What is a Delegate

A **delegate** is like a "function pointer" in C#. It allows you to store a reference to a method and call that method later. In simpler words, a delegate is a variable that can hold a reference to a function.

Why Use Delegates?

- To **call methods dynamically**.
- To **pass methods as arguments** to other methods.
- To allow **flexibility** in defining which method gets executed.

Basic Example

Here’s a simple analogy: Imagine a remote control (the delegate). You can assign different TV functions (methods) to the remote control and use it to operate the TV.

Delegate Syntax

1. Define a Delegate

You declare a delegate type that specifies the **method signature** (return type and parameters).

```
public delegate void MyDelegate(string message); // A delegate type
```

2. Assign a Method to the Delegate

Create a delegate instance and assign a method to it.

```
MyDelegate del = PrintMessage; // Assign a method to the delegate
```

3. Call the Method Using the Delegate

Use the delegate variable to call the method.

```
del("Hello, Delegates!"); // Calls the method PrintMessage
```

Complete Example

```
using System;
public class Program
{       // Step 1: Define a delegate type
        public delegate void MyDelegate(string message);
        // Step 2: Create a method that matches the delegate signature
        public static void PrintMessage(string message)
        {       Console.WriteLine(message);    }
        public static void Main()
        {       // Step 3: Create a delegate instance and assign the method
        MyDelegate del = PrintMessage;
        // Step 4: Call the method using the delegate
        del("Hello, Delegates!");    }}
```

Output:

Hello, Delegates!

Another Example: Passing a Method as a Parameter

Delegates can be passed as parameters to methods. This allows you to call different methods dynamically.

```
using System;
public class Program
{       // Define a delegate type
        public delegate void Operation(int num1, int num2);
        // Create methods matching the delegate signature
        public static void Add(int a, int b)
        {       Console.WriteLine($"Sum: {a + b}");       }
        public static void Subtract(int a, int b)
        {       Console.WriteLine($"Difference: {a - b}");        }
```

```
    // Method that accepts a delegate as a parameter
    public static void PerformOperation(int x, int y, Operation op)
    {      op(x, y); // Call the delegate  }
    public static void Main()
    {      // Create delegate instances
    Operation addOp = Add;
    Operation subOp = Subtract;
    // Call PerformOperation with different methods
    PerformOperation(10, 5, addOp);  // Output: Sum: 15
    PerformOperation(10, 5, subOp);  // Output: Difference: 5
    }
}
```

Multicast Delegates

A delegate can hold references to multiple methods. This is called a **multicast delegate**.

```
using System;
public class Program
{       public delegate void Notify();
        public static void Method1()
        {       Console.WriteLine("Method1 called");        }
        public static void Method2()
        {       Console.WriteLine("Method2 called");        }
        public static void Main()
        {       Notify del = Method1;   // Assign Method1 to the delegate
        del += Method2;         // Add Method2 to the delegate
        del(); // Calls both Method1 and Method2
        }
}
```

Output:

Method1 called

Method2 called

Built-in Delegates

C# provides some built-in delegates to simplify your work:

1. **Action**: Represents a method that returns `void` and can take up to 16 parameters.
2. **Func**: Represents a method that returns a value and can take up to 16 parameters.
3. **Predicate**: Represents a method that takes one parameter and returns `bool`.

What is `Action`?

`Action` is a built-in delegate that you use when:

- The method **does not return anything** (`void`).
- The method can take **0 to 16 parameters**.

Example 1: Using `Action` with One Parameter

```
using System;
public class Program
{       public static void Greet(string name)
        {       Console.WriteLine($"Hello, {name}!");   }
        public static void Main()
        {
        Action<string> action = Greet; // Assign Greet method to Action
        action("Alice");                // Call the method through Action
        // Output: Hello, Alice!
        }
}
```

`Action<string>` means the method takes one `string` parameter and returns `void`.

Example 2: Using `Action` with Two Parameters

```
using System;

public class Program

{       public static void PrintSum(int a, int b)

        {       Console.WriteLine($"The sum is: {a + b}");   }

        public static void Main()

        {

        Action<int, int> action = PrintSum; // Assign PrintSum method to Action

        action(5, 7);                       // Call the method through Action

        // Output: The sum is: 12

        }

}
```

`Action<int, int>` means the method takes two `int` parameters and returns `void`.

Example 3: Using `Action` with No Parameters

```
using System;

public class Program

{	public static void PrintMessage()

	{	Console.WriteLine("Hello, world!");	}

	public static void Main()

	{

	Action action = PrintMessage; // Assign PrintMessage method to Action

	action();			// Call the method through Action

	// Output: Hello, world!

	}

}
```

`Action` (with no type arguments) means the method takes no parameters and returns `void`.

