# Delegates in C#

Delegates in C# are a powerful feature that allow methods to be encapsulated in objects, enabling methods to be passed as parameters, stored in data structures, or returned from other methods.

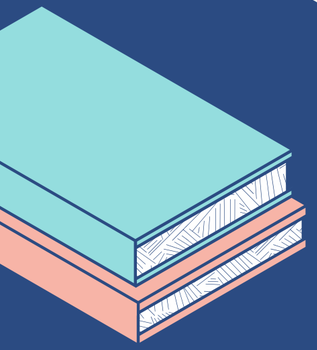
They serve as a bridge between the caller and callee, providing a flexible and safe way to handle function pointers. Delegates support single-casting and multi-casting, which refers to invoking one or more methods through a single delegate instance.

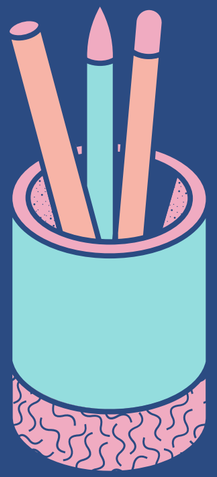
They form the foundation of event-driven programming in C#, making them integral to many .NET patterns and practices. Let us explore those.

@lijotech

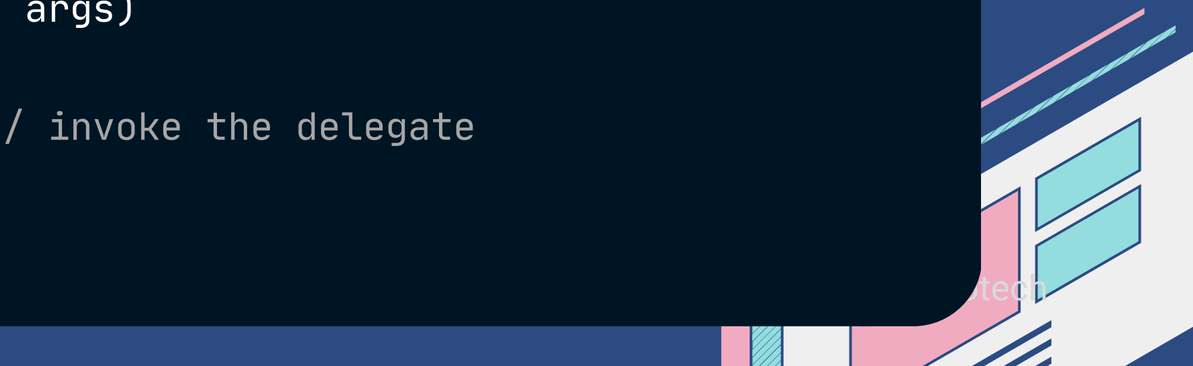
# Action Delegates

These are built-in delegates provided by .NET that represent methods with a void return type. They are used when you're working with a method that doesn't return a value





```
Action<string> myAction = (message) =>
{
    Console.WriteLine(message);
};

public static void Main(string[] args)
{
    myAction("Hello, World!"); // invoke the delegate
}
```



# Named Delegates

These are explicitly defined by the user. It also does not return any value.

```
delegate void MyDelegate(string msg); // declare a delegate

// This is the method that matches the signature of the delegate.
void MyMethod(string message)
{
    Console.WriteLine(message);
}

public static void Main(string[] args)
{
    MyDelegate del = MyMethod; // instantiate the delegate
    del("Hello, World!"); // invoke the delegate
}
```

@lijotech

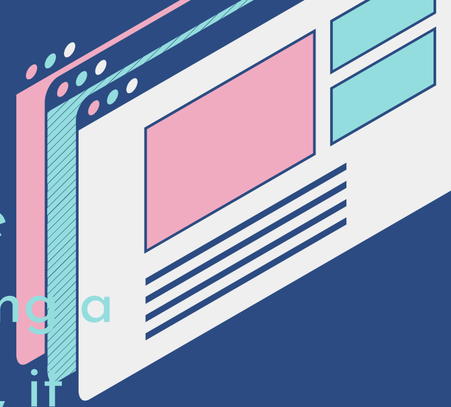
# Func Delegates

These are also built-in delegates provided by .NET that represent methods with a non-void return type. They are used when you want to encapsulate a method that returns a value and can take zero to sixteen input parameters. The type of the last parameter is considered as the return type of the delegate.

```
Func<int, int, int> add = (a, b) =>
{
    return a + b;
};

public static void Main(string[] args)
{
    int result = add(1, 2); // invoke the delegate
    Console.WriteLine(result); // prints 3
}
```

@lijotech

# Multicast Delegates



These delegates can hold references to more than one method. They are used to invoke multiple methods using a single delegate. When a multicast delegate is invoked, it calls the methods in the order they were added.

```
delegate void MyDelegate(string msg); // declare a delegate

void Method1(string message)
{
    Console.WriteLine("Method1: " + message);
}

void Method2(string message)
{
    Console.WriteLine("Method2: " + message);
}
```

```
public static void Main(string[] args)
{
 MyDelegate del = Method1; // instantiate the delegate
 del += Method2; // add another method to the delegate
 del("Hello, World!"); // invoke the delegate
}
```

@lijotech

# Predicate Delegates

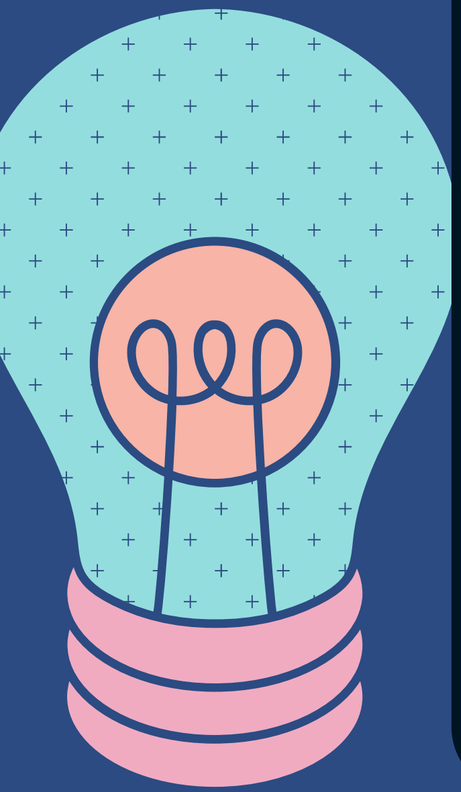
These are used when we want to create a delegate that represents a method that returns a bool and takes one input parameter.

```
Predicate<string> isUpperCase = IsUpperCase;

bool IsUpperCase(string str)
{
    return str.Equals(str.ToUpper());
}

public static void Main(string[] args)
{
    bool result = isUpperCase("Hello World");
    Console.WriteLine(result); // prints False
}
```

@lijotech

# Event Delegates

These are used to declare events. They act as a event handler that will be called in response to an event being triggered, allowing for a flexible and decoupled design where an object can notify other objects when something of interest occurs.



```
public delegate void MyDelegate();

public class MyClass
{
    public event MyDelegate MyEvent;

    public void RaiseEvent()
    {
        MyEvent?.Invoke();
    }
}
```

```
public static void Main(string[] args)
{
 MyClass obj = new MyClass();
 obj.MyEvent += () =>
Console.WriteLine("Event Raised");
 obj.RaiseEvent(); // prints "Event Raised"
}
```

@lijotech

# Stay Focused & Happy Coding

Follow Lijo Sebastian for more coding tips

@lijotech