# Delegates

And what are **Delegates** in C#? A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance. Delegates are used to pass methods as arguments to other methods. Event handlers are nothing more than methods that are invoked through delegates. You create a custom method, and a class such as a windows control can call your method when a certain event occurs. Any method from any accessible class or struct that matches the delegate type can be assigned to the delegate. The method can be either static or an instance method. This flexibility means you can programmatically change method calls, or plug new code into existing classes.

Στην C#, οι **delegates** είναι τύποι αναφοράς που επιτρέπουν την αποθήκευση αναφορών σε μεθόδους. Χρησιμοποιούνται κυρίως για να εκτελούν μεθόδους asyncronously ή να περνούν μεθόδους ως παραμέτρους. Οι delegates μπορούν να θεωρηθούν σαν pointers για μεθόδους.

Βασικά χαρακτηριστικά των delegates:

* Επιτρέπουν τη χρήση callbacks.
* Μπορούν να συνδυαστούν (multicast delegates).
* Υποστηρίζουν τις ανώνυμες μεθόδους (anonymous methods) και τις εκφράσεις lambda.

# Παραδείγματα χρήσης:

## Βασικός Delegate:

public delegate void MyDelegate(string message);

public class Program

{

public static void DisplayMessage(string message)

{

Console.WriteLine(message);

}

static void Main()

{

MyDelegate del = DisplayMessage;

del("Γειά σου, Κόσμε!");

}

}

## Multicast Delegate:

public delegate void MyMultiDelegate(string message);

public class Program

{

public static void ShowMessage1(string message)

{

Console.WriteLine("Message 1: " + message);

}

public static void ShowMessage2(string message)

{

Console.WriteLine("Message 2: " + message);

}

static void Main()

{

MyMultiDelegate del = ShowMessage1;

del += ShowMessage2; // Προσθήκη δεύτερης μεθόδου

del("Γειά σου, Κόσμε!"); // Καλεί και τις δύο μεθόδους

}

}

## Anonymous Method:

public delegate void MyDelegate(int number);

public class Program

{

static void Main()

{

MyDelegate del = delegate(int n) {

Console.WriteLine("Ο αριθμός είναι: " + n);

};

del(5); // Καλεί την ανώνυμη μέθοδο

}

}

## Lambda Expressions

public delegate int MathOperation(int x, int y);

public class Program

{

static void Main()

{

MathOperation add = (x, y) => x + y;

MathOperation multiply = (x, y) => x \* y;

Console.WriteLine("Άθροισμα: " + add(5, 4));

Console.WriteLine("Πολλαπλασιασμός: " + multiply(5, 4));

}

}

**Understanding the** Program.cs **File in .NET Core 8: An In-Depth Exploration**

In .NET Core 8, the Program.cs file serves as the main entry point for any web application, responsible for setting up the application's configuration, dependency injection (DI), middleware, and the HTTP request pipeline. Over recent versions, particularly since .NET 6, the Program.cs file has been simplified into a minimalistic, streamlined structure, combining what used to be split into the Startup.cs and Program.cs files into a single unified approach.

In this detailed blog post, we’ll dive deeper into each section of the Program.cs file, discussing not only the basic components but also additional configurations, advanced features, and customization options that are relevant in .NET Core 8.

**Key Components of the** Program.cs **File**

1. **Namespaces and Usings: Importing Essential Libraries**
2. **Creating the Builder:** WebApplication.CreateBuilder(args)
3. **Configuring Services**: builder.Services
4. **Building the Application:** builder.Build()
5. **Configuring the Middleware Pipeline**
6. **Environment-Specific Configuration**
7. **Running the Application:** app.Run()

Let’s break each component down into greater detail.

**1. Namespaces and Usings: Importing Essential Libraries**

At the top of the Program.cs file, we see several namespaces that bring in functionality necessary to build and run an ASP.NET Core application. These imports provide access to essential features such as configuring middleware, setting up routing, and registering services for dependency injection.

**using** Microsoft**.**AspNetCore**.**Builder**;**

**using** Microsoft**.**Extensions**.**DependencyInjection**;**

**using** Microsoft**.**Extensions**.**Hosting**;**

* Microsoft.AspNetCore.Builder: This namespace contains classes and extension methods for building the HTTP request pipeline, such as WebApplication, UseRouting(), and UseMiddleware().
* Microsoft.Extensions.DependencyInjection: This namespace provides methods for configuring services in the dependency injection (DI) container.
* Microsoft.Extensions.Hosting: Contains classes that support hosting functionality for the application, such as environment detection (IHostEnvironment) and running long-lived services.

