# 01\_Delegates

### **Delegates in C#**

Delegates in C# are type-safe function pointers, meaning they hold references to methods that have a matching signature. This allows for methods to be passed around like variables and invoked dynamically. Delegates are especially useful for scenarios such as event handling, callback methods, and defining custom behaviors.

#### **Key Characteristics of Delegates:**

1. **Type Safety**:
   * The method signature (return type and parameters) must match the delegate's signature. This ensures type safety, meaning the delegate can only point to methods that match its signature.
2. **Syntax Similarity**:
   * Delegate declarations look similar to method declarations, with the keyword delegate preceding them. This makes it easy to understand and remember the syntax.
3. **Instance Creation**:
   * Delegates are similar to classes; you can create instances of them. When creating a delegate instance, you pass the method name as an argument to the delegate's constructor.
4. **Invocation**:
   * Once a delegate is created and points to a method, you can invoke the method through the delegate using the delegate instance.

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### 

#### **Example of a Delegate:**

Here is a sample C# program that demonstrates the use of a delegate:

using System;

// Delegate Declaration

public delegate void HelloFunctionDelegate(string message);

class Pragim

{

public static void Main()

{

// Creating an instance of the delegate and passing the method name

HelloFunctionDelegate del = new HelloFunctionDelegate(Hello);

// Invoking the delegate, which in turn invokes the method

del("Hello from Delegate");

}

// A method that matches the delegate signature

public static void Hello(string message)

{

Console.WriteLine(message);

}

}

#### **Explanation:**

1. **Delegate Declaration**:
   * public delegate void HelloFunctionDelegate(string message);
   * This declares a delegate named HelloFunctionDelegate that can point to any method that returns void and takes a single string parameter.
2. **Delegate Instance Creation**:
   * HelloFunctionDelegate del = new HelloFunctionDelegate(Hello);
   * Here, we create an instance of the delegate and pass the method name Hello to its constructor. The method Hello has the same signature as the delegate.
3. **Delegate Invocation**:
   * del("Hello from Delegate");
   * This line invokes the method Hello through the delegate del, passing in the string "Hello from Delegate".
4. **Method Definition**:
   * public static void Hello(string message)
   * This method matches the delegate's signature and will be called when the delegate del is invoked.

#### **Practical Use Cases for Delegates:**

* **Event Handling**: Delegates are commonly used in event handling, where you subscribe methods to events.
* **Callback Methods**: Delegates can be used to define callback methods, allowing for more flexible and dynamic code.
* **Encapsulation of Method Calls**: Delegates allow methods to be passed as parameters, enabling higher-order functions and more reusable code.

By using delegates, you can create more modular, flexible, and reusable code in C#. They are a fundamental part of the language, especially when dealing with events and asynchronous programming.

The example allows for flexible promotion criteria that can be defined at runtime.

### **Step-by-Step Implementation:**

1. **Define the Delegate:**
   * The delegate EligibleToPromote will be used to pass different promotion criteria.
2. **Create the Employee Class:**
   * The Employee class has properties like Id, Name, Experience, and Salary.
   * The PromoteEmployee method uses the delegate to determine which employees should be promoted based on the criteria provided.
3. **Implement the PromoteEmployee Method:**
   * The method loops through a list of employees and applies the promotion criteria.

### **Implementation:**

using System;

using System.Collections.Generic;

// Define the delegate

public delegate bool EligibleToPromote(Employee employeeToPromote);

public class Employee

{

// Properties of the Employee class

public int Id { get; set; }

public string Name { get; set; }

public int Experience { get; set; }

public int Salary { get; set; }

// Method to promote employees based on a delegate criteria

public static void PromoteEmployee(List<Employee> employeesList, EligibleToPromote isEmployeeEligible)

{

foreach (Employee employee in employeesList)

{

// Delegate determines if the employee is eligible for promotion

if (isEmployeeEligible(employee))

{

Console.WriteLine($"Employee {employee.Name} Promoted");

}

}

}

}

public class Program

{

public static void Main()

{

// Create a list of employees

List<Employee> employees = new List<Employee>

{

new Employee { Id = 1, Name = "John", Experience = 5, Salary = 5000 },

new Employee { Id = 2, Name = "Jane", Experience = 2, Salary = 4000 },

new Employee { Id = 3, Name = "Jake", Experience = 8, Salary = 8000 },

new Employee { Id = 4, Name = "Jess", Experience = 4, Salary = 4500 }

};

// Define promotion criteria based on experience

EligibleToPromote promoteByExperience = emp => emp.Experience >= 5;

Console.WriteLine("Promotion based on Experience:");

