**GAME PROJECT PART**

Hello again, here I will try to talk you through a simple console game project. This project is basically a project based on a character and the structures of that character. It is a console game with 2 characters as Player and NPC and their own structures.

I am not going to follow a sequence here, so I will want you to focus carefully on the moment.

Now let's review the topics by going through the code in sequence.

**GameRun.cs**

Here we see a loading function**. (Line 17)**

Let's review it. As we can see, we haven't sent any parameters here.

But when we go to the function itself, we see that it actually takes 3 parameters. **(Line 81)**

Here, as you can see when defining the parameters of the function, we can give default values. This means that if we don't send a value to that parameter, it will accept it by default.

As you can see where we call them, there are 2 uses, 1 of them does not send any parameters, while the other one defines the first parameter as 15. (Line 34)

So how would we do if we wanted to send the 3rd parameter? Here we will send the value of the parameter using the name we gave for the parameter. When we open parentheses like this, it shows us the parameter names. Here, if I want to change the loadingText value, I send it this way by writing the parameter name and its value next to it. The others remain as default. (Line 36)

**ReturnMultiParameter.cs**

While we're talking about functions and parameters, I'd like to share a little known and underused construct that C# supports. Multiple Parameter Returns.

Sometimes returning a single parameter is not enough, but you don't want to create a class or struct for this structure since it will be a one-time use. Then you can easily get the job done by using multiple parameters in this way.

As you can see here, we write the types and names of the types we will return next to the Access modifier of our engine. Then we match them with their values in the return section. Finally, we can use them directly with the names we give them where we call the function. (ReturnMultiParameter)

There are also other ways to do this without using return. As you can see here, there are ref and out keywords. Since these send the reference of the variable on ram to the function you call, you change the value of that variable wherever it is used. So you can change your values without the need for a return. (REF & OUT)

Okay. We are done with functions and paramets for now. Let's go back to our main function. (GameRun.cs)

We go to our **CreateNPC function**. Here we see that we are creating different objects from our NonPlayer class. (Line 143)

As you can see when creating the object here, it was created in 2 different ways. One with the only new keyword and the other with new NonPlayer.

In the newer versions of C#, if we give the type of the left side, we don't need to repeat it on the right side.

We can call our constructor directly by typing new. Or we can continue with the classic method. The usage option is optional for you.

We are continue. Here we see an NPCTypeEnums. (NPCTypeEnums.cs)

Let's talk about enums shortly. Let's not see the description field at first, I will explain it later.

Enum basically in our code base, if we have types or kinds we need in a model, class or entity, or types we use in general, it allows us to create it in a more organized and readable way and use it easily.

**TestEnum.cs**

For example, here you see the seasons below. There is only 1 next to January, the reason for this is to start the first value of the enum from 1, otherwise it continues starting from 0. Here we can give different values to each of them. It can be adjusted completely according to the situation. (Line 15)

Below we see a code part about its usage. Here, there is a code structure that executes the operations in the month sent with a switch-case structure.

This function basically takes an enum value of type Months and sends it to where it needs to go according to switch-cases and executes the operation. It's very readable, we can understand what will happen in which month and what type of parameters the function will take.

Couldn't we do this structure here without using Enum? Of course we could, but the readability would be much lower and it would be error prone.

For example, since we know that it is the month and the name of the function is adapted, there is no problem, we can understand it. (Line 85)

It would have been more difficult to understand if we hadn't given it a consistent name.

Is it more legible to write 1,2,3,4 or the names of the months? The names of the months, of course.

Also, one of the important things is that it is error-prone.

The user can pass any int value here, for example 55 has no equal value.

It either returns default value or throws an error if you don't write default. But in the model made with the previous Enum, since we cannot send a value other than the enum types you give to the function, everything will be under our control.

**NonPlayer.cs**

As you can see here, we can assign the types of NPC characters to the Type field and create operations according to them. (Line 3)

For example, here we see the Voice function in the NonPlayer class. It's a method that tells only the name and type of the character. Here we can assign our character from the Enum types we created before. And we are sure that they are the only ones.

You can also see here a function where we can get the direct name of the Enum we send. (Line 12)

Let's look at the definition here. ( right-click on Enum and go to defination then find GetName )

As you can see, we can see what the function does with its description. This is how you can easily understand and use the system functions you will use on the track.

Here is information on what parameters it takes, what it returns and what errors it throws.

