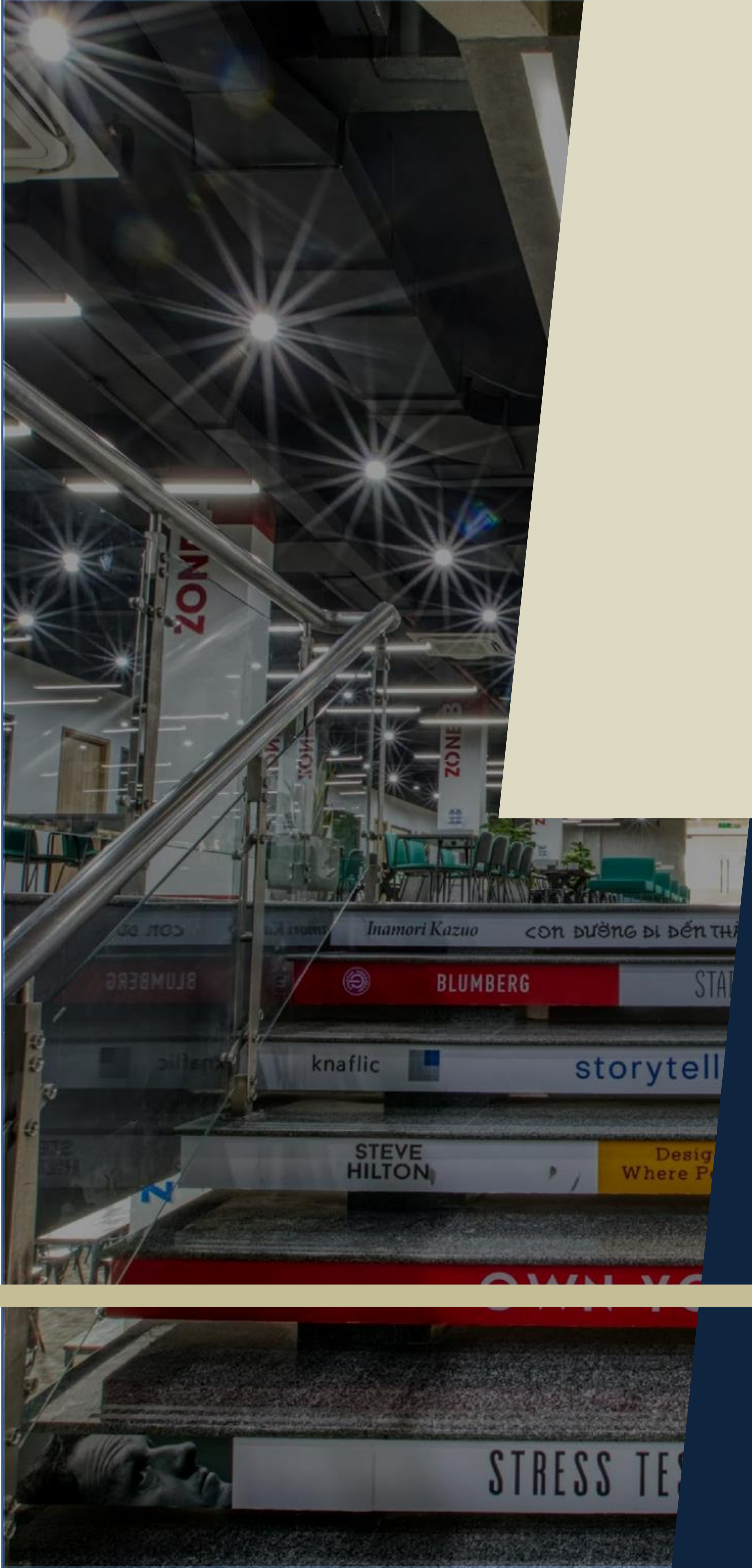


EIU

TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG

EASTERN
INTERNATIONAL
UNIVERSITY

Coding Practice

Object Oriented Programming

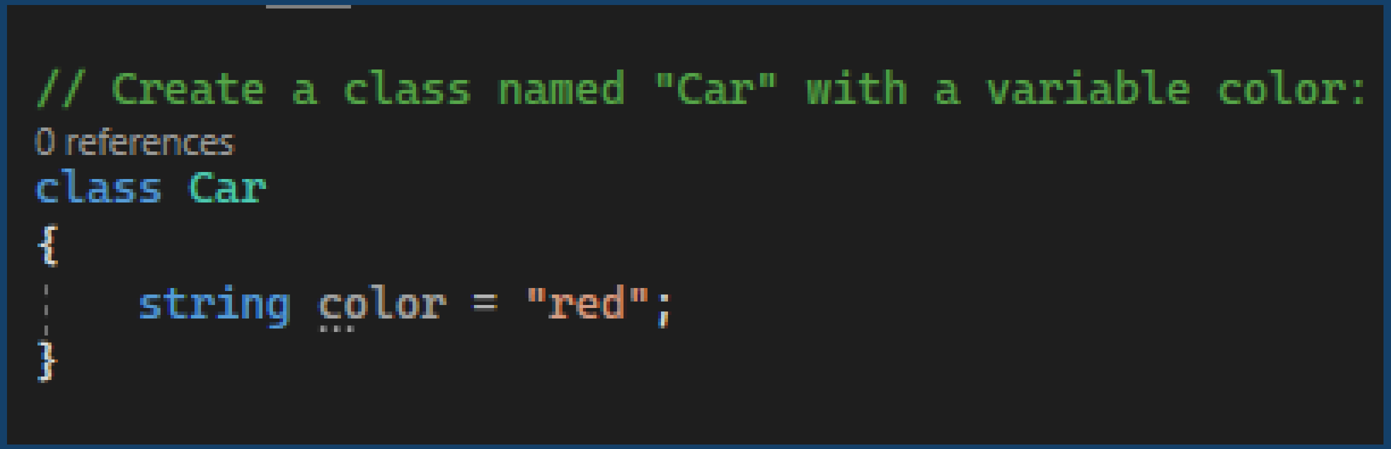
Object-oriented programming with C#

1. What is OOP?

- OOP stands for Object-Oriented Programming.
- Object-oriented programming has several advantages over procedural programming:
  - OOP is faster and easier to execute
  - OOP provides a clear structure for the programs
  - OOP makes the code easier to maintain, modify and debug
  - OOP makes it possible to create full reusable applications with less code and shorter development time
- Classes and objects are the two main aspects of object-oriented programming.
- A class is a template for objects, and an object is an instance of a class

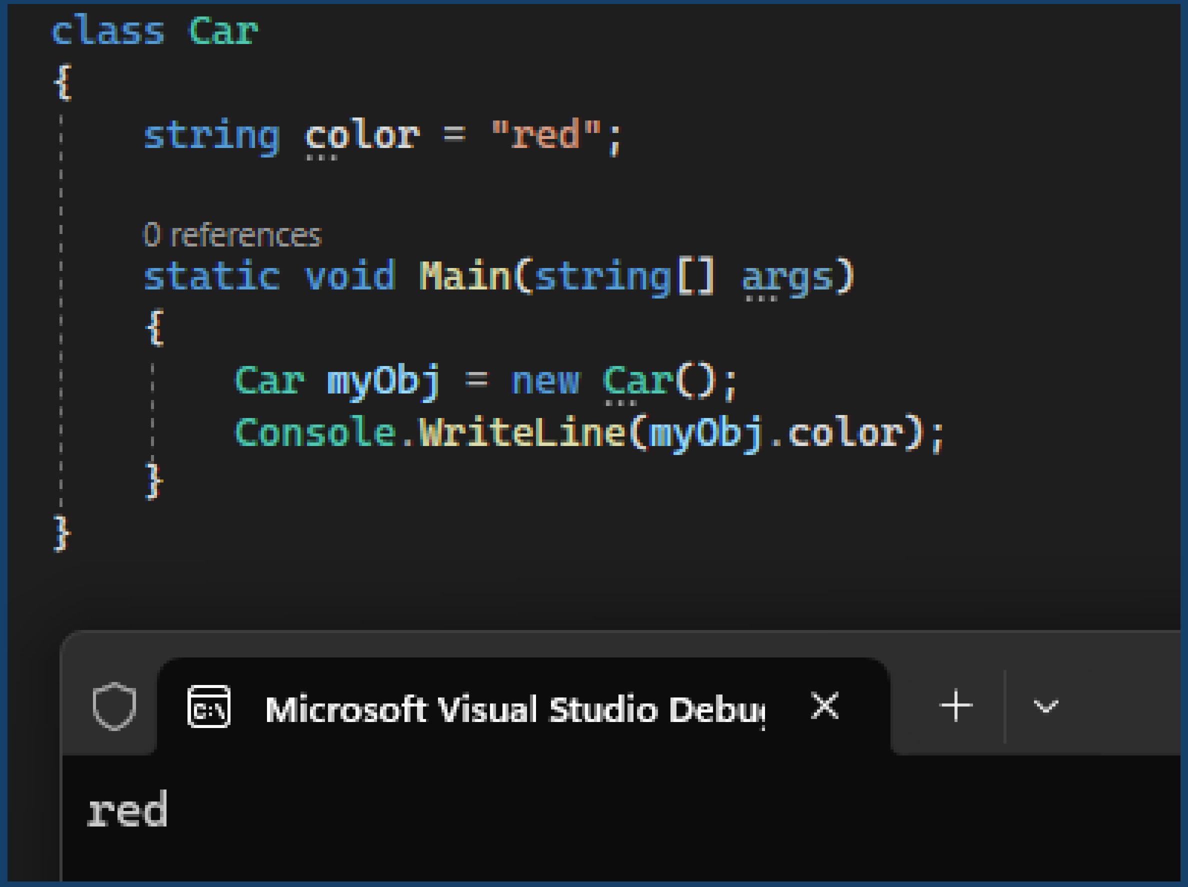
2. Classes and Objects

To create a class, use the class keyword:



```
// Create a class named "Car" with a variable color:
0 references
class Car
{
    string color = "red";
}
```

