. So we can say that the Lambda Expression in C# is nothing but to simplify the anonymous function in C#. To create a lambda expression in C#, we need to specify the input parameters (if any) on the left side of the lambda operator **=>**, and we need to put the expression or statement block on the other side.

A *lambda expression* is an expression of any of the following two forms:

1. *Expression lambda* that has an expression as its body:

```
(input-parameters) => expression
```

2. *Statement lambda* that has a statement block as its body:

```
(input-parameters) => { <sequence-of-statements> }
```

Any lambda expression can be converted to a delegate type. The delegate type to which a lambda expression can be converted is defined by the types of its parameters and return value. If a lambda expression doesn't return a value, it can be converted to one of the **Action delegate** types; otherwise, it can be converted to one of the **Func delegate** types. For example, a lambda expression that has two parameters and returns no value can be converted to an **Action<T1,T2> delegate**. A lambda expression that has one parameter and returns a value can be converted to a **Func<T,TResult> delegate**. In the following example, the lambda expression x => x * x, which specifies a parameter that's named x and returns the value of x squared, is assigned to a variable of a delegate type:

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

You can also use lambda expressions when you write LINQ in C#, as the following example shows:

```
int[] numbers = { 2, 3, 4, 5 };
var squaredNumbers = numbers.Select(x => x * x);
Console.WriteLine(string.Join(" ", squaredNumbers));
// Output:
// 4 9 16 25
```

The body of a statement lambda can consist of any number of statements; however, in practice there are typically no more than two or three.

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

Input parameters of a lambda expression

You enclose input parameters of a lambda expression in parentheses. Specify zero input parameters with empty parentheses:

```
Action line = () => Console.WriteLine();
```

If a lambda expression has only one input parameter, parentheses are optional:

```
Func<double, double> cube = x => x * x * x;
```

Two or more input parameters are separated by commas:

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

Sometimes the compiler can't infer the types of input parameters. You can specify the types explicitly as shown in the following example:

```
Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;
```

Lambda expressions and tuples

Starting with C# 7.0, the C# language provides built-in support for tuples. You can provide a tuple as an argument to a lambda expression, and your lambda expression can also return a tuple. In some cases, the C# compiler uses type inference to determine the types of tuple components.

You define a tuple by enclosing a comma-delimited list of its components in parentheses. The following example uses tuple with three components to pass a sequence of numbers to a lambda expression, which doubles each value and returns a tuple with three components that contains the result of the multiplications.

```
Func<(int n1, int n2, int n3), (int, int, int)> doubleThem = ns => (2 * ns.n1, 2
* ns.n2, 2 * ns.n3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
// Output:
// The set (2, 3, 4) doubled: (4, 6, 8)
```

Ordinarily, the fields of a tuple are named `Item1`, `Item2`, etc. You can, however, define a tuple with named components, as shown in example.

Lambdas with the standard query operators

LINQ to Objects, among other implementations, have an input parameter whose type is one of the `Func<TResult>` family of generic delegates. These delegates use type parameters to define the number and type of input parameters, and the return type of the delegate. `Func` delegates are very useful for encapsulating user-defined expressions that are applied to each element in a set of source data. For example, consider the `Func<T,TResult>` delegate type:

```
public delegate TResult Func<in T, out TResult>(T arg)
```

The delegate can be instantiated as a Func<int, bool> instance where int is an input parameter and bool is the return value. The return value is always specified in the last type parameter. For example, Func<int, string, bool> defines a delegate with two input parameters, int and string, and a return type of bool. The following Func delegate, when it's invoked, returns Boolean value that indicates whether the input parameter is equal to 5:

```
Func<int, bool> equalsFive = x => x == 5;
bool result = equalsFive(4);
Console.WriteLine(result);   // False
```

