Task 1:

Stadium file

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Test_Resit
{
    internal class Stadium
    {
        List<Character> _npcs = new List<Character>();

        public Stadium() { }

        public void AddCharacter(Character character)
        {
            _npcs.Add(character);
        }

        public void Attack(int damage)
        {
            if (_npcs.Count == 0)
            {
                Console.WriteLine("Not very effective...");
            } else
            {
                Console.WriteLine("Bring it on!");
                _npcs[0].GetHit(damage);
            }
        }

        public void AttackAll(int damage)
        {
            if (_npcs.Count == 0)
            {
                Console.WriteLine("Why?");
            } else
            {
                Console.WriteLine("Charge!");
                foreach (Character npc in _npcs)
                {
                    npc.GetHit(damage);
                }
            }

        }
    }
}
```

Immortal file

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace Test_Resit
{
    internal class Immortal : Character
    {

        public Immortal(string name) : base(name)
        {
        }

        public override void GetHit(int damage)
        {
            Console.WriteLine($"{Name}: Ha, Nice try.");
        }
    }
}
```

Mortal file

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Test_Resit
{
    public class Mortal : Character
    {
        private int _health = 15;

        public Mortal(string name, int health) : base(name)
        {
            _health = health;
        }

        public override void GetHit(int damage)
        {
            _health -= damage;
            if (_health <= 0)
            {
                Console.WriteLine($"{Name}: You already got me!");
            } else
            {
                Console.WriteLine($"{Name}: Ow!");
            }
        }
    }
}
```

Character file

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace Test_Resit
{
    public abstract class Character
    {
        string _name;

        public Character(string name)
        {
            _name = name;
        }

        public abstract void GetHit(int damage);

        public string Name
        {
            get { return _name; }
        }
    }
}
```

Program file

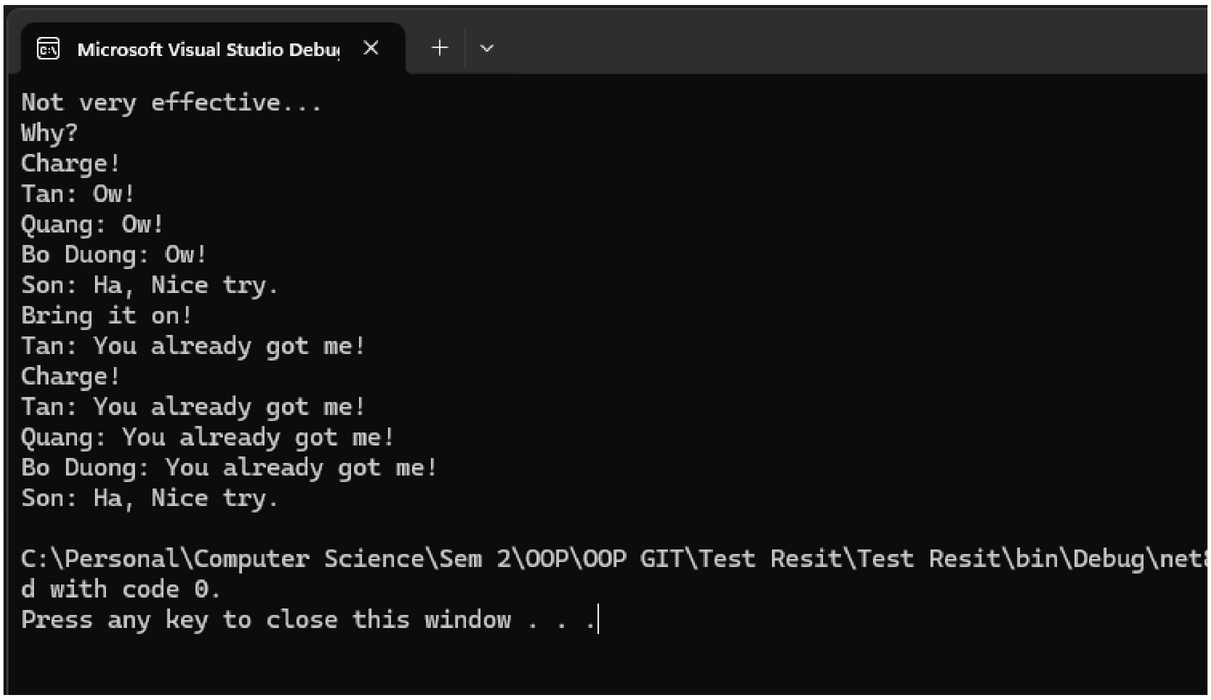
```
using System.Diagnostics;

namespace Test_Resit
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Stadium stadium = new Stadium();
            stadium.Attack(10);
            stadium.AttackAll(5);

            Mortal mortal1 = new Mortal("Tan", 15);
            Mortal mortal2 = new Mortal("Quang", 15);
            Mortal mortal3 = new Mortal("Bo Duong", 15);
            Immortal immortal = new Immortal("Son");
            stadium.AddCharacter(mortal1);
            stadium.AddCharacter(mortal2);
            stadium.AddCharacter(mortal3);
            stadium.AddCharacter(immortal);

            stadium.AttackAll(8);
            stadium.Attack(10);
            stadium.AttackAll(9);
        }
    }
}
```