Create an Object: Create an object called "myObj" and use it to print the value of color:

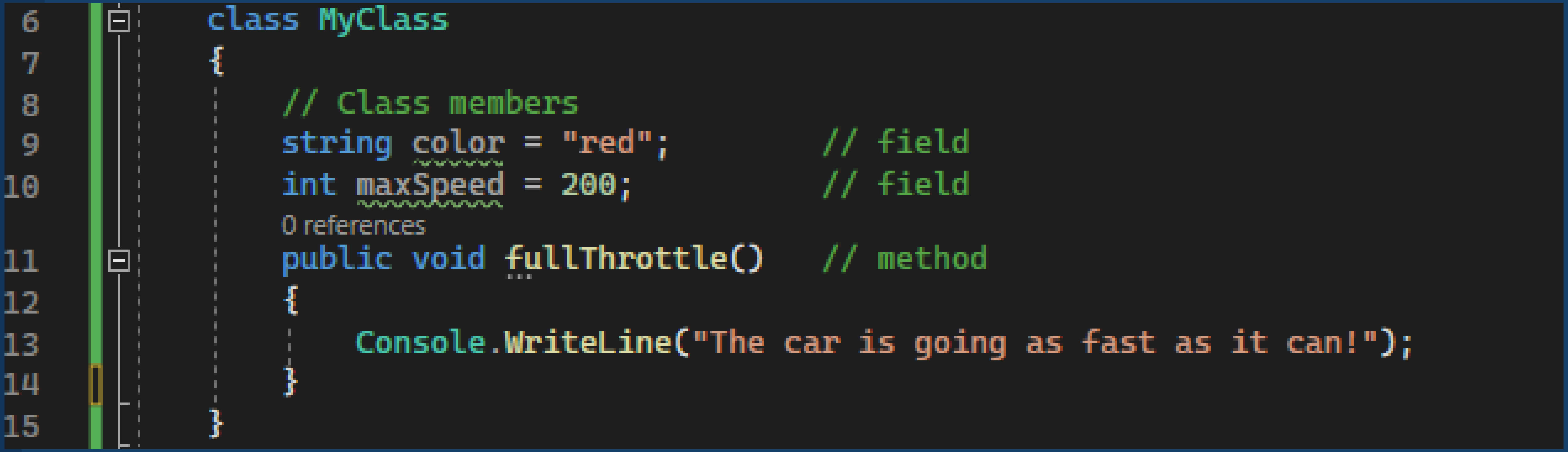


```
class Car
{
    string color = "red";

    0 references
    static void Main(string[] args)
    {
        Car myObj = new Car();
        Console.WriteLine(myObj.color);
    }
}
```

```
red
```

3. Class Members

Fields and methods inside classes are often referred to as "Class Members":



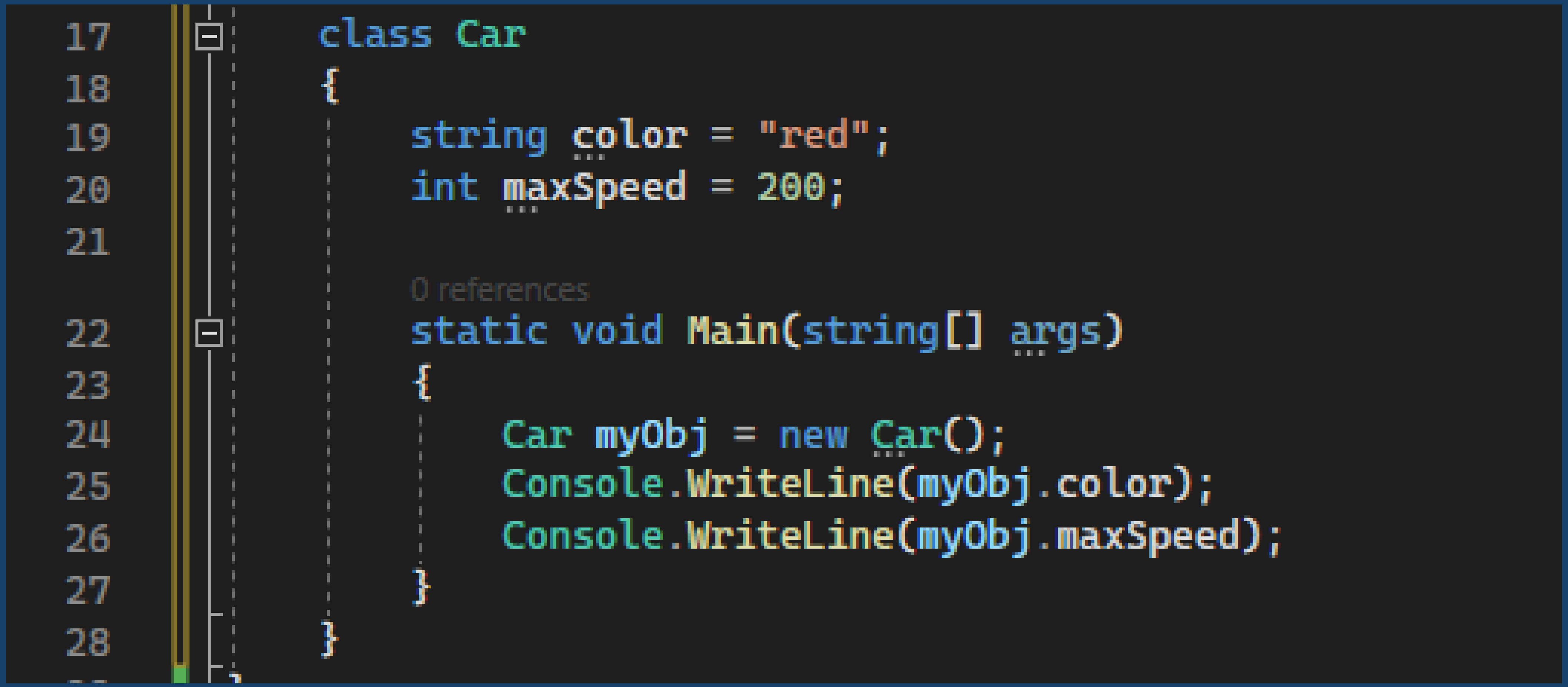
```
class MyClass
{
    // Class members
    string color = "red";         // field
    int maxSpeed = 200;           // field
    0 references
    public void fullThrottle()    // method
    {
        Console.WriteLine("The car is going as fast as it can!");
    }
}
```

3. Class Members

Fields

Variables inside a class are called fields, you can access them by creating an object of the class, and by using the dot syntax (.).

The following example will create an object of the Car class, with the name myObj. Then we print the value of the fields color and maxSpeed:



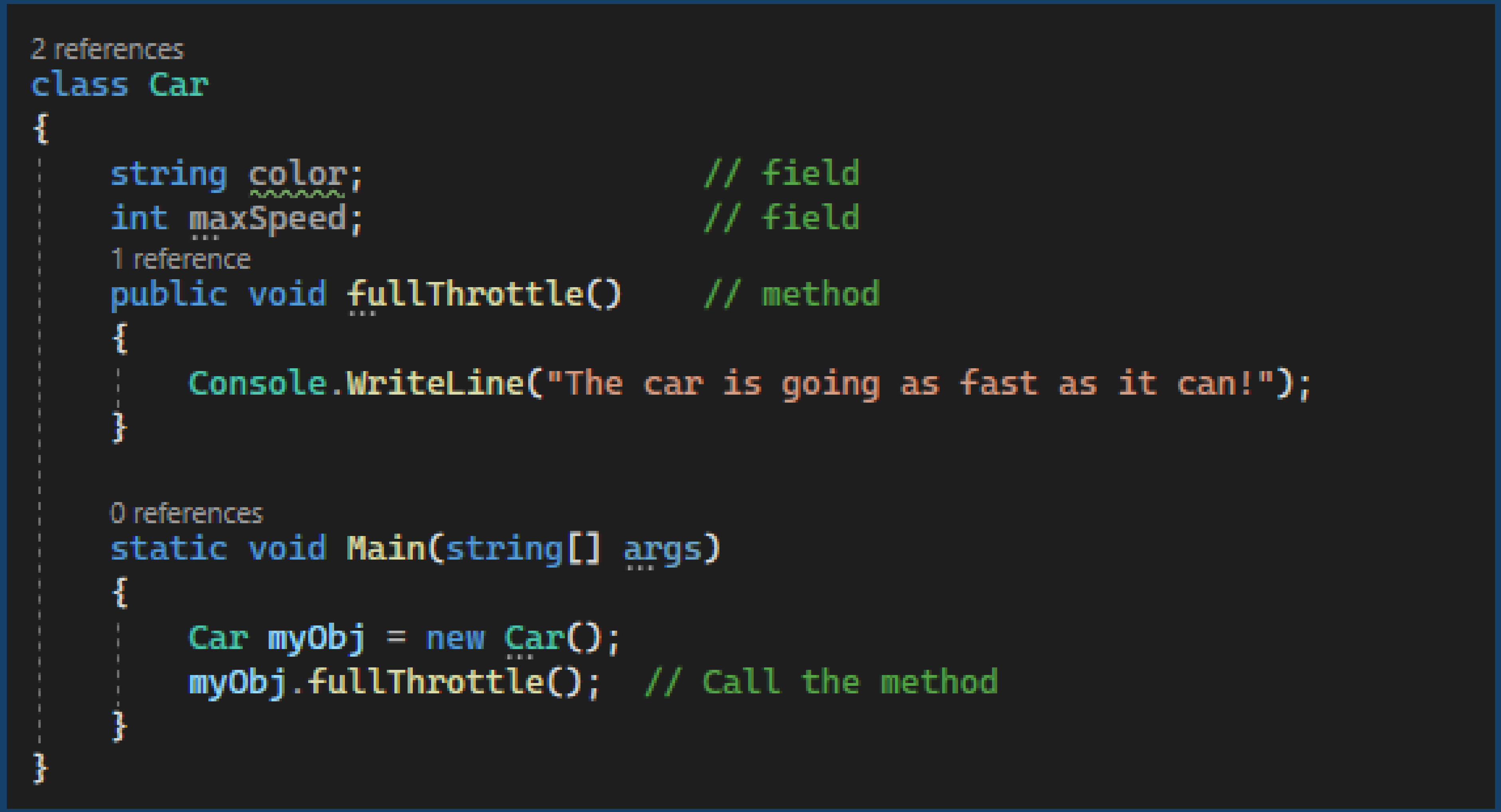
```
    class Car
    {
        string color = "red";
        int maxSpeed = 200;

        0 references
        static void Main(string[] args)
        {
            Car myObj = new Car();
            Console.WriteLine(myObj.color);
            Console.WriteLine(myObj.maxSpeed);
        }
    }
```

3. Class Members

Object Methods

Methods normally belong to a class, and they define how an object of a class behaves.

