Tiny Test

Thank you for the opportunity to revisit some of my basics. I'm going to be doing these in broad strokes and there is just not enough time to get into everything in the detail that these things warrant. I thought that this would be simple enough and then spent several days trying to figure out how to clearly communicate these concepts… I may have gone too far and I still can't guarantee that it will make any kind of sense, but here goes! Let's start with OOP.

-OOP in General (what is it?)

OOP stands for Object-Oriented Programming and is a programming paradigm of designing and structuring code around the use of 'Objects' and the principles of *Abstraction*, *Encapsulation*, *Inheritance*, and *Polymorphism*.

Objects can be any collection of data; from a single variable to a collection of attributes and functions.
In class-based OOP languages, objects are based on 'Classes' that act as blueprints for the individual object instances. This includes any methods and attributes that we wish for our object instances to have access to.

Abstraction is the practice of focusing on relevant details to the excursion of irrelevant ones.
It is a way of simplifying development and keeping functionality in focus.

Encapsulation is a way of protecting the properties of classes from unauthorised outside interference. We do this by making the fields and methods we want to protect private. Now only the class itself has access to its properties and we can now make public Getters and Setters to regulate the interaction with the outside of the class.

Example:

```
namespace TinyTest
{
      public class ExampleClass
      {
      private int id;
      public int Id
      {
            get { return id; }
      }

      private string name;
      public string Name
      {
            get { return name; }
            set { name = value; }
```

```
        }

        public ExampleClass(int id = 0, string name = "")
        {
                this.id = id;
                this.name = name;
        }
        }
}
```

Here in our ExampleClass we have the private variables ‘id’ and ‘name’, both can be accessed from outside the class using the public getters Id and Name, but we have not provided a setter for id, thus making it functionally read-only outside the class. You will only be able to set the Id on instantiation using the Constructor.

Inheritance is the practice of deriving classes from a base class. A derived class will inherit all of the base classes' fields and methods.

This is useful for code reusability, often you will need a lot of objects to have a certain behavior but have some of them be distinctly different. Now you COULD write normal classes for all types, OR make a single class broad enough to cover the types you want, but that’s messy and you’ll repeat yourself a lot in the first case or end up with a bloated and overly complicated class in the second case.

Instead we can have our classes inherit from a base class containing the fields and methods that they all share. And the distinct derived classes only need to cover their specific implementations

Example:

```
namespace TinyTest
{
        public class ExampleDerivedClass : ExampleClass
        {
        private string description;

        public string Description
        {
                get { return description; }
                set { description = value; }
        }
        public ExampleDerivedClass(int id, string name, string description
= "") : base(id, name)
        {
                this.description = description;
        }
        }
}
```

Here we have made a derived class of our previous ExampleClass, it will still have all the attributes of said class; Id and Name, but additionally it has a Description string.

Polymorphism means ‘many forms’ and is a term borrowed from biology. Polymorphic objects can be multiple things at the same time. There are two types of Polymorphism; Static and Dynamic.

Static or Compile Time Polymorphism happens when you create methods with the same name and class, but different parameters. It's also known as overloading methods.

This means that the same function call can trigger different behavior depending on its parameters.

Example:

```
namespace TinyTest
{
      public class Overloading
      {
      public string Combine(string[] texts)
      {
            StringBuilder sb = new StringBuilder();
            foreach (string t in texts)
            {
                  sb.Append(t + ", ");
            }

            return sb.ToString();
      }

      public string Combine(int[] numbers)
      {
            StringBuilder sb = new StringBuilder();
            foreach (int n in numbers)
            {
                  sb.Append(n + ", ");
            }

            return sb.ToString();
      }
      }
}
```

Here in the Overloading class we have two methods with the same name; Combine. One which accepts an array of strings and one that takes an array of ints. Outside the class we can now call the same method with either type of parameters and get the right result.