You can also supply a lambda expression when the argument type is an Expression<TDelegate>, for example in the standard query operators that are defined in the Queryable type. When you specify an Expression<TDelegate> argument, the lambda is compiled to an expression tree. The following example uses the Count standard query operator:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
Console.WriteLine($"There are {oddNumbers} odd numbers in {string.Join(" ",
numbers)}");
```

The compiler can infer the type of the input parameter, or you can also specify it explicitly. This particular lambda expression counts those integers (n) which when divided by two have a remainder of 1.

Type inference in lambda expressions

When writing lambdas, you often don't have to specify a type for the input parameters because the compiler can infer the type based on the lambda body, the parameter types, and other factors as described in the C# language specification. For most of the standard query operators, the first input is the type of the elements in the source sequence. If you are querying an IEnumerable<Customer>, then the input variable is inferred to be a Customer object, which means you have access to its methods and properties:

```
customers.Where(c => c.City == "London");
```

The general rules for type inference for lambdas are as follows:

- The lambda must contain the same number of parameters as the delegate type.
- Each input parameter in the lambda must be implicitly convertible to its corresponding delegate parameter.
- The return value of the lambda (if any) must be implicitly convertible to the delegate's return type.

Capture of outer variables and variable scope in lambda expressions

Lambdas can refer to *outer variables*. These are the variables that are in scope in the method that defines the lambda expression, or in scope in the type that contains the lambda expression. Variables that are captured in this manner are stored for use in the lambda expression even if the variables would otherwise go out of scope and be garbage collected. An outer variable must be definitely assigned before it can be consumed in a lambda expression. The following example demonstrates these rules:

```
public static class VariableScopeWithLambdas
{
    public class VariableCaptureGame
    {
        internal Action<int> updateCapturedLocalVariable;
        internal Func<int, bool> isEqualToCapturedLocalVariable;
        public void Run(int input)
        {
            int j = 0;
            updateCapturedLocalVariable = x =>
            {
                j = x;
                bool result = j > input;
                Console.WriteLine($"{j} is greater than {input}: {result}");
            };
            isEqualToCapturedLocalVariable = x => x == j;
            Console.WriteLine($"Local variable before lambda invocation: {j}");
            updateCapturedLocalVariable(10);
            Console.WriteLine($"Local variable after lambda invocation: {j}");
        }
    }
    public static void Main()
    {
        var game = new VariableCaptureGame();
        int gameInput = 5;
        game.Run(gameInput);
        int jTry = 10;
        bool result = game.isEqualToCapturedLocalVariable(jTry);
        Console.WriteLine($"Captured local variable is equal to {jTry}:
{result}");
        int anotherJ = 3;
        game.updateCapturedLocalVariable(anotherJ);
        bool equalToAnother = game.isEqualToCapturedLocalVariable(anotherJ);
        Console.WriteLine($"Another lambda observes a new value of captured
variable: {equalToAnother}");
    }
    // Output:
    // Local variable before lambda invocation: 0
    // 10 is greater than 5: True
    // Local variable after lambda invocation: 10
    // Captured local variable is equal to 10: True
    // 3 is greater than 5: False
    // Another lambda observes a new value of captured variable: True
}
```

The following rules apply to variable scope in lambda expressions:

- A variable that is captured will not be garbage-collected until the delegate that references it becomes eligible for garbage collection.
- Variables introduced within a lambda expression are not visible in the enclosing method.
- A lambda expression cannot directly capture an `in`, `ref`, or `out` parameter from the enclosing method.

- A `return` statement in a lambda expression doesn't cause the enclosing method to return.
- A lambda expression cannot contain a `goto`, `break`, or `continue` statement if the target of that `jump` statement is outside the lambda expression block. It's also an error to have a `jump` statement outside the lambda expression block if the target is inside the block.

Example:

```
namespace LambdaExpressionDemo
{
    public class LambdaExpression
    {
        public delegate string GreetingsDelegate(string name);
        static void Main(string[] args)
        {
            GreetingsDelegate obj = (name) =>
            {
                return "Hello " + name + " welcome to SSM Infotech Solutions Pvt
                                                                            Ltd";
            };
            string GreetingsMessage = obj.Invoke("Dhaval");
            Console.WriteLine(GreetingsMessage);
            Console.ReadKey();
        }
    }
}
```