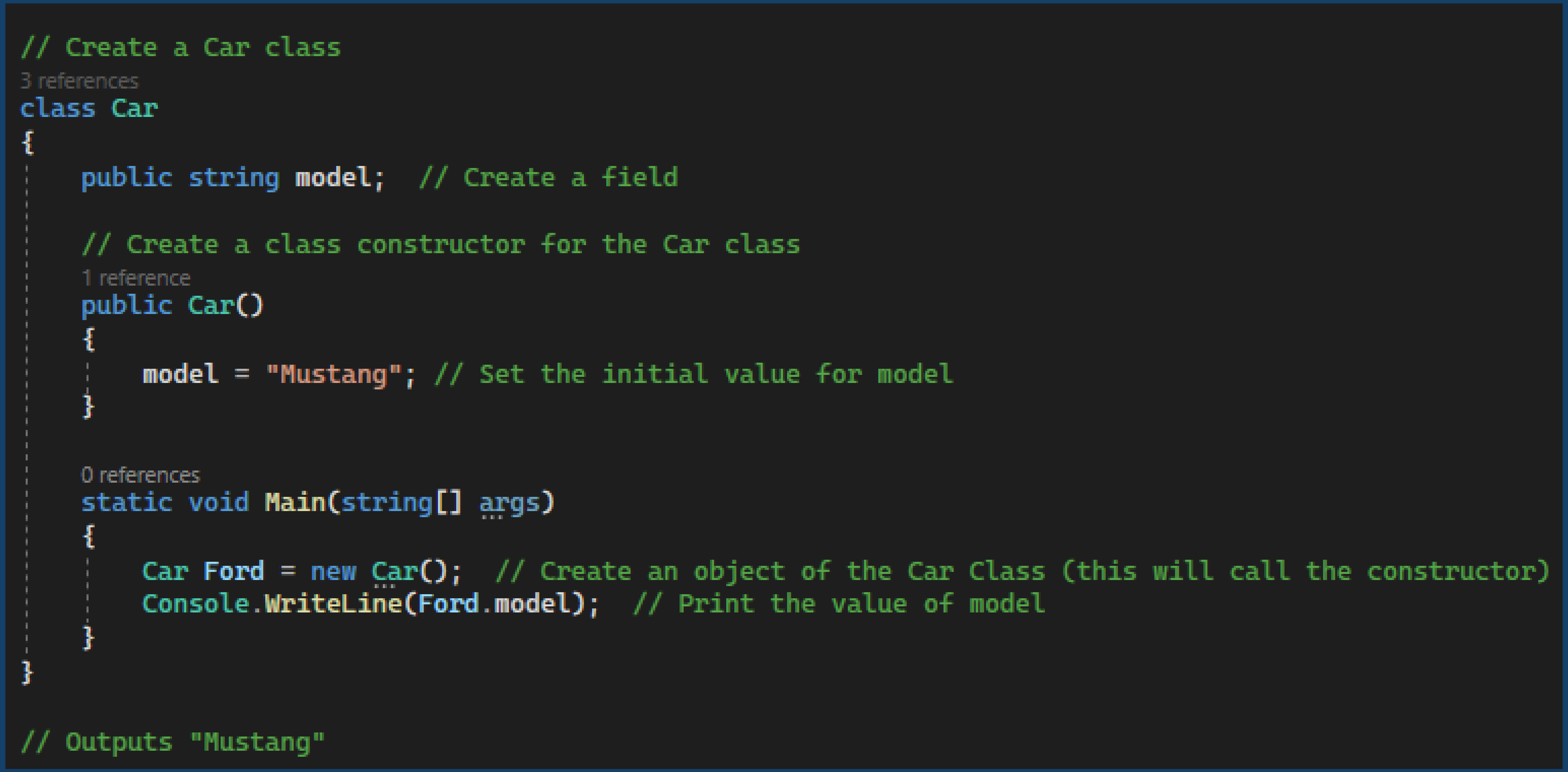
Just like with fields, you can access methods with the dot syntax. However, note that the method must be public. And remember that we use the name of the method followed by two parentheses () and a semicolon ; to call (execute) the method:



```
2 references
class Car
{
    string color;                   // field
    int maxSpeed;                   // field
    1 reference
    public void fullThrottle()      // method
    {
        Console.WriteLine("The car is going as fast as it can!");
    }

    0 references
    static void Main(string[] args)
    {
        Car myObj = new Car();
        myObj.fullThrottle();  // Call the method
    }
}
```

4. Constructors

A constructor is a special method that is used to initialize objects. The advantage of a constructor, is that it is called when an object of a class is created. It can be used to set initial values for fields:



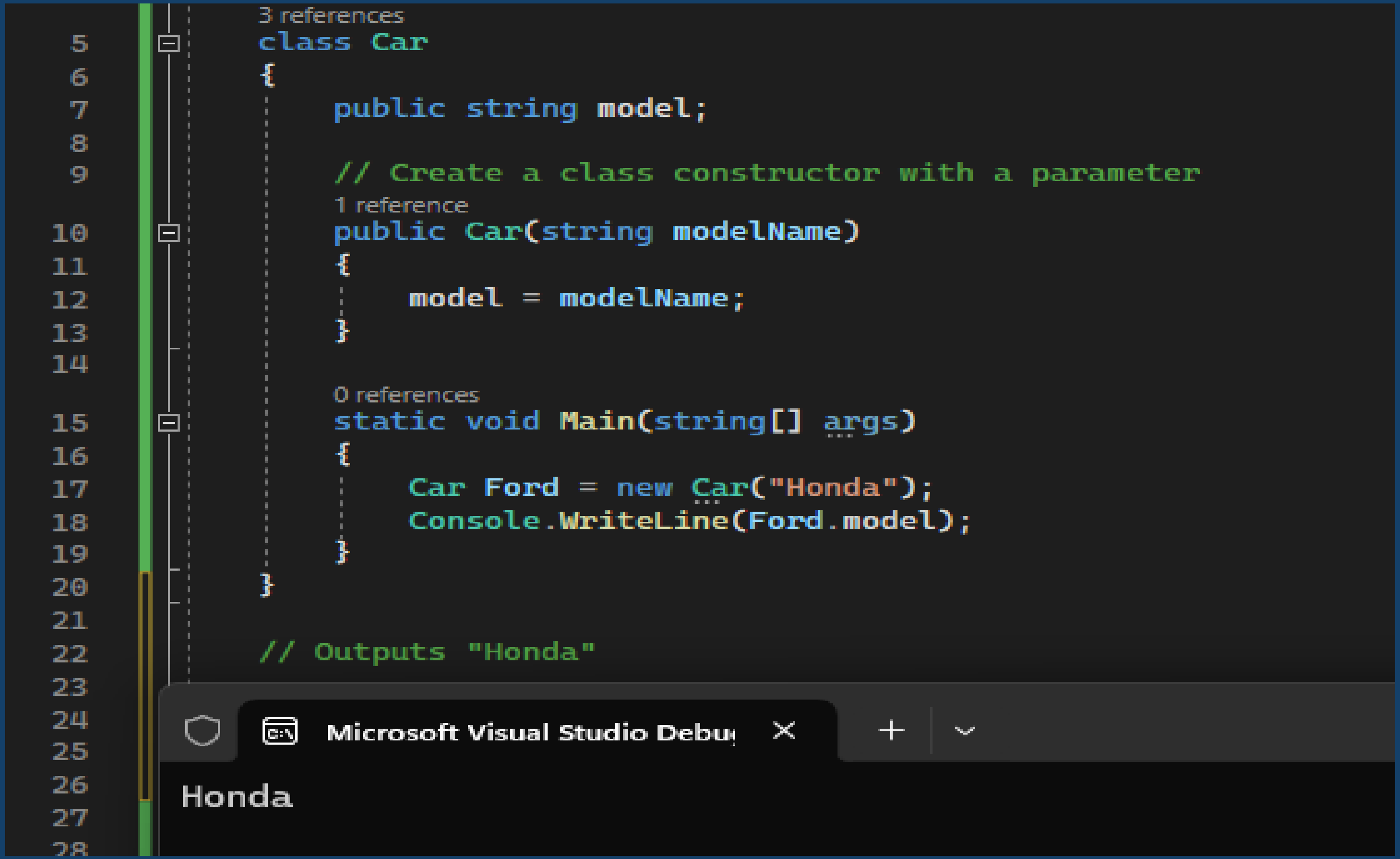
```
// Create a Car class
3 references
class Car
{
    public string model;  // Create a field

    // Create a class constructor for the Car class
    1 reference
    public Car()
    {
        model = "Mustang"; // Set the initial value for model
    }

    0 references
    static void Main(string[] args)
    {
        Car Ford = new Car();  // Create an object of the Car Class (this will call the constructor)
        Console.WriteLine(Ford.model);  // Print the value of model
    }
}

// Outputs "Mustang"
```

4. Constructors

Constructor Parameters

Constructors can also take parameters, which is used to initialize fields.