Employee.PromoteEmployee(employees, promoteByExperience);

// Define promotion criteria based on salary

EligibleToPromote promoteBySalary = emp => emp.Salary > 4500;

Console.WriteLine("\nPromotion based on Salary:");

Employee.PromoteEmployee(employees, promoteBySalary);

}

}

### 

### **Explanation:**

1. **Delegate Definition**:
   * public delegate bool EligibleToPromote(Employee employeeToPromote);
   * This delegate takes an Employee object as input and returns a bool indicating whether the employee should be promoted.
2. **Employee Class**:
   * Properties like Id, Name, Experience, and Salary represent the employee's details.
   * The PromoteEmployee method takes a list of Employee objects and a delegate EligibleToPromote. It iterates through the list and checks if each employee meets the promotion criteria defined by the delegate.
3. **Main Method**:
   * In Main, a list of employees is created.
   * Two different promotion criteria are defined: one based on experience (promoteByExperience) and another based on salary (promoteBySalary).
   * The PromoteEmployee method is called twice with different criteria, demonstrating the flexibility of using delegates for conditional logic.

### **Output:**

When you run the program, it will print:

plaintext

Copy code

Promotion based on Experience:

Employee John Promoted

Employee Jake Promoted

Promotion based on Salary:

Employee John Promoted

Employee Jake Promoted

This approach makes the PromoteEmployee method flexible and reusable, as the promotion criteria can be changed without modifying the method itself, simply by passing different delegate instances.

## Multicast delegate

is a delegate that can reference multiple methods. When invoked, it calls all the methods in its invocation list in the order they were added. This is useful for scenarios where you want to notify multiple methods or perform multiple operations in response to a single event.

### **Key Points:**

1. **Invocation Order:** The methods in the multicast delegate's invocation list are called in the order they were added.
2. **Return Type:** If the delegate has a return type, only the value of the last invoked method will be returned.
3. **Output Parameters:** If the delegate has an output parameter, the value assigned by the last method in the invocation list will be returned.

### **Code Examples and Expected Outputs**

#### **Example 1: Basic Multicast Delegate with void Return Type**

using System;

namespace Sample

{

public delegate void SampleDelegate();

public class Sample

{

static void Main()

{

SampleDelegate del1 = new SampleDelegate(SampleMethodOne);

SampleDelegate del2 = new SampleDelegate(SampleMethodTwo);

SampleDelegate del3 = new SampleDelegate(SampleMethodThree);

SampleDelegate del4 = del1 + del2 + del3 - del2;

del4();

}

public static void SampleMethodOne()

{

Console.WriteLine("SampleMethodOne Invoked");

}

public static void SampleMethodTwo()

{

Console.WriteLine("SampleMethodTwo Invoked");

}

public static void SampleMethodThree()

{

Console.WriteLine("SampleMethodThree Invoked");

}

}

}

**Expected Output:**

SampleMethodOne Invoked

SampleMethodThree Invoked

* SampleMethodOne and SampleMethodThree are invoked because SampleMethodTwo was removed using the - operator.

#### **Example 2: Basic Multicast Delegate with void Return Type (Using +=)**

using System;

namespace Sample

{

public delegate void SampleDelegate();

public class Sample

{

static void Main()

{

SampleDelegate del = new SampleDelegate(SampleMethodOne);

del += SampleMethodTwo;

del += SampleMethodThree;

del -= SampleMethodTwo;

del();

}

public static void SampleMethodOne()

{

Console.WriteLine("SampleMethodOne Invoked");

}

public static void SampleMethodTwo()

{

Console.WriteLine("SampleMethodTwo Invoked");

}

public static void SampleMethodThree()

{

Console.WriteLine("SampleMethodThree Invoked");

}

}

}

**Expected Output:**

SampleMethodOne Invoked

SampleMethodThree Invoked

* Similar to the previous example, SampleMethodTwo is removed, so only SampleMethodOne and SampleMethodThree are invoked.

#### 

#### 

#### 

#### 

#### 

#### **Example 3: Multicast Delegate with int Return Type**

using System;

namespace Sample

{

public delegate int SampleDelegate();

public class Sample

{

static void Main()

{

SampleDelegate del = new SampleDelegate(SampleMethodOne);

del += SampleMethodTwo;

int ValueReturnedByDelegate = del();

Console.WriteLine("Returned Value = {0}", ValueReturnedByDelegate);

}

public static int SampleMethodOne()

{

return 1;

}

public static int SampleMethodTwo()

{

return 2;

}

}

}

**Expected Output:**

Returned Value = 2

* The delegate calls SampleMethodOne and SampleMethodTwo, but only the return value of SampleMethodTwo is captured because it is the last method in the invocation list.

