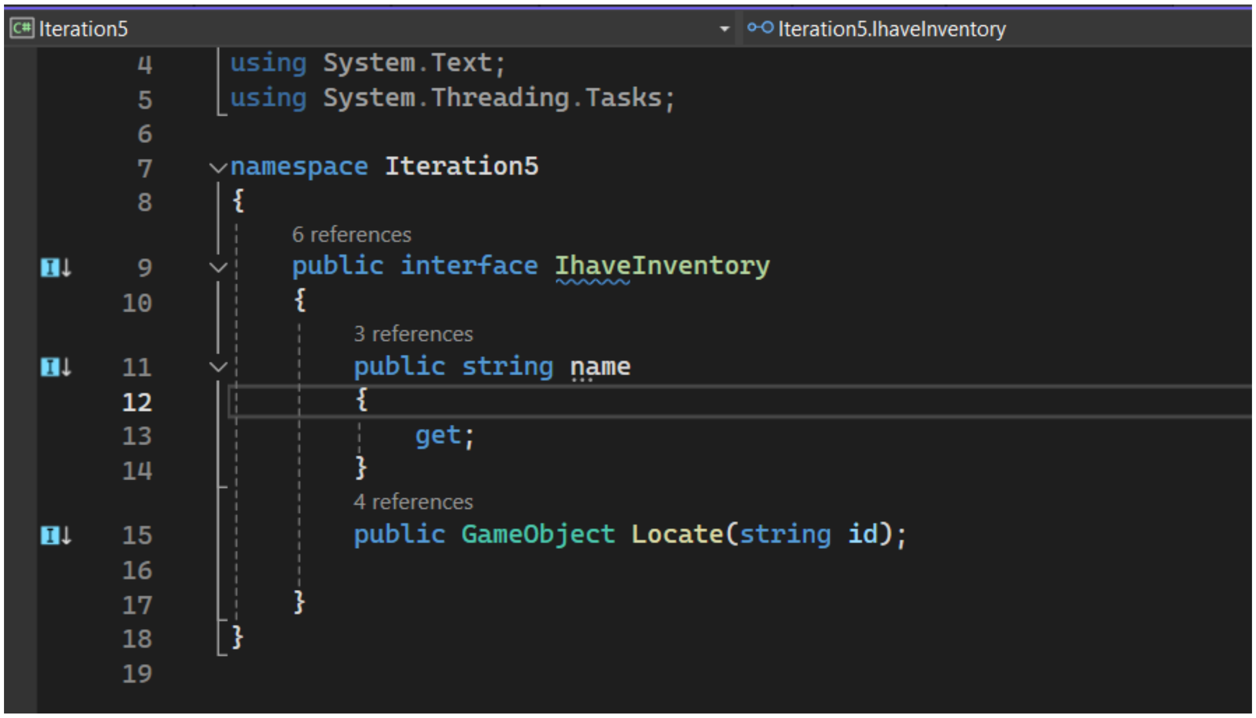
Object Oriented Programming
Pass Task 6.2: Key Object-Oriented Concepts

Concept

1. Abstraction: Means that focusing on the main important points while hiding the unwanted parts of the code. This will be easy for the user to access and get what they wanted to do immediately. When it comes to the real-world problem, when we get a phone we can do many things done with it like we can call someone, we can capture the golden moments in our lives and also we can either play games on it or browse the internet and can watch what we want. But while we are doing it, we cannot see the process of how those are being done in there. So, we get this as the real-world example.

We can get an idea of the functionality of it from the example below.