The following example adds a string modelName parameter to the constructor. Inside the constructor we set model to modelName (model=modelName). When we call the constructor, we pass a parameter to the constructor ("Mustang"), which will set the value of model to "Mustang":



```
3 references
class Car
{
    public string model;

    // Create a class constructor with a parameter
    1 reference
    public Car(string modelName)
    {
        model = modelName;
    }

    0 references
    static void Main(string[] args)
    {
        Car Ford = new Car("Honda");
        Console.WriteLine(Ford.model);
    }
}

// Outputs "Honda"
```

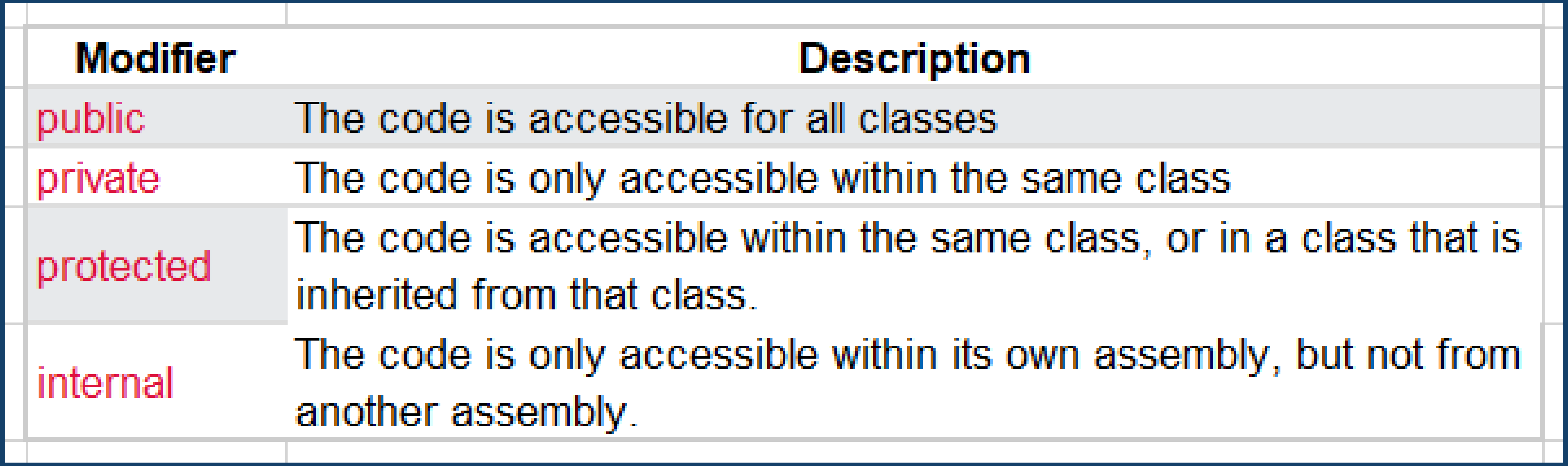
```
Honda
```

5. Access Modifiers

public string color;

The public keyword is an access modifier, which is used to set the access level/visibility for classes, fields, methods and properties.

C# has the following access modifiers:



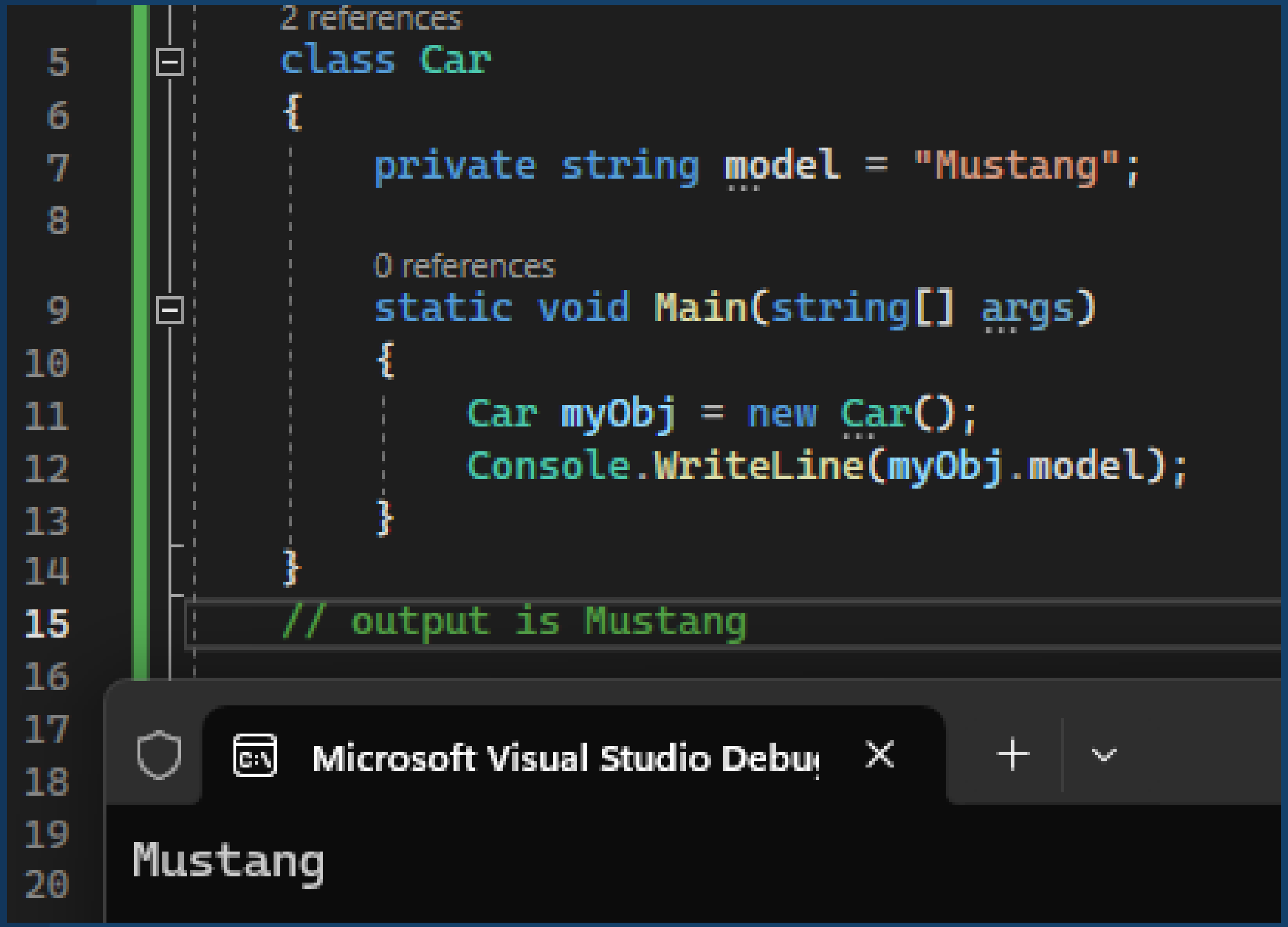
Modifier	Description
public	The code is accessible for all classes
private	The code is only accessible within the same class
protected	The code is accessible within the same class, or in a class that is inherited from that class.
internal	The code is only accessible within its own assembly, but not from another assembly.

5. Access Modifiers

Private Modifier

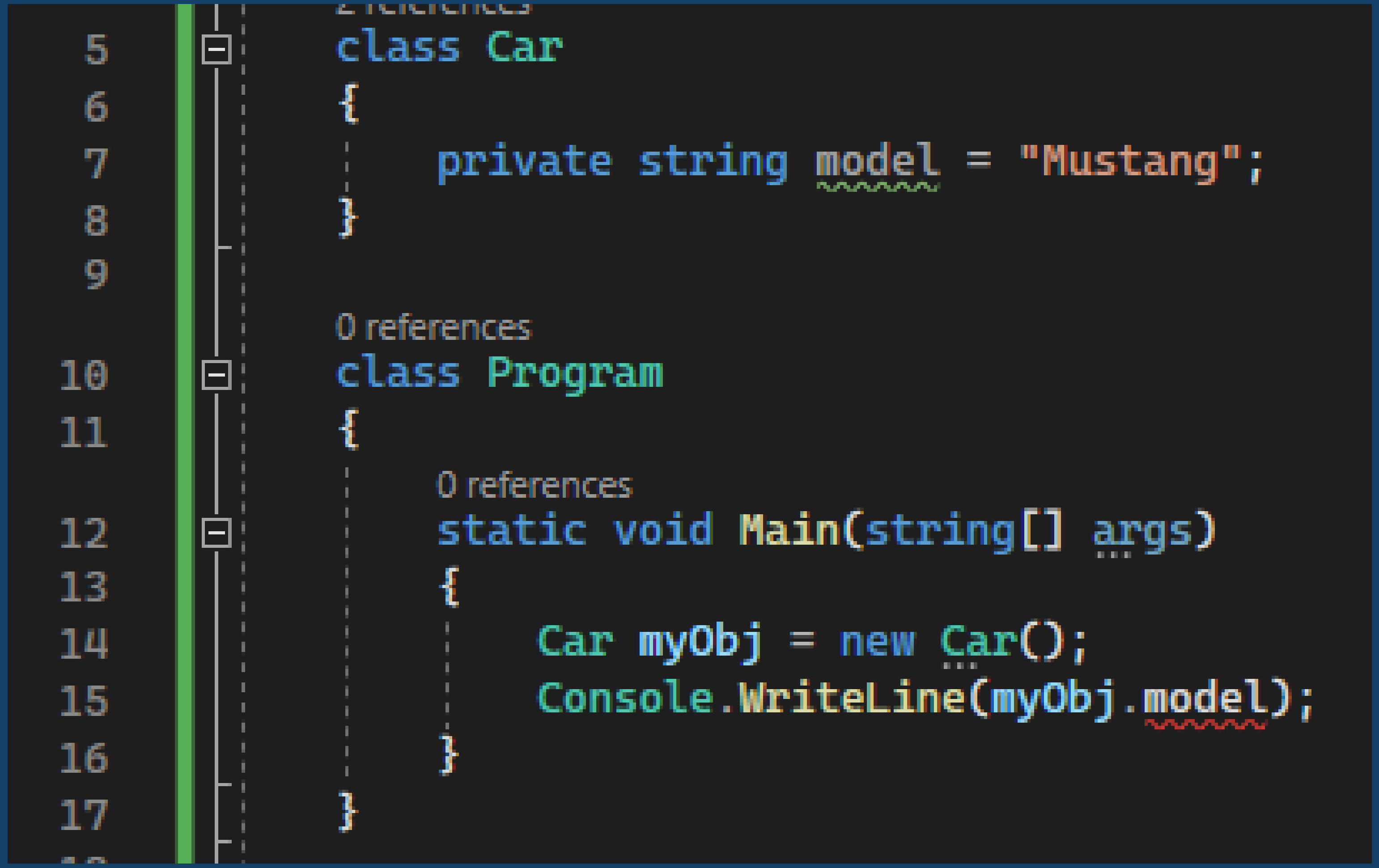
If you declare a field with a private access modifier, it can only be accessed within the same class.