Also, if you pay attention here, we see a method overloading event. We have 2 methods named GetName but with different signatures.

In the 2nd one we see a generic function structure. So what are these generics? (Line 94 in Enum.cs)

Generics is a structure that allows the class, method, interface or parameter we create to work with any type that fits the structure of the template we have created, not for a specific type.

So, it prevents us from writing methods and classes over and over again by specifying the type each time and allows us to build a better quality, manageable and flexible code structure. If we want to explain it superficially, we can define Generics in this way.

When we say Generics, the keywords T and where should come to mind. Like T, TEnum, TKey, TValue, TEntity. This is a pattern and starts with a capital T.

Although the letter T scares you at first, it is actually a classic like the variable i that we use constantly in the for loop. The letter T is usually used because it stands for type.

Where allows us to limit the types that our method can take.

For example, in this GetName function, the generic parameter TEnum can only take values of type struct and Enum

**Generics.cs**

Don't worry! We use this structure in Unity without realizing it.

For example, when we look at GetComponent, we can use it by giving our type where we specify the generic type. We use this structure frequently in Unity. Here we used the Rigibody2d component for example. (Line 37 – ComponentOp())

The FindAllCharecterByType method we created here takes a Character type for example and returns a list of Character types. (Line-24)

Here we see how the method (TestFindCharacter) is used. You can say that you wrote character next to Where but here you used NonPlayer and Player. This is because Player and NonPlayer classes are subclasses inherited from the Character class. As I explained in the OOP presentation. (Line – 46)

So the more subclasses we have, the more we can use them on this function. This saves us from writing the same code over and over again.

Or we don't need to go too far, we actually use it all the time when creating a list. (Line 14-17)

As you can see here, we have created a list of character, string, integer or Vector type. Here we can see that the List structure is generic. What would happen if it wasn't this way?

Here we can also use the ArrayList structure. As you can see, it does not ask for any type information when creating it. So we can add values of different types inside. The problem here is that it doesn't provide us with Type-safe structure. In other words, it does not provide a safe code structure during runtime because we cannot be sure what type of data it contains. That's why it is recommended to use generic list.

Having talked about List and ArrayList, let's talk about Collections in C# now.

**Collections.cs**

Basically, we can categorize collections as Generic and Non-Generic, but there are different categories such as Specialized and Concurrent, but we will review these 2.

Generic ones are the recommended constructs and non-generic ones are not recommended because they do not provide type safety. In other words, since it does not contain a certain type int, string, float, etc., it can crash your project due to an overlooked structure during runtime.

Our first structure is the List structure that we use and know the most. As you can see, it is a structure that we can create as int, string and Vector and add only these types into it. As you said before, it is one of the most used and most useful structures. It is very easy to use. It has many methods and properties such as create, add, Clear, AddRange and learn the number of elements. (Line 17-35)

If you are wondering what kind of operations you can do here, you can go to the definition or the microsoft documentation I gave above and find out what methods or properties there are and what they do. (Go to defination in List)

Let's go to Defination for example.

- Count : Gets the number of elements contained

- Capacity: Gets or sets the total number of elements the internal data structure can hold without resizing.

What does this mean, for example, when you first create it, Count and Capacity are 0. But when you add the first element, the list is resized and after Count 1, Capacity becomes 4. In this way, it continues to increase as 4,8,16,32,64.

- AddRange: Adds the elements of the specified collection to the end of the list.

It is one of the ways to add elements to the list. You can either add them one by one or you can add them as a list.

As you can see here, we have created the even and odd number lists separately and then we add them into the numbers list. (Line 23-27 in collection.cs)

Dictionary is next. As you can see in this structure, it is created with 2 Generic structures. We can say that it is a key value pair consisting of key and value. (Line - 38)

Dictionary is different from a list in that it works with O(1). So if you are going to search in it, it finds where it is in one go. This is because Key has a unique structure. There cannot be more than one of the same key in a Dictionary.

For example, we can use the id of the users as key and add location to the value side. In this way, we can have a user-location data.

Like the list, here you can go to the defination and examine the methods and properties.

**LinkedList is next.** LinkedList is an example of a dynamic data structure. You can do most of the operations in the list. The difference from a list is that it is much more performant because it holds data with pointers. But since there is no random access, its usage is more limited. (Line – 53)

So you can't tell me that I want to reach the 6th person as it is in the list. (Line – 58)

Example of where linked lists are used in game development is to implement a waypoint system for AI agents, such as guards patrolling around a series of points. A linked list can store the waypoints and allow the AI agents to traverse them in a loop or reverse direction.