Dynamic or runtime polymorphism is the result of Inheritance or Interfaces. In a way we have already done this, as our ExampleDerivedClass is already ALSO a ExampleClass at the same time.
A function in a base class will be able to be called in all derived classes. This way we can share functionality across types of objects but we can also make specific implementations that override the base classes

Example:

```
        public virtual string Everything()
        {
                string str = "Name: " + name + ", ID; " + id.ToString();
                return str;
        }
```

In our ExampleClass we make a function that returns everything that the object contains. But since our ExampleDerivedClass has an extra Description property we can override the method.

```
        public override string Everything()
        {
                string str = base.Everything() + " Description: " +
Description;
                return str;
        }
```

Now the same function call will give different results based on the type of object we use it on.

Interfaces?

Interfaces in computer science is the boundary connection between two or more components of a system. The components can be anything from hardware, software or people using the system.

Because nothing can ever just be simple, there is ANOTHER type of Interfaces.
In C#; Interfaces are abstract templates that cannot be instantiated as objects themselves, instead they define what we might call ‘Contractual Obligations’ that every class that implements them must abide by.

An interface could for example be something like this; IInteractable:

```
namespace TinyTest
{
      public interface IInteractable
      {
      void Interact();
      }
}
```

That's it, the whole interface. With this, any class that implements IInteractable will be contractually obligated to have an Interact function. It says nothing about HOW to do so.

That might seem a bit useless on its own but it can help us inject a bit of polymorphism into classes that otherwise don't have anything in common. Now all classes that implement IInteractable can be accessed using the same function call.

-Linq/HashMap/Dictionary (how to filter by)

LINQ stands for Language-Integrated Query and is a uniform query syntax in C# and VB.NET. It is used to retrieve data from different sources and provides functionality to filter, sort and group the data based on the operators used.

Example:

```
var query = from e in exampleClasses
                        where e.Id > 5000
                        select e;
```

I've made a list of items from our old ExampleClasses and given them some random values as Id to illustrate how simple a filtering operation with LINQ can be.

HashMaps are a **Java** specific Key/Value pair collection... So I'll assume Hashtable was meant instead.

A **Dictionary** is a Generic Collection that stores key-value pairs in no particular order.

A **Hashtable** is a Non-generic Collection where the key/value pairs are arranged based on the hash code of the key.

Both methods of Key/Value collection should be simple to implement as long as Keys are kept unique and not null.

-Queue and Stack (lifo and fifo)

Queues and Stacks are both linear data structures. They differ in their approach to adding and removing items from them.

A **Queue** is FIFO (First In First Out) meaning that items can only be inserted in the rear end of the queue and removed from the front. So the item inserted first in the list, is the first item to be removed from the list.

Items are added to a Queue with an **Enqueue** operation and removed with a **Dequeue** operation.

A **Stack** is LIFO (Last In First Out) meaning that items can only be added and removed from the top of the stack. So the item inserted last, is the first item to come out.
Items are added to a Stack with a **Push** operation and removed with a **Pop** operation.

-Serialization (C# vs PHP) JSON and serialization

Serialization is the act of converting a data object into a series of bytes for ease of transmission and storage.

In PHP, this can be done using the **serialize** function to return a string containing a byte-stream representation of the value serialized.. Or so I'm told. I have never used PHP myself.

In C# if we are serializing to a JSON format its as simple as using the JsonSerializer.Serialize function in System.Text.Json

```
public void SerializeJSON(Object obj)
      {
            var options = new JsonSerializerOptions { WriteIndented =
true };
            string jsonString = JsonSerializer.Serialize(obj,options);

            Console.WriteLine(jsonString);
      }
```

If we are saving the JSON to a file or converting to a binary format then things get more complicated. **But I'm not going to do that now!** I've already made things complicated enough for myself and it's 4 in the morning!

If any of this has made sense, then I will be happy. This became a bit more of a deep dive into the basics than I intended and I STILL only scratched the surface.

I've made a little project called TinyTest to help me out with implementation and uploaded it to github:
https://github.com/NilTestudo/TinyTest

Thank you for your patience.

- Morten Holck Vesterager