If your app requires additional functionality, such as logging, authentication, or database interaction, you will need to import the appropriate namespaces here as well, e.g.:

**using** Microsoft**.**Extensions**.**Logging**;**

**using** Microsoft**.**EntityFrameworkCore**;**

**2. Creating the Builder:** WebApplication.CreateBuilder(args)

var builder **=** WebApplication.CreateBuilder**(**args**)**;

The CreateBuilder method is a high-level entry point that abstracts much of the setup previously required in .NET Core 2.x and 3.x applications. This single line of code does the following:

* **Loads Configuration**: It loads configuration settings from various sources like appsettings.json, environment variables, command-line arguments, and any custom configuration sources you add.
* **Sets Up Logging**: It configures logging based on the current environment (e.g., development, production) and any specified logging settings.
* **Initializes Dependency Injection**: The DI container is initialized, allowing services to be registered and injected throughout the app.

This method simplifies the application startup process by handling all the default setup needed to run the app while still allowing customization if required.

**Advanced Customizations**

You can modify the builder configuration to include additional settings, such as adding support for custom configuration providers or enhancing logging:

builder**.**Configuration**.**AddJsonFile**(**"customsettings.json"**,** optional**:** **true,** reloadOnChange**:** **true);**

builder**.**Logging**.**AddConsole**();**

Here, you can load a custom configuration file or modify logging behavior.

**3. Configuring Services:** builder.Services

After creating the builder, you can configure the services that will be used across the application. This is where Dependency Injection (DI) is set up by registering services within the DI container, allowing them to be injected into controllers, middleware, and other parts of the app.

builder**.**Services**.**AddControllers**();**

* **AddControllers()**: Registers the necessary services for using MVC pattern, enabling controllers to handle HTTP requests. It adds support for API controllers and routing, setting the stage for the MapControllers**()** call later in the middleware setup.

**Additional Service Configurations**

In real-world applications, this section of the Program.cs file can include many other services. Some common examples:

* **Database Context**:

builder**.**Services**.**AddDbContext**<**MyDbContext**>(**options **=>**

options**.**UseSqlServer**(**builder**.**Configuration**.**GetConnectionString**(**"DefaultConnection"**)));**

This registers MyDbContext with the DI container, configuring it to use SQL Server based on connection string settings from the configuration file.

* **Authentication & Authorization**:

builder**.**Services**.**AddAuthentication**(**"Bearer"**)**

**.**AddJwtBearer**(**options **=> {**

options**.**Authority **=** "https://your-identity-provider"**;**

options**.**Audience **=** "your-api"**;**

**});**

builder**.**Services**.**AddAuthorization**();**

This configures the app to use JWT authentication for securing APIs and adds authorization services to control access to endpoints.

* **Swagger (API Documentation)**:

Swagger helps in generating API documentation and testing endpoints:

builder.Services.AddSwaggerGen**()**;

Once this is registered, you can configure Swagger middleware later in the pipeline to automatically generate and serve API documentation.

**Dependency Injection with Scopes**: Services in .NET Core are typically registered with one of three lifetimes:

* **Singleton**: A single instance of the service is created and shared across the entire app.
* **Scoped**: A new instance of the service is created for each request.
* **Transient**: A new instance is created every time the service is requested.

//Singleton

builder.Services.AddSingleton**<**IMyService, MyService**>()**;

//Scoped

builder.Services.AddScoped**<**IUserService, UserService**>()**;

//Transient  
builder.Services.AddTransient**<**ILoggingService, LoggingService**>()**;

* **Adding HttpClient**:

Modern applications often consume external APIs. Instead of manually creating instances of HttpClient, you should use the built-in HttpClientFactory:

builder.Services.AddHttpClient**()**;

* **Configure Options Pattern**:

You can use the Options Pattern to bind configuration settings from appsettings.json to strongly typed classes. For example, if you have the following configuration:

"AppSettings": {  
 "ApiKey": "your-api-key"  
**}**

You can bind this to a class:

builder.Services.Configure**<**AppSettings**>(**builder.Configuration.GetSection**(**"AppSettings"**))**;

* **Caching**:

builder.Services.AddDistributedMemoryCache**()**;  
builder.Services.AddSession**()**;

Adds support for distributed caching and session management in the app.

* **Third-Party Services**:You can also register third-party services, such as SendGrid for email, Stripe for payments, etc., using this section.

**Using** IServiceCollection **Extensions:**

IServiceCollection can be extended by adding custom service configurations. You can write your own extension methods to simplify service registration:

public static class MyCustomServiceExtensions  
**{**  
 public static IServiceCollection AddMyServices**(**

this IServiceCollection services

**)**  
 **{**  
 services.AddSingleton**<**IMyService, MyService**>()**;  
 //Add more services here  
 return services;  
 **}**  
**}**  
  
//Registering in Program.cs  
builder.Services.AddMyServices**()**;

**4. Building the Application:** builder.Build()

var app **=** builder.Build**()**;

Once services have been registered, the Build() method is called to create the WebApplication object. This object encapsulates the application and its entire configuration, ready for use.