Program Output



```
Not very effective...
Why?
Charge!
Tan: Ow!
Quang: Ow!
Bo Duong: Ow!
Son: Ha, Nice try.
Bring it on!
Tan: You already got me!
Charge!
Tan: You already got me!
Quang: You already got me!
Bo Duong: You already got me!
Son: Ha, Nice try.

C:\Personal\Computer Science\Sem 2\OOP\OOP GIT\Test Resit\Test Resit\bin\Debug\net
d with code 0.
Press any key to close this window . . .
```

Task 2

1. Encapsulation

- The act of placing data and the methods that use that data into a single class is known as encapsulation. Subsequently, we can determine which piece of information from this class is concealed or available to public. This promotes flexibility and scalability, which are essential factors to consider while developing large applications and optimizes the quantity of information that is exposed in other classes and files.

- Example: In the clock program we have created in task 3.1P, we have the Counter class which is initialized with a name and integer. We later created methods which enable the integer to increment and reset.

➔ This class later can be initialized in other files and reused countless times to create the clock. However, the core functionalities of the class will not be displayed, only those which are defined as 'public' – The functions are called and used. This makes the program more optimized and modular.

- In this task in particular, each Character class like Immortal or Mortal is essentially an example of encapsulation. Instead of laying all information bare in the program.cs file, we conceal them in individual files and then create objects of them in the program file.

2. Inheritance

- Inheritance is an inherent feature of OOP classes in which they can inherit others' functionality and extend or reform them. This practice promotes high reusability and a hierarchal structure in which classes are based off previous iterations. I love OOP because of this. This makes it easier to track problems within the code because to pinpoint issues present in child classes, we can always trace back to its parent classes. It also makes it easier to improve on the functionalities of a class or completely revamp it for multiple purposes.

- Example: A perfect example of Inheritance is in The Shape Drawer we have done. We made a base class called ‘Shape’ which later other class shapes (Rectangle, Circle, Line) are based on. Within this class shapes are basic variables and methods that every shape has (color, coordinates, and Draw, Save, Load method). These variables and methods can later be reused within the childrens’ classes.

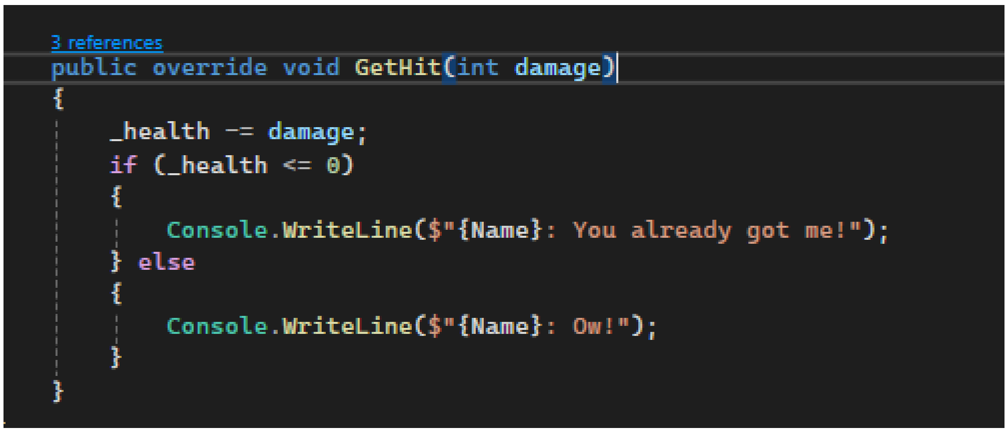
- In this task in particular, inheritance is showcased when the Immortal and Mortal class inherits from the Character class which gives them a base constructor and abstract functions and properties of their parent class (GetHit and Name)

3. Polymorphism:

- In my personal view, I think polymorphism is based on Inheritance in a way. It still enables classes to inherit a parent class but allow that class to take on various different forms depending on the context that it is reused in. In other words, while Inheritance relationship is “is-a”, the relationship that Polymorphism is more of “act-like”: classes can perform different actions while still being categorized as the same type.