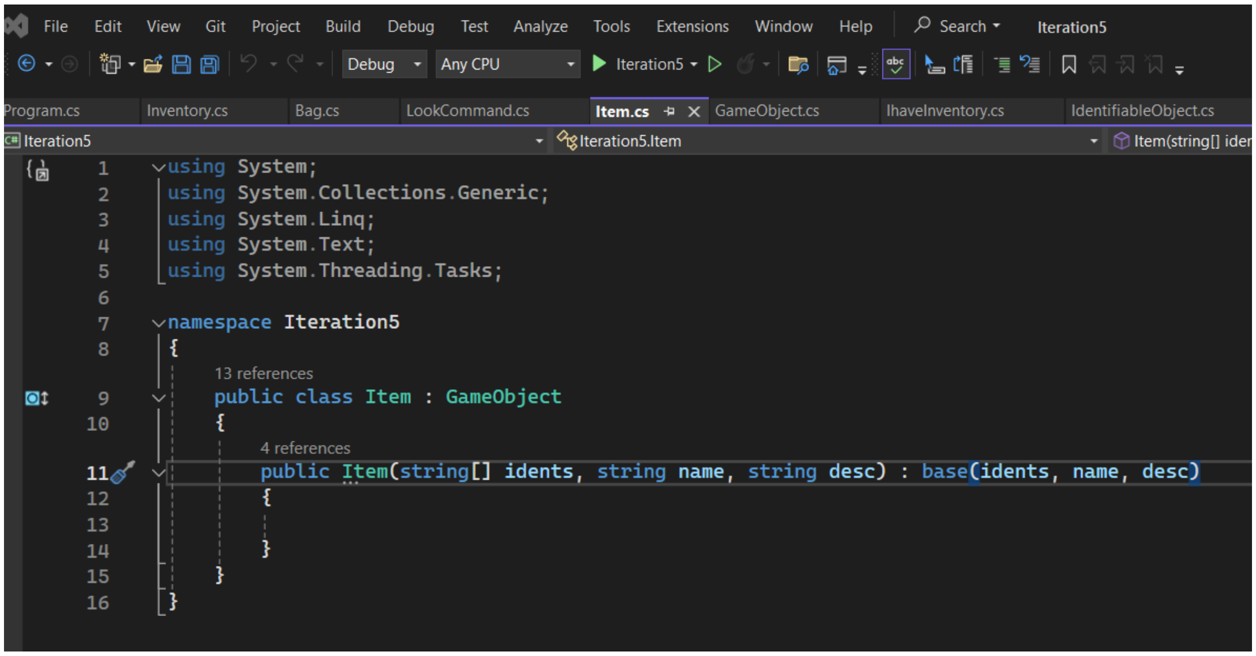


```
using System.Text;
using System.Threading.Tasks;

namespace Iteration5
{
    6 references
    public interface IhaveInventory
    {
        3 references
        public string name
        {
            get;
        }
        4 references
        public GameObject Locate(string id);

    }
}
```

This is the "IhaveInventory" class that we have implemented in Iteration 5. This interface follows the concept of abstraction, and it provides a basic structure for objects with an inventory. It defines that any class using this interface will have a Name property and a Locate method, but it doesn't specify how items are stored or managed. This ensures that classes like Bag can implement their own versions of Name and Locate, offering flexibility and reusability. Essentially, it abstracts the concept of having an inventory, allowing different classes to implement the same basic functionalities in their own ways.

2. Encapsulation: This means the idea of the bundling data and methods that work on that data within one unit. So, through this we can wrap up data and member function(method) together into a one single unit. It binds the data in a single unit along with methods or functionalities. It makes the code more modular, and it supports the data hiding ideology where the internal state is kept hidden, and interaction occurs through public methods.

We can get an idea of the functionality of it from the example below.



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

namespace Iteration5
{
    13 references
    public class Item : GameObject
    {
        4 references
        public Item(string[] idents, string name, string desc) : base(idents, name, desc)
        {