This is an important step, as it finalizes the setup process, resolves all services from the DI container, and prepares the HTTP request pipeline.

**5. Configuring the Middleware Pipeline**

After building the app, the next step is to configure the middleware. The middleware pipeline is responsible for handling all incoming HTTP requests, and it defines how requests are processed by the application.

if **(**app.Environment.IsDevelopment**())**  
**{**  
 app.UseDeveloperExceptionPage**()**;  
**}**  
else  
**{**  
 app.UseExceptionHandler**(**"/Home/Error"**)**;  
 app.UseHsts**()**;  
**}**  
  
app.UseHttpsRedirection**()**;  
app.UseStaticFiles**()**;  
app.UseRouting**()**;  
app.UseAuthorization**()**;  
app.MapControllers**()**;

**Common Middleware Components:**

* app.Environment.IsDevelopment(): Checks whether the app is running in the development environment. This is useful for configuring environment-specific behavior.
* UseDeveloperExceptionPage(): Displays detailed error information in development mode, which is useful for debugging.
* UseExceptionHandler("/Home/Error"): In production, this middleware handles exceptions and directs users to a custom error page.
* UseHsts(): Enforces HTTP Strict Transport Security (HSTS), improving security by instructing browsers to use HTTPS for future requests.
* UseHttpsRedirection(): Automatically redirects HTTP requests to HTTPS to ensure encrypted communication.
* UseStaticFiles(): Serves static files such as CSS, JavaScript, and images from the wwwroot folder.
* UseRouting(): Enables endpoint routing, which maps incoming requests to the appropriate route handlers like controllers.
* UseAuthorization(): Adds authorization checks, ensuring that only authenticated and authorized users can access protected routes.
* MapControllers(): Maps HTTP routes to controller actions, enabling the MVC architecture.

**Custom Middleware**

You can add custom middleware to the pipeline to perform tasks such as logging, request validation, or even altering HTTP responses. A custom middleware might look like this:

app.Use**(**async **(**context, next**)** **=>**  
**{**  
 //Perform an action before the next middleware  
 Console.WriteLine**(**"Processing request..."**)**;  
   
 await next.Invoke**()**; //Call the next middleware in the pipeline  
  
 //Perform an action after the next middleware  
 Console.WriteLine**(**"Response sent."**)**;  
**})**;

**6. Environment-Specific Configuration**

The Program.cs file supports environment-specific configurations, allowing you to change behaviors or settings depending on whether the app is running in Development, Production, or Staging.

if **(**app.Environment.IsDevelopment**()){**  
 app.UseDeveloperExceptionPage**()**;  
**}**else**{**  
 app.UseExceptionHandler**(**"/Home/Error"**)**;  
 app.UseHsts**()**;  
**}**

**Environment Variables:**

You can define different environment variables by setting the ASPNETCORE\_ENVIRONMENT variable. For example, in a development machine, you might set:

set ASPNETCORE\_ENVIRONMENT**=**Development

This helps you load different appsettings files like appsettings.Development.json or apply different configurations.

**7. Running the Application:** app.Run()

app.Run**()**;

The Run**()** method starts the application and begins listening for incoming HTTP requests. At this point, the middleware pipeline is fully configured, and the app is ready to handle requests. This is where the long-running process begins that keeps the web server alive and responsive to incoming traffic.

**Advanced Features in .NET Core 8**

While the basic structure of Program.cs remains relatively straightforward, .NET Core 8 brings additional features and flexibility:

**Minimal APIs**

Starting with .NET 6 and continuing in .NET 8, Minimal APIs allow you to build lightweight, high-performance APIs without the overhead of the traditional MVC structure. Here’s an example of configuring a minimal API within Program.cs:

var app **=** builder.Build**()**;  
  
app.MapGet**(**"/hello", **()** **=>** "Hello World!"**)**;  
  
app.Run**()**;

This is a quick way to set up API endpoints directly in Program.cs, ideal for microservices or simple applications.

**Background Services**

.NET Core 8 also allows you to register long-running services that can execute in the background:

builder.Services.AddHostedService**<**BackgroundWorker**>()**;

Here, BackgroundWorker could be a class implementing IHostedService that runs background tasks while the app is running.

**Conclusion**

The Program.cs file in .NET Core 8 is the foundational entry point for configuring and running your application. By understanding its components—such as creating the application builder, registering services, configuring the middleware pipeline, and running the application—you gain control over how your application behaves, performs, and scales.

Whether you’re building a simple API, a full-featured web app, or leveraging advanced features like background services, mastering the Program.cs file equips you with the tools to build modern, efficient, and robust applications in .NET Core 8.