If you try to access it outside the class, an error will occur.

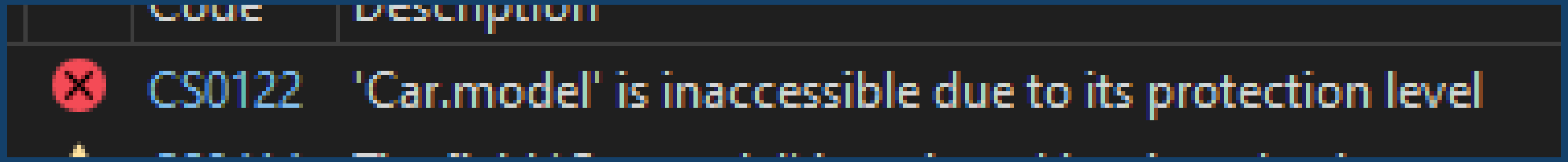


```
2 references
class Car
{
    private string model = "Mustang";

    0 references
    static void Main(string[] args)
    {
        Car myObj = new Car();
        Console.WriteLine(myObj.model);
    }
}
// output is Mustang
```

```
Mustang
```



```
2 references
class Car
{
    private string model = "Mustang";
}

0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Car myObj = new Car();
        Console.WriteLine(myObj.model);
    }
}
```



Code	Description
CS0122	'Car.model' is inaccessible due to its protection level

5. Access Modifiers

WHY ACCESS MODIFIERS?

To control the visibility of class members (the security level of each individual class and class member).

To achieve "Encapsulation" - which is the process of making sure that "sensitive" data is hidden from users. This is done by declaring fields as private.

6. Properties (Get and Set)

Properties and Encapsulation

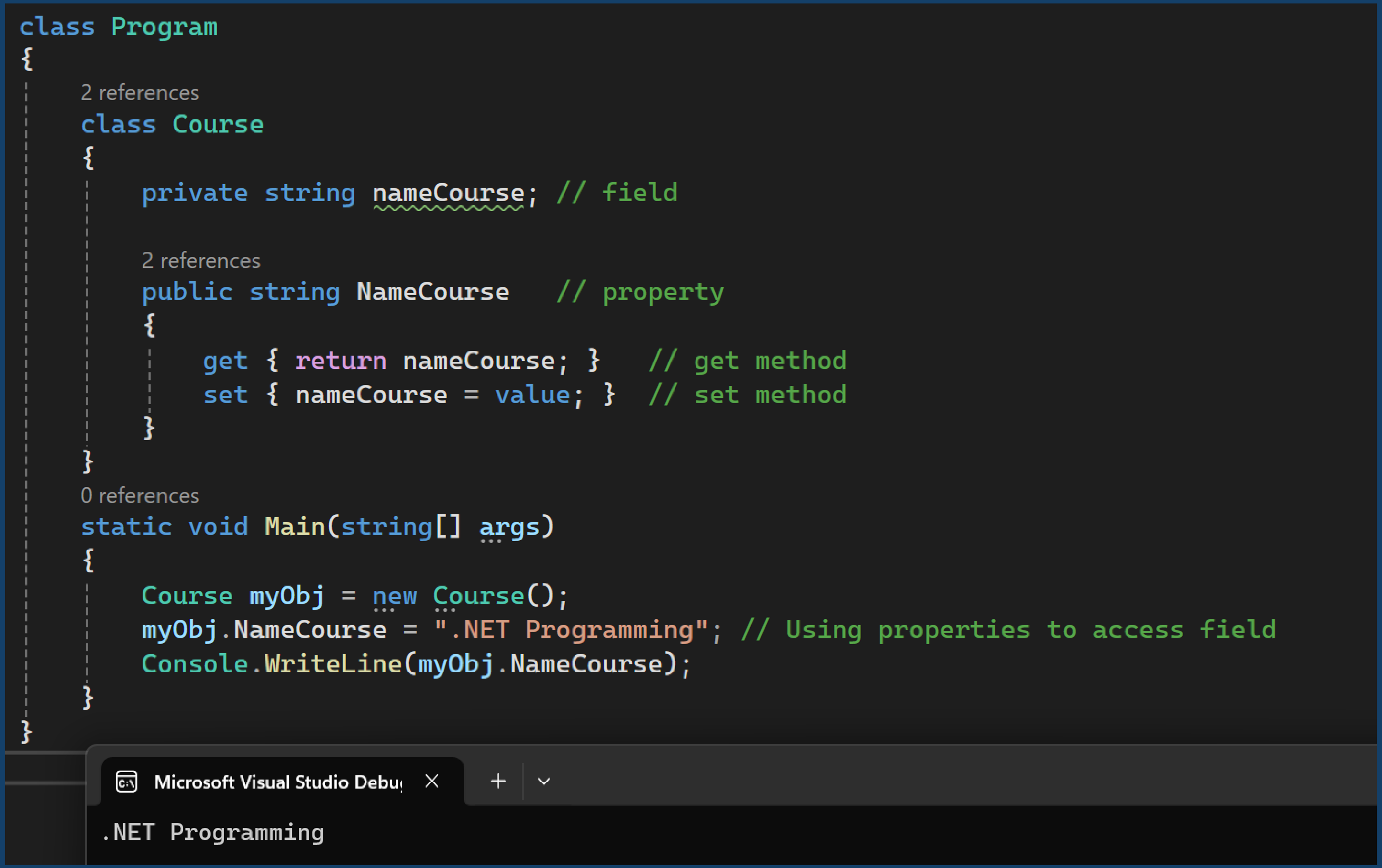
Encapsulation mean is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

- Declare fields/variables as private
- Provide public get and set methods, through properties, to access and update the value of a private field

6. Properties (Get and Set)

Properties

- Private variables can only be accessed within the same class (an outside class has no access to it)
- A property is like a combination of a variable and a method, and it has two methods: a get and a set method:



```
class Program
{
    2 references
    class Course
    {
        private string nameCourse; // field

        2 references
        public string NameCourse   // property
        {
            get { return nameCourse; }   // get method
            set { nameCourse = value; }  // set method
        }
    }
    0 references
    static void Main(string[] args)
    {
        Course myObj = new Course();
        myObj.NameCourse = ".NET Programming"; // Using properties to access field
        Console.WriteLine(myObj.NameCourse);
    }
}
```