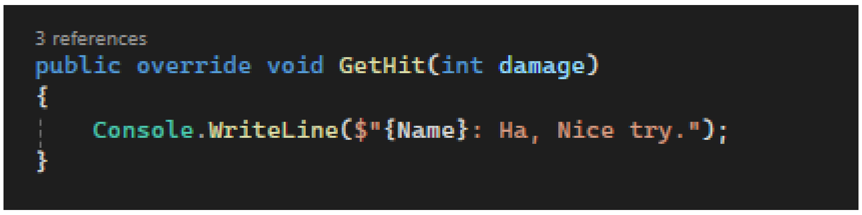
- Same example as Inheritance but we’re going to talk about the Draw method. While in Shape class it’s declared as Abstract and thus has no code. In Rectangle class, the Draw class is overridden to draw a rectangle using its own width and height variable. Similarly, in the Circle class, the Draw class is reused but to draw a circle with its own radius variable. As you can see, each shape class utilize the Draw method in its own way. When we call them in program file: MyRectangle.Draw() will perform differently to MyCircle.Draw(). This is Polymorphism.

- In this task in particular, the abstract function of GetHit is an example of polymorphism. Although both Immortal and Mortal class have the GetHit functions, their GetHit functions works differently to each other.



```
3 references
public override void GetHit(int damage)
{
    _health -= damage;
    if (_health <= 0)
    {
        Console.WriteLine($"{Name}: You already got me!");
    } else
    {
        Console.WriteLine($"{Name}: Ow!");
    }
}
```

Figure 1: Mortal's GetHit Function



```
3 references
public override void GetHit(int damage)
{
    Console.WriteLine($"{Name}: Ha, Nice try.");
}
```

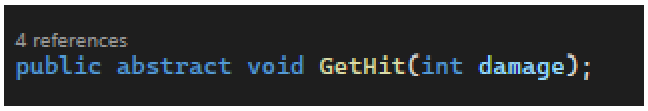
Figure 2: Immortal's GetHit Function

4. Abstraction:

- Abstraction describes the state in which a method is declared without any implementation. The method details will be specified later by the other methods which inherit it. Therefore, abstraction allows programmers to focus on what does an object do, instead of how it do it. I think abstraction simplifies the program a lot and facilitates scalability by reducing code complexity.

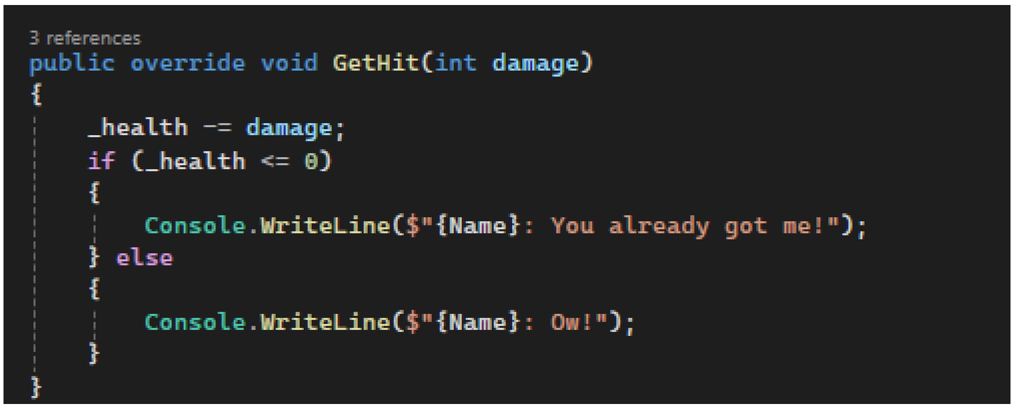
- The example used for Polymorphism describes Abstraction pretty well. The original Draw method in 'Shape' class has no actual implementation but rather acts as a barebone structure which other subclass shape can import their own implementation into.

- In this task, the entire Character class is an abstract class which means it cannot be initialized in main as well as use its function directly. Only children classes such as Mortal/Immortal class can give meaning to its abstract functions (GetHit)



```
4 references
public abstract void GetHit(int damage);
```

Figure 3: Character's Abstract Function



```
3 references
public override void GetHit(int damage)
{
    _health -= damage;
    if (_health <= 0)
    {
        Console.WriteLine($"{Name}: You already got me!");
    } else
    {
        Console.WriteLine($"{Name}: Ow!");
    }
}
```

Figure 4: Mortal's Overridden Function