        }
    }
}
```

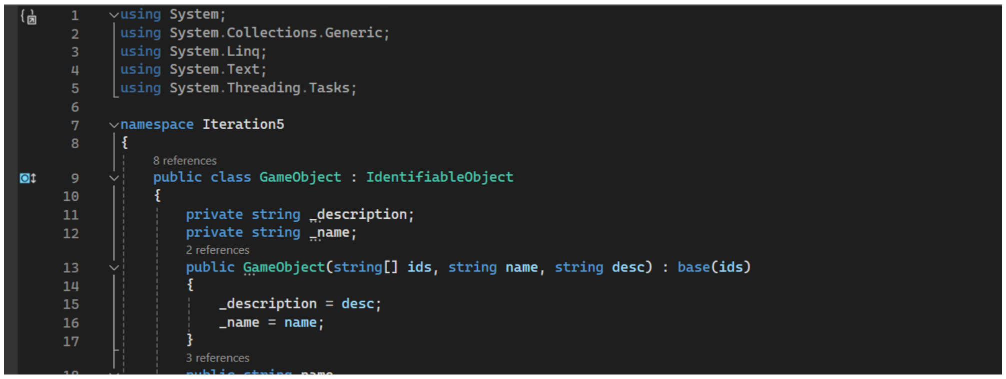
In here,

This principle has two main features: Data Hiding and Access through Public Methods.

In Data Hiding, the Item class extends to the Game Object class, which has private attributes (name and description). These attributes cannot be accessed directly from outside the Game Object class, preventing the name of an item from being changed directly. To interact with these private fields, Access through Public Methods is used. The Game Object class provides public methods or properties, such as a read-only property for accessing the _name attribute indirectly.

3. Inheritance: Inheritance means the mechanism of acquiring the properties and the behaviors from the parent class to child class. For example, a child inherits the traits of their parents.

We can get an idea of the functionality of it from the example below.



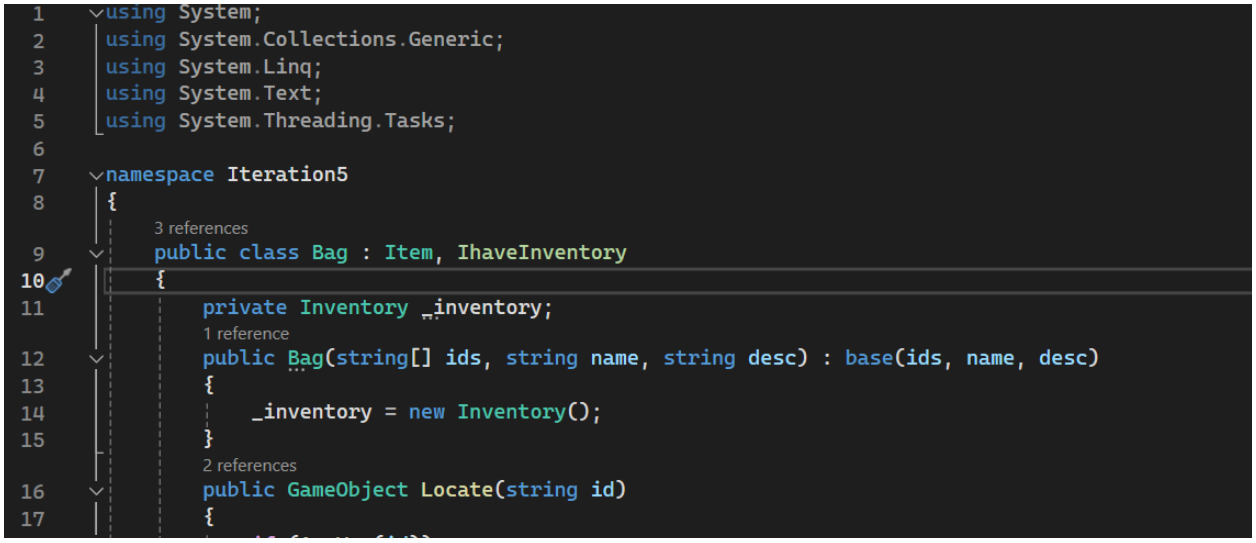
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Iteration5
{
    8 references
    public class GameObject : IdentifiableObject
    {
        private string _description;
        private string _name;
        2 references
        public GameObject(string[] ids, string name, string desc) : base(ids)
        {
            _description = desc;
            _name = name;
        }
        3 references
```

In this scenario, the behavior of the child class (Item) is to inherit properties and methods from its parent class (Game Object). Specifically, the child class automatically gains access to the read-only ` Name ` property and methods like ` Short Description ` from the parent class. This inheritance allows the child class to utilize the functionalities defined in the parent class without explicitly redefining them.