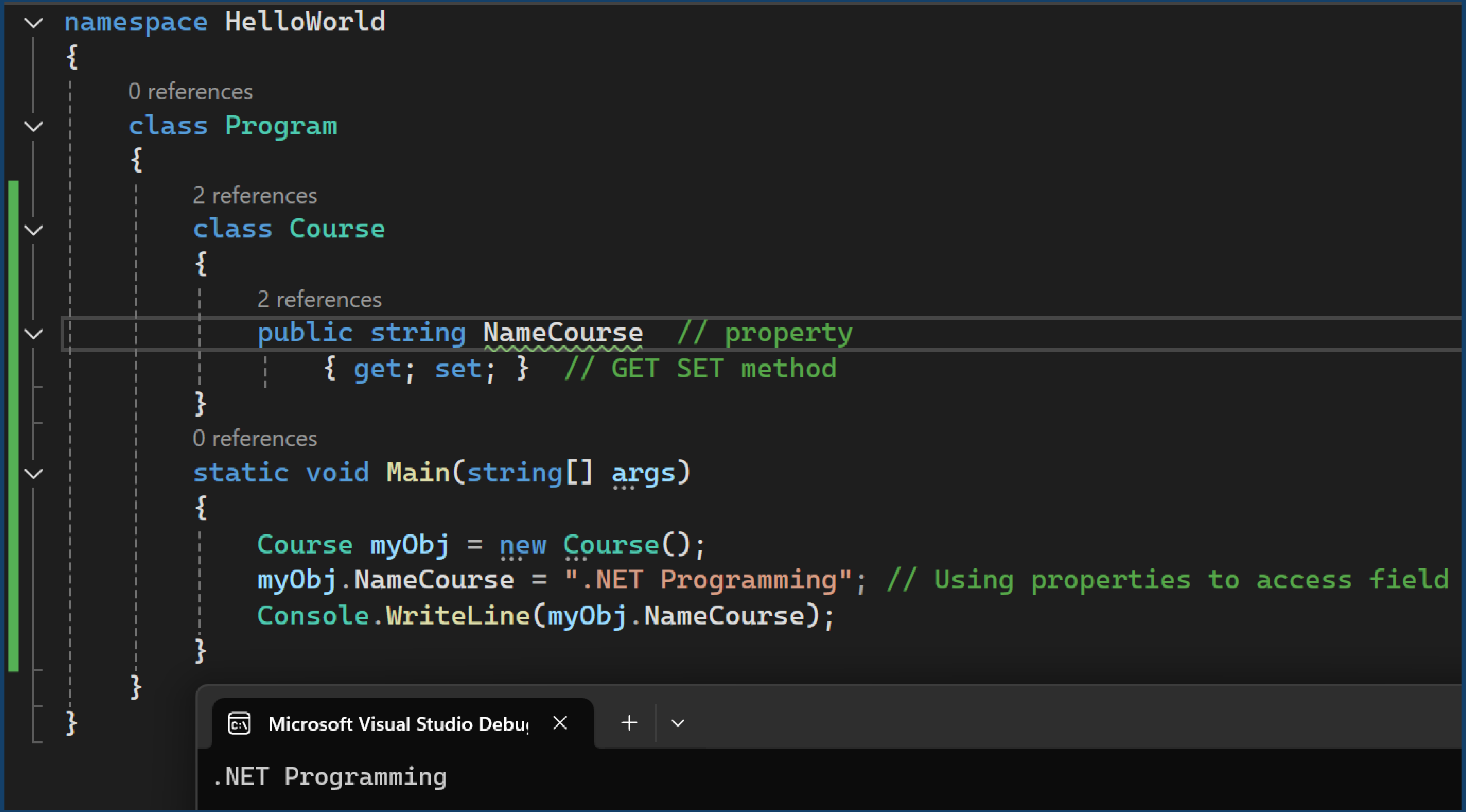
```
.NET Programming
```

6. Properties (Get and Set)

Automatic Properties (Short Hand)

C# also provides a way to use short-hand / automatic properties, where you do not have to define the field for the property, and you only have to write get; and set; inside the property.

The result is the same; the only difference is less code.



```
namespace HelloWorld
{
    0 references
    class Program
    {
        2 references
        class Course
        {
            2 references
            public string NameCourse   // property
                { get; set; }   // GET SET method
        }
        0 references
        static void Main(string[] args)
        {
            Course myObj = new Course();
            myObj.NameCourse = ".NET Programming"; // Using properties to access field
            Console.WriteLine(myObj.NameCourse);
        }
    }
}
```

```
.NET Programming
```

7. Inheritance

Inheritance (Derived and Base Class)

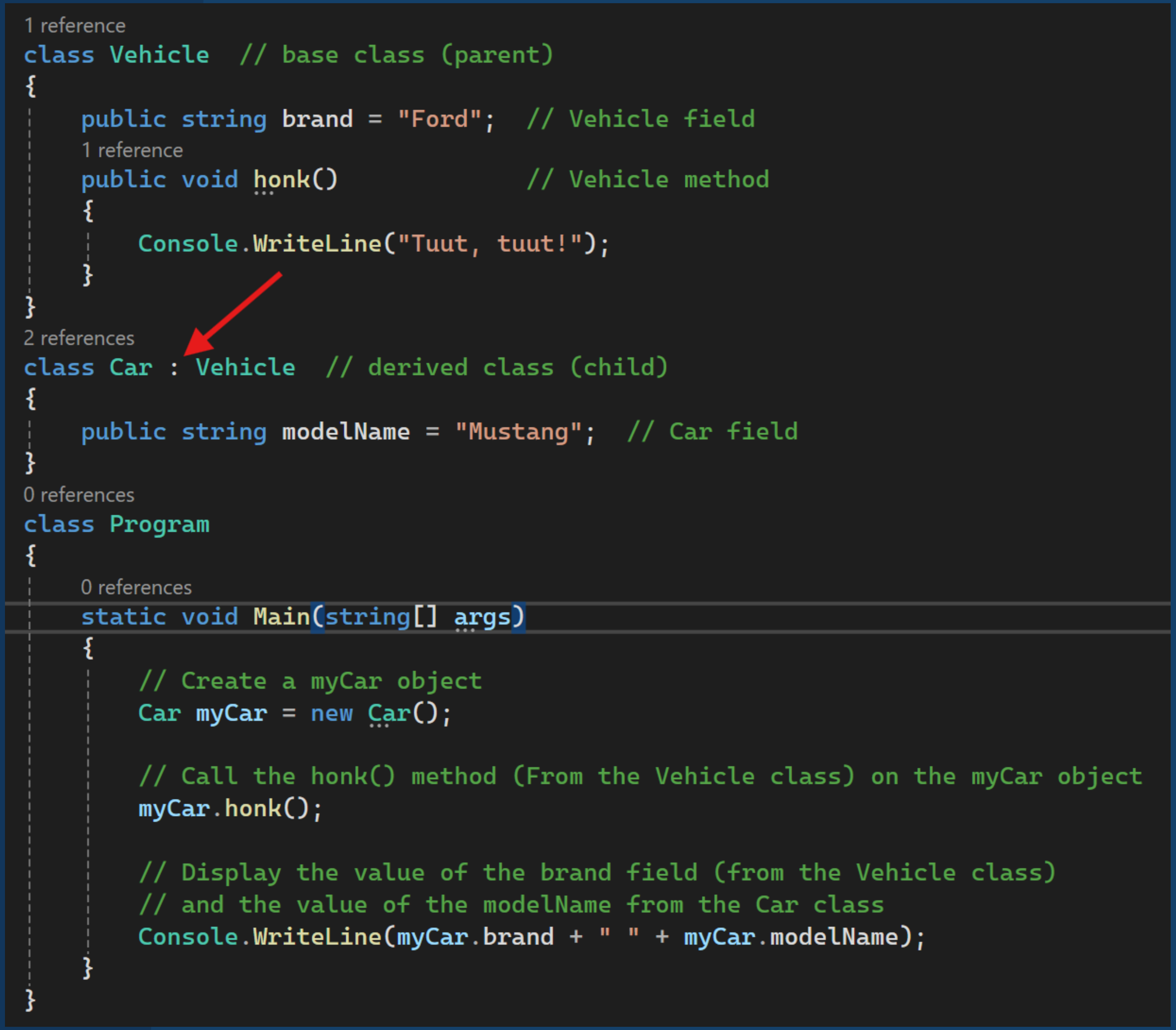
In C#, it is possible to inherit fields and methods from one class to another. We group the "inheritance concept" into two categories:

Derived Class (child) - the class that inherits from another class
Base Class (parent) - the class being inherited from

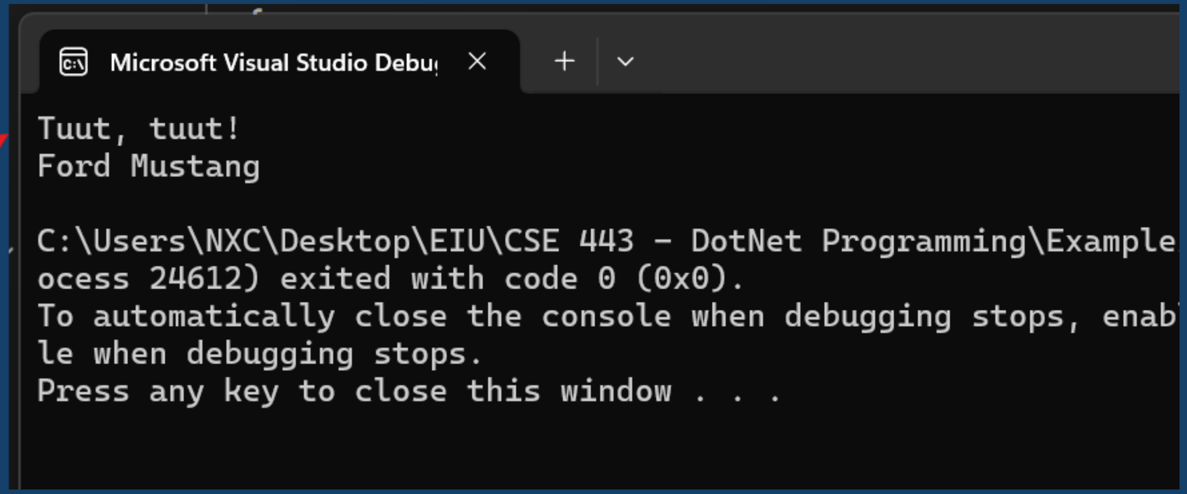
To inherit from a class, use the “ : ” symbol.