#### **Example 4: Multicast Delegate with an out Parameter**

using System;

namespace Sample

{

public delegate void SampleDelegate(out int Number);

public class Sample

{

static void Main()

{

SampleDelegate del = new SampleDelegate(SampleMethodOne);

del += SampleMethodTwo;

int ValueFromOutPutParameter = -1;

del(out ValueFromOutPutParameter);

Console.WriteLine("Returned Value = {0}", ValueFromOutPutParameter);

}

public static void SampleMethodOne(out int Number)

{

Number = 1;

}

public static void SampleMethodTwo(out int Number)

{

Number = 2;

}

}

}

**Expected Output:**

Returned Value = 2

* The delegate sets the value of Number in SampleMethodOne and SampleMethodTwo, but only the value assigned by SampleMethodTwo is captured because it is the last method in the invocation list.

### **Where to Use Multicast Delegates?**

Multicast delegates are commonly used in scenarios like implementing the **Observer Design Pattern** (also known as **Publish/Subscribe**). In such patterns, multiple observers (methods) can subscribe to an event (delegate), and when the event is triggered, all the observers are notified.

This approach simplifies the implementation of event-driven systems where multiple components need to react to a single event.

# 02\_Generics

Generics were introduced in C# 2.0 to allow developers to create classes, methods, and interfaces that are type-safe and reusable across different data types without sacrificing performance. Generics enable you to design code components that are decoupled from specific data types, making them more versatile and flexible.

### **Example Without Generics**

Consider the following example where we have a method AreEqual(int value1, int value2) in the Calculator class. This method only works with integers:

using System;

namespace Pragim

{

public class MainClass

{

private static void Main()

{

bool Equal = Calculator.AreEqual(1, 2);

if (Equal)

{

Console.WriteLine("Equal");

}

else

{

Console.WriteLine("Not Equal");

}

}

}

public class Calculator

{

public static bool AreEqual(int value1, int value2)

{

return value1 == value2;

}

}

}

**Problem:**

* This AreEqual() method is tightly coupled to the int data type. If you try to use it with any other data type, like string, you'll get a compiler error:

bool Equal = Calculator.AreEqual("A", "B"); // Compile-time error

### **Approach with object Type**

To make AreEqual() reusable across different data types, you might consider using object parameters, like this:

using System;

namespace Pragim

{

public class MainClass

{

private static void Main()

{

bool Equal = Calculator.AreEqual("A", "B");

if (Equal)

{

Console.WriteLine("Equal");

}

else

{

Console.WriteLine("Not Equal");

}

}

}

public class Calculator

{

public static bool AreEqual(object value1, object value2)

{

return value1 == value2;

}

}

}

**Problems with object Type:**

1. **Type Safety:** The AreEqual() method is no longer type-safe. It allows comparisons between different types, like an integer and a string, which doesn't make sense and could lead to runtime errors.
2. **Performance:** Using object results in performance degradation due to **boxing** and **unboxing** when dealing with value types.

### **Solution with Generics**

Generics provide a solution to both of these problems. By making the AreEqual() method generic, you can ensure type safety and avoid performance issues:

using System;

namespace Pragim

{

public class MainClass

{

private static void Main()

{

bool Equal = Calculator.AreEqual<int>(2, 1);

if (Equal)

{

Console.WriteLine("Equal");

}

else

{

Console.WriteLine("Not Equal");

}

}

}

public class Calculator

{

public static bool AreEqual<T>(T value1, T value2)

{

return value1.Equals(value2);

}

}

}

### **Explanation:**

* **Generic Method:** public static bool AreEqual<T>(T value1, T value2)  
  The method is defined using a type parameter T within angular brackets. This makes the method generic.
* **Type Specification:**When calling the AreEqual() method, the client code must specify the data type to use with the method:
  + For integers: bool Equal = Calculator.AreEqual<int>(2, 1);
  + For strings: bool Equal = Calculator.AreEqual<string>("A", "B");

### **Advantages of Using Generics:**

1. **Type Safety:** The compiler ensures that only the specified data type is used with the method, preventing invalid comparisons.
2. **Performance:** Since no boxing or unboxing is required, the code runs more efficiently.
3. **Reusability:** The same method can be reused across different data types without rewriting the logic.

### **Conclusion**

Generics are a powerful feature in C# that promotes code reusability, type safety, and performance. They allow you to write more flexible and maintainable code by decoupling methods and classes from specific data types. You can also apply generics to interfaces and delegates, further expanding their utility in various programming scenarios.