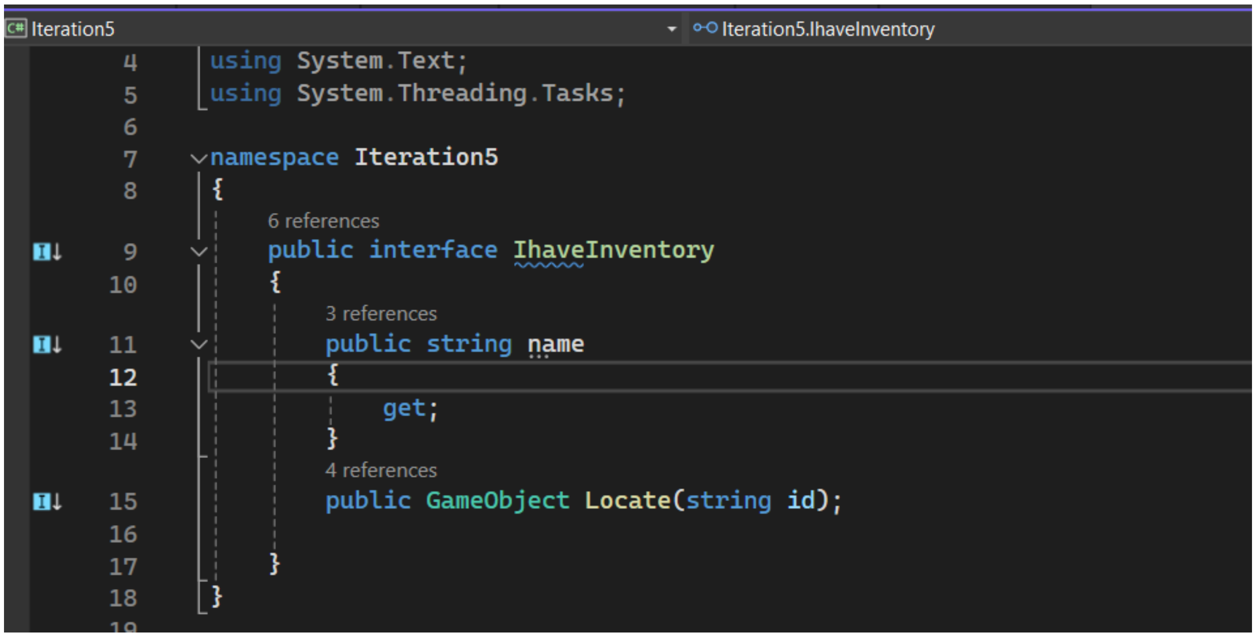
4. Polymorphism: Polymorphism means the ability to take on many forms. So, it allows the diverse types to be treated as objects of common class. So, if we get an example if we get a girl, she can be a mother, student, and a nurse at the same time.

In here there are two types of Polymorphism.

- Compile-time Polymorphism (Static): attained through operator and method overloading.
- Dynamic Polymorphism: or also known as run-time polymorphism, attained through method overriding (which is basically a particular implementation of a method which has been defined in the superclass).



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

namespace Iteration5
{
    3 references
    public class Bag : Item, IhaveInventory
    {
        private Inventory _inventory;
        1 reference
        public Bag(string[] ids, string name, string desc) : base(ids, name, desc)
        {
            _inventory = new Inventory();
        }
        2 references
        public GameObject Locate(string id)
        {
```

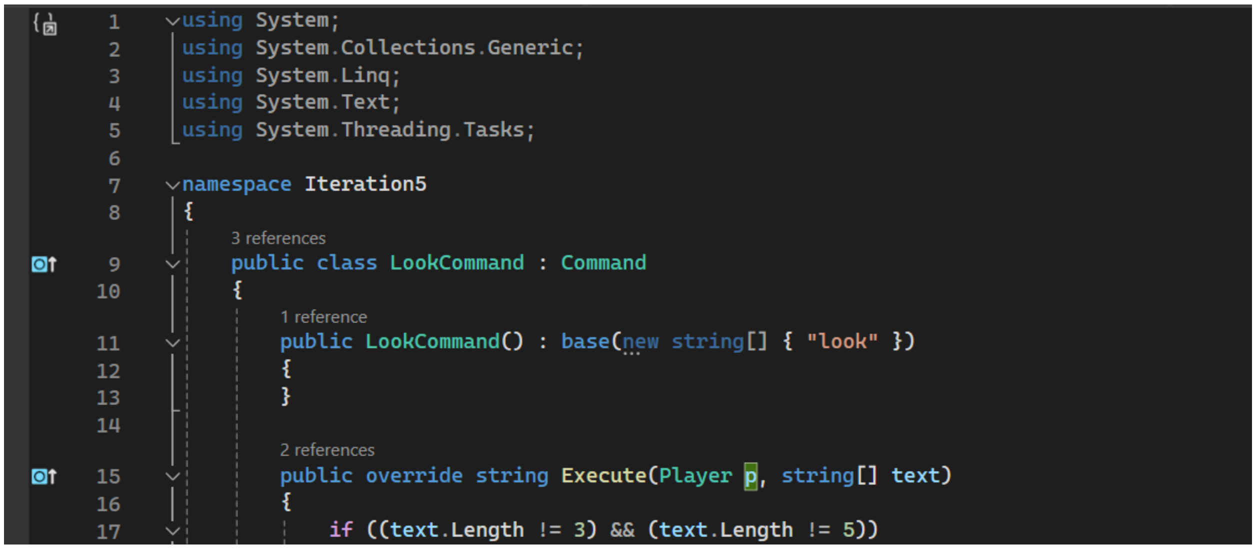