7. Inheritance

Example:



```
1 reference
class Vehicle  // base class (parent)
{
    public string brand = "Ford";  // Vehicle field
    1 reference
    public void honk()             // Vehicle method
    {
        Console.WriteLine("Tuut, tuut!");
    }
}
2 references
class Car : Vehicle  // derived class (child)
{
    public string modelName = "Mustang";  // Car field
}
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        // Create a myCar object
        Car myCar = new Car();

        // Call the honk() method (From the Vehicle class) on the myCar object
        myCar.honk();

        // Display the value of the brand field (from the Vehicle class)
        // and the value of the modelName from the Car class
        Console.WriteLine(myCar.brand + " " + myCar.modelName);
    }
}
```

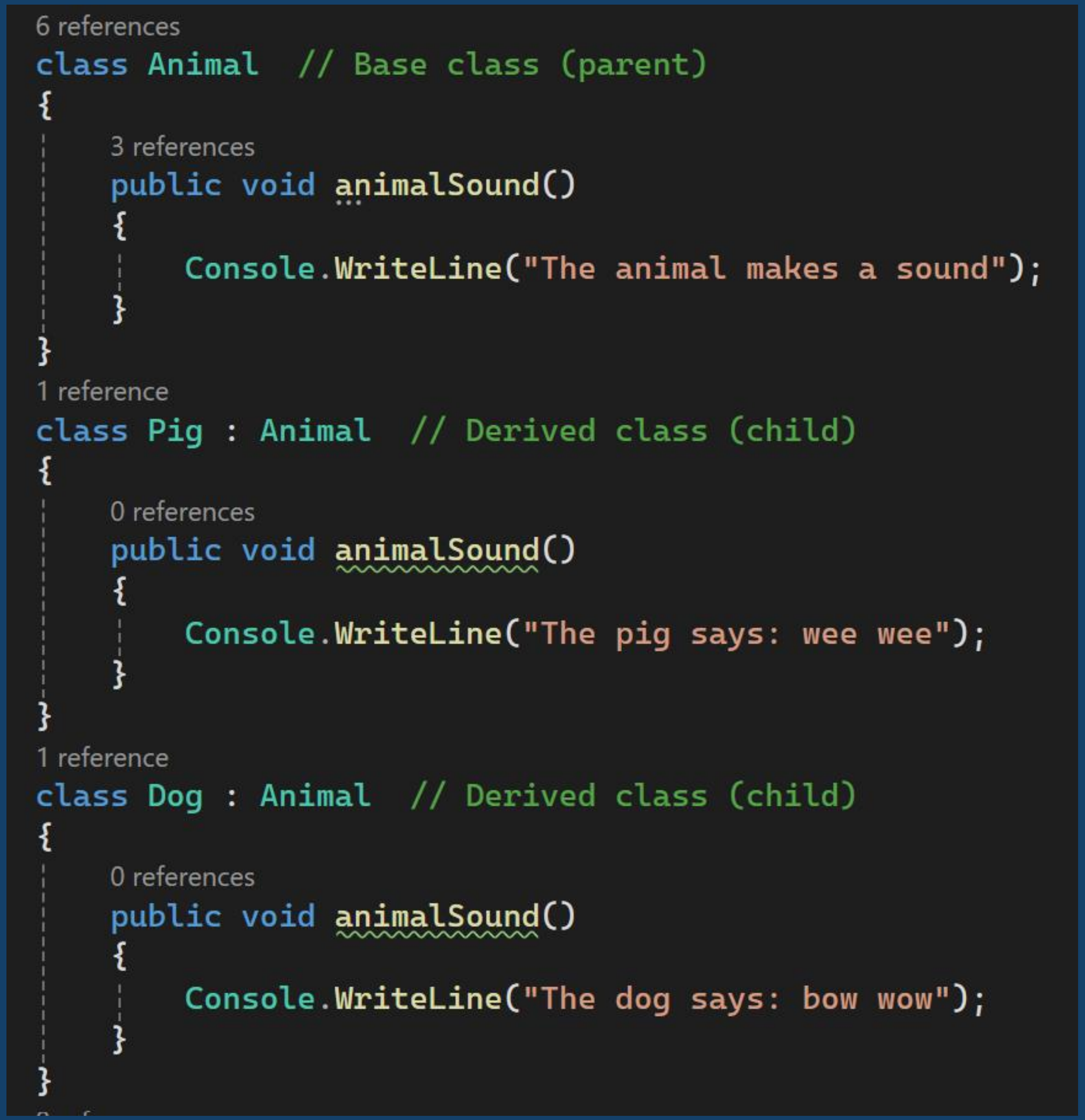


```
Tuut, tuut!
Ford Mustang

C:\Users\NXC\Desktop\EIU\CSE 443 - DotNet Programming\Example
ocess 24612) exited with code 0 (0x0).
To automatically close the console when debugging stops, enab
le when debugging stops.
Press any key to close this window . . .
```

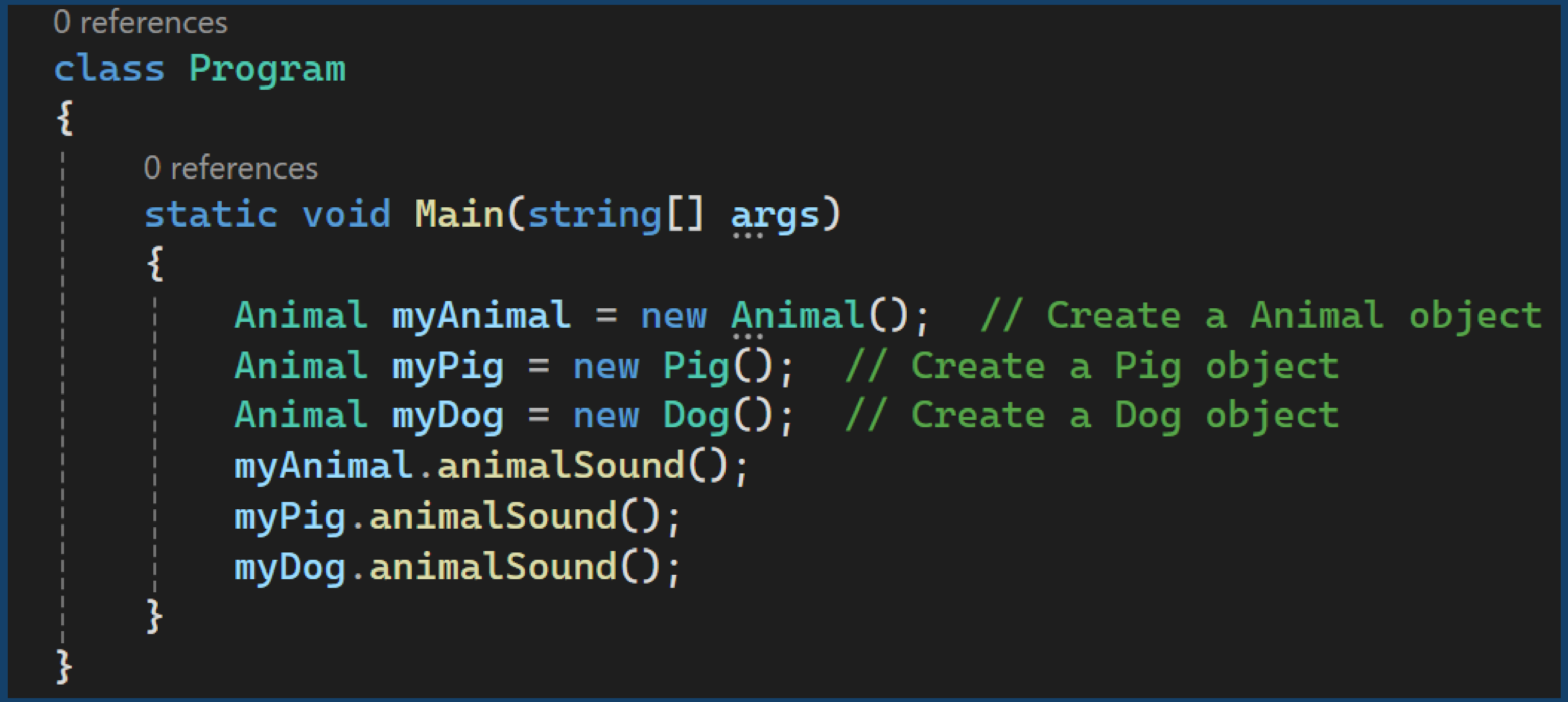
8. Polymorphism

Polymorphism and Overriding Methods

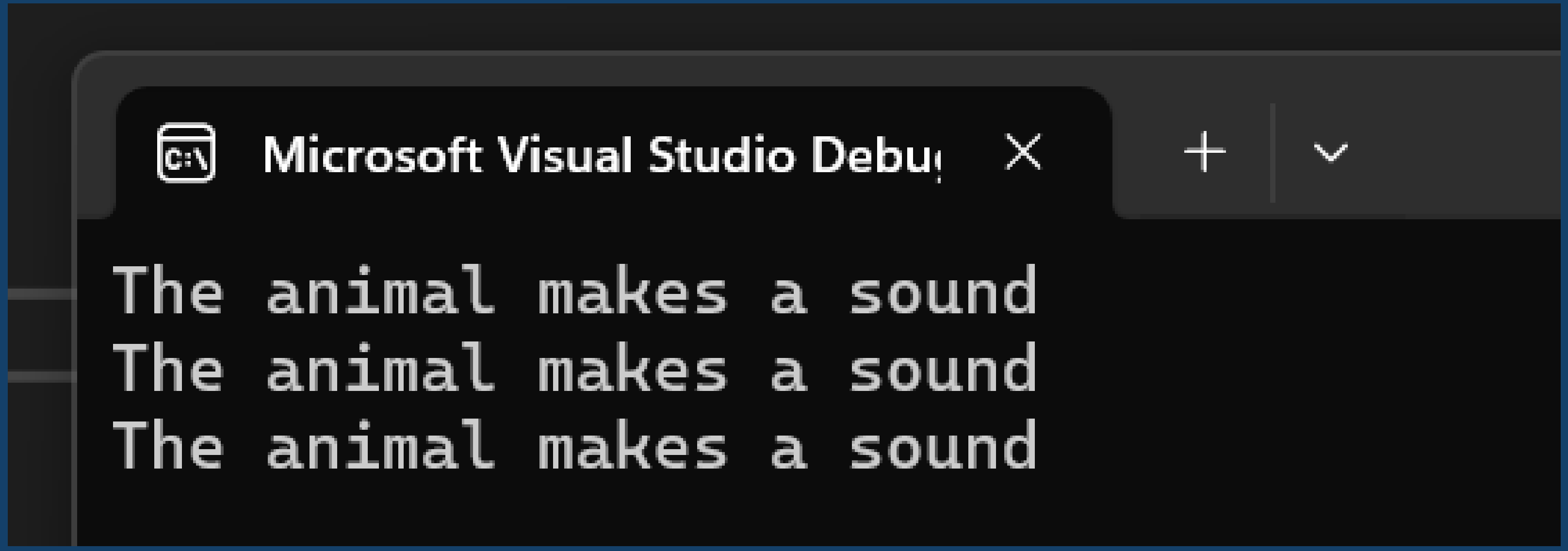
Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.



```
6 references
class Animal  // Base class (parent)
{
    3 references
    public void animalSound()
    {
        Console.WriteLine("The animal makes a sound");
    }
}
1 reference
class Pig : Animal  // Derived class (child)
{
    0 references
    public void animalSound()
    {
        Console.WriteLine("The pig says: wee wee");
    }
}
1 reference
class Dog : Animal  // Derived class (child)
{
    0 references
    public void animalSound()
    {
        Console.WriteLine("The dog says: bow wow");
    }
}
```