**Queue is next.** Queue<T> is a first-in, first-out (FIFO) collection of objects that can be accessed by index and supports adding and removing items.

Different from a list, a queue only allows you to add items to the end of the queue and remove items from the beginning of the queue.

**Stack is next.** Stack<T> is a last-in, first-out (LIFO) collection of objects that can be accessed by index and supports adding and removing items.

Difference from the list is that you can only add and remove items from the top of the stack.

The removed element is deleted in Queue va Stack. This improves performance as it does not allow unnecessary elements to remain in the array.

**HashSet is next.** HashSet<T> is a generic collection that stores unique elements and provides methods to manipulate them. Similar to List<T> but unique elements.

As you can see here, when you try to add it again, you will not get an error at compile time or runtime, it will just be ignored and not added again. ( Line 82)

Now it's time to look at the Non-Generic collections. (Line 85)

The only structural difference here is that the Generic feature has been removed.

Arraylist is a non-generic version of the List<T> class.

As you can see, both int, string and vector types are added in a single array. Because there is no type limitation. (Line 92-97)

Hashtable is a non-generic collection of key-value pairs. That is, it is the Dictionary without type limitations. ( Line 102-104 )

As you can see, Stack and Queue can be used both generic and non-generic.(Line – 108 – 116)

But the recommended uses for type safety are always Generic ones, so let's take care to use Generic collections in our projects.

Additionally, we can create our own custom collections.

For example, we can write a special collection that takes only int values and accepts only even numbers within them. (Line – 129)

Here, for example, since the Add function is in List, we can hide it with the new keyword and use the method we have written. (Line – 136)

The base keyword allows us to access methods or properties of the inherited class. (Line – 141)

The update method below is a method specific to the custom collection I created, which is not included in List. (Line – 150)

As you can see here, it does not give an error when adding 2 and 4, but if we want to update the number 2 with 3, it will throw us the error we define below and will not perform the update. (Line – 119 - 122)

**Extensions.cs**

Extension is next. Extensions are a structure that allows us to add new methods without modifying an existing class or struct. By using these methods, we get rid of the extra workload and we don't have to write the same code all the time. Let's explain it through an example, I think you will understand it better.

Here we see a string extension static class. To write an extension, the class and our method must be static. (Line – 21)

Then we see our method. The difference here is that the parameter field starts with this keyword. This keyword ensures that this method is used as an extension for the type you have given next to it and that it is sent directly as a parameter.

As we saw in the test function, we can use this method directly by putting .(dot) after our string variable.(Line – 49)

So what will it do if you have a structure that you constantly repeat in your code base, you can manage it from a single place by making it an extension and you will prevent code repeat.

Now that we know about generics and collections, we can write a generic collection extension. (Line -34)

This method first groups the list by each value, then finds the maximum number of any group, and finally returns the group(s) with that number.

Because it is generic, it can do this for any type.

As we have seen in the test method, it finds the most repeated ones from a string or integer array and returns them back. (Line53-55)

Yes, we've come to the end of the extension section.

**Reflection.cs**

Reflection in C# is the process of collecting information on its features and operating on itself. The collecting information includes the properties, type, events, and methods of an object; reflection is useful in finding all types of assemblies.

Reflection allows you to inspect and manipulate classes, constructors, methods, and fields at run time. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties.

Reflection provides objects (of type Type) that describe assemblies, modules, and types. You can use the System.Reflection namespace to perform reflection. The Type class is the primary way to access the metadata by using methods and properties

Basically, when we analyze this code, we include an external dll file that is not included in our project during compile time during runtime.(Line -31)

We scan the classes in the dll and find the classes that are subclasses of Monobehevior. Then we create an instance from those classes and add the instance as a component to a game object. Then we call the start method of that class if it exists and make it run.

As you can see, we have written a code that will run completely at runtime in a dll. This is how we can use a dll in our projects that is not directly included in the application and will be included later if necessary.

Let's go back to the description field here.

As you can see here we have a generic extension. This method basically gets the filedinfo of the value sent to it from the specified enum type.

Then it takes the value in the Description attribute that we wrote in square brackets above and converts it to string and returns to us again.

That's it for the Reflection part.

**Finish**