```
using System.Text;
using System.Threading.Tasks;

namespace Iteration5
{
    6 references
    public interface IhaveInventory
    {
        3 references
        public string name
        {
            get;
        }
        4 references
        public GameObject Locate(string id);

    }
}
```

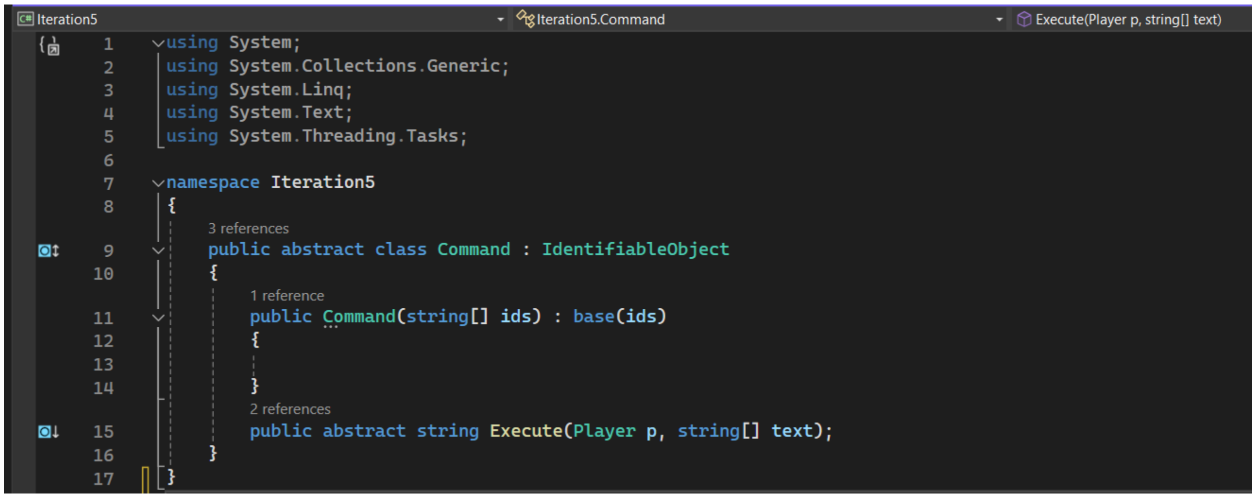
in here the “IhaveInventory” interface serves as a blueprint, specifying the structure that any class implementing it must adhere to. When a class implements this interface, it is obligated to provide concrete implementations for all the members declared by the interface. This ensures consistency and interoperability among different classes that use the interface.

We can see it in another scenario in our code in between the “Command Class” and its subclass, “Look Command” Class. In the Command class, there's an abstract method called Execute. This method sets a pattern for what actions any command should perform.

The Look Command class extends from Command, inheriting its structure including the Execute method. This demonstrates polymorphism in action because a Look Command object can be treated as a Command object. This means that any function designed to handle Command objects can seamlessly work with a Look Command object too, leveraging the Execute method to carry out specific actions related to looking within the application's context.



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

namespace Iteration5
{
    3 references
    public class LookCommand : Command
    {
        1 reference
        public LookCommand() : base(new string[] { "look" })
        {
        }

        2 references
        public override string Execute(Player p, string[] text)
        {
            if ((text.Length != 3) && (text.Length != 5))
```



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

namespace Iteration5
{
    3 references
    public abstract class Command : IdentifiableObject
    {
        1 reference
        public Command(string[] ids) : base(ids)
        {

        }
        2 references
        public abstract string Execute(Player p, string[] text);
    }
}
```