What is Func?

Func is a built-in delegate that you use when:

- The method **returns a value**.
- The method can take **0 to 16 parameters**.

Example 1: Using Func with One Parameter

```
using System;
public class Program
{      public static int Square(int number)
       {      return number * number;    }
       public static void Main()
       {      Func<int, int> func = Square; // Assign Square method to Func
       int result = func(5);       // Call the method through Func
       Console.WriteLine(result);  // Output: 25  }
}
```

`Func<int, int>` means:

- The method takes one `int` parameter.
- The method returns an `int`.

Example 2: Using `Func` with Two Parameters

```
using System;

public class Program

{       public static int Add(int a, int b)

        {       return a + b;   }

        public static void Main()

        {       Func<int, int, int> func = Add; // Assign Add method to Func

        int result = func(5, 7);            // Call the method through Func

        Console.WriteLine(result);      // Output: 12   }

}
```

`Func<int, int, int>` means:

- The method takes two `int` parameters.
- The method returns an `int`.

Example 3: Using `Func` with No Parameters

```
using System;

public class Program

{       public static string GetMessage()

        {       return "Hello from Func!";      }

        public static void Main()

        {       Func<string> func = GetMessage; // Assign GetMessage method to Func

        string message = func();        // Call the method through Func

        Console.WriteLine(message);             // Output: Hello from Func!     }

}
```

`Func<string>` means:

- The method takes no parameters.
- The method returns a `string`.

What is `Predicate`?

`Predicate` is a built-in delegate that you use when:

- The method **returns a `bool`** (true or false).
- The method takes **exactly one parameter**.

Example 1: Using `Predicate` to Check a Number

```
public class Program
{      public static bool IsEven(int number)
       {      return number % 2 == 0;       }
       public static void Main()
       {
       Predicate<int> predicate = IsEven; // Assign IsEven method to Predicate
       bool result = predicate(4);      // Call the method through Predicate
       Console.WriteLine(result);      // Output: True
       }
}
```

`Predicate<int>` means:

- The method takes one `int` parameter.
- The method returns a `bool`.

Example 2: Using `Predicate` with Strings

```
public class Program
{       public static bool IsLongerThanFive(string str)
        {       return str.Length > 5;  }
        public static void Main()
        {
        Predicate<string> predicate = IsLongerThanFive; // Assign method to Predicate
        bool result = predicate("Hello, world!");       // Call the method through Predicate
        Console.WriteLine(result);                      // Output: True
        }
}
```

`Predicate<string>` means:

- The method takes one `string` parameter.
- The method returns a `bool`.

Key Differences Between `Action`, `Func`, and `Predicate`

Delegate Type	Return Type	Number of Parameters	Example
`Action`	`void`	0 to 16	`Action<int, int>` for `void Add(int a, int b)`
`Func`	Any (e.g., `int`)	0 to 16	`Func<int, int, int>` for `int Multiply(int a, int b)`
`Predicate`	`bool`	Exactly 1	`Predicate<string>` for `bool IsLongerThanFive(string s)`

```
using System;
public class Program
{      public static void PrintGreeting(string name)
       {      Console.WriteLine($"Hello, {name}!");}
       public static void Main()
       {
       Action<string> greet = PrintGreeting; // Assign method to Action delegate
       greet("Alice"); // Output: Hello, Alice!
       }
}
```

Example with **Func**:

```
using System;

public class Program

{

	public static int Multiply(int a, int b)

	{	return a * b;	}

	public static void Main()

	{

	Func<int, int, int> multiplyFunc = Multiply; // Assign method to Func delegate

	int result = multiplyFunc(3, 4);

	Console.WriteLine($"Result: {result}"); // Output: Result: 12

	}

}
```

Example with **Predicate**:

```
using System;

public class Program

{

	public static bool IsEven(int number)

	{	return number % 2 == 0;	}

	public static void Main()

	{

	Predicate<int> isEvenPredicate = IsEven; // Assign method to Predicate delegate

	bool result = isEvenPredicate(10);

	Console.WriteLine($"Is 10 even? {result}"); // Output: Is 10 even? True

	}

}
```

Summary

- A **delegate** is a type-safe function pointer.
- You can use delegates to:
  - Dynamically call methods.
  - Pass methods as arguments.
  - Chain multiple methods (multicast delegates).
- **Built-in delegates** like `Action`, `Func`, and `Predicate` simplify common use cases.