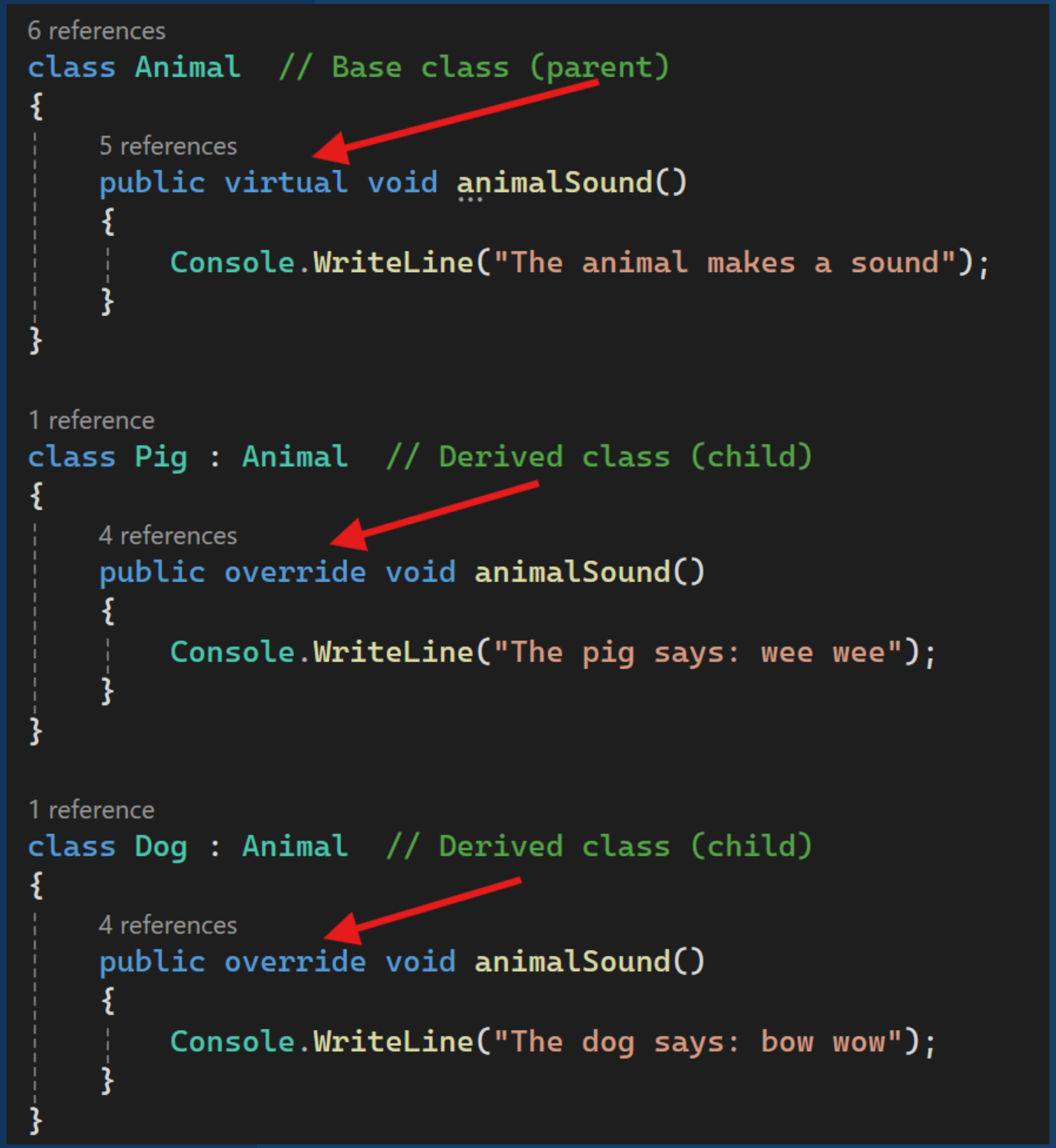
```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Animal myAnimal = new Animal();  // Create a Animal object
        Animal myPig = new Pig();  // Create a Pig object
        Animal myDog = new Dog();  // Create a Dog object
        myAnimal.animalSound();
        myPig.animalSound();
        myDog.animalSound();
    }
}
```



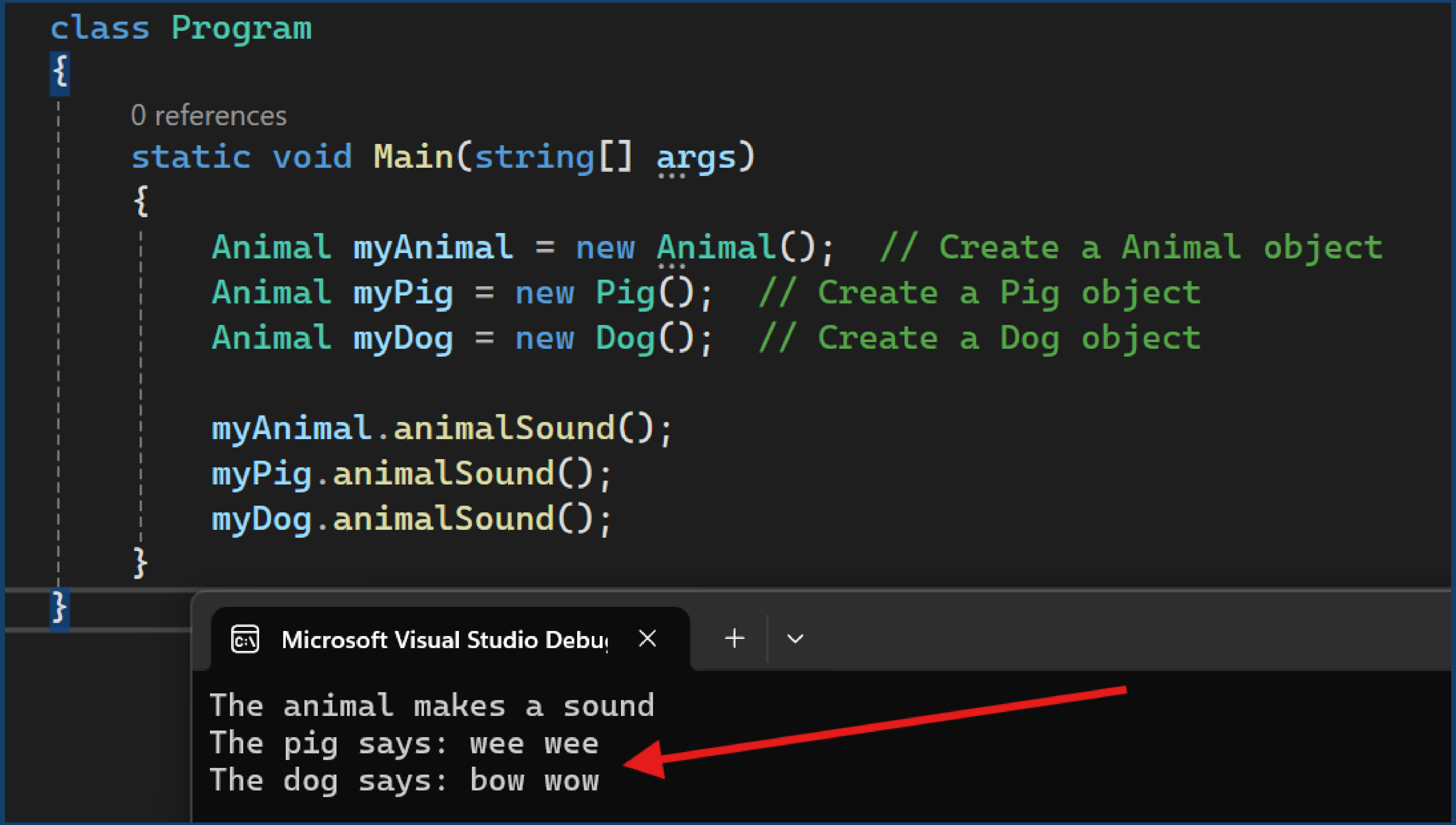
```
The animal makes a sound
The animal makes a sound
The animal makes a sound
```

8. Polymorphism

C# provides an option to override the base class method, by adding the virtual keyword to the method inside the base class, and by using the override keyword for each derived class methods



```
6 references
class Animal  // Base class (parent)
{
    5 references
    public virtual void animalSound()
    {
        Console.WriteLine("The animal makes a sound");
    }
}

1 reference
class Pig : Animal  // Derived class (child)
{
    4 references
    public override void animalSound()
    {
        Console.WriteLine("The pig says: wee wee");
    }
}

1 reference
class Dog : Animal  // Derived class (child)
{
    4 references
    public override void animalSound()
    {
        Console.WriteLine("The dog says: bow wow");
    }
}
```



```
class Program
{
    0 references
    static void Main(string[] args)
    {
        Animal myAnimal = new Animal();  // Create a Animal object
        Animal myPig = new Pig();  // Create a Pig object
        Animal myDog = new Dog();  // Create a Dog object

        myAnimal.animalSound();
        myPig.animalSound();
        myDog.animalSound();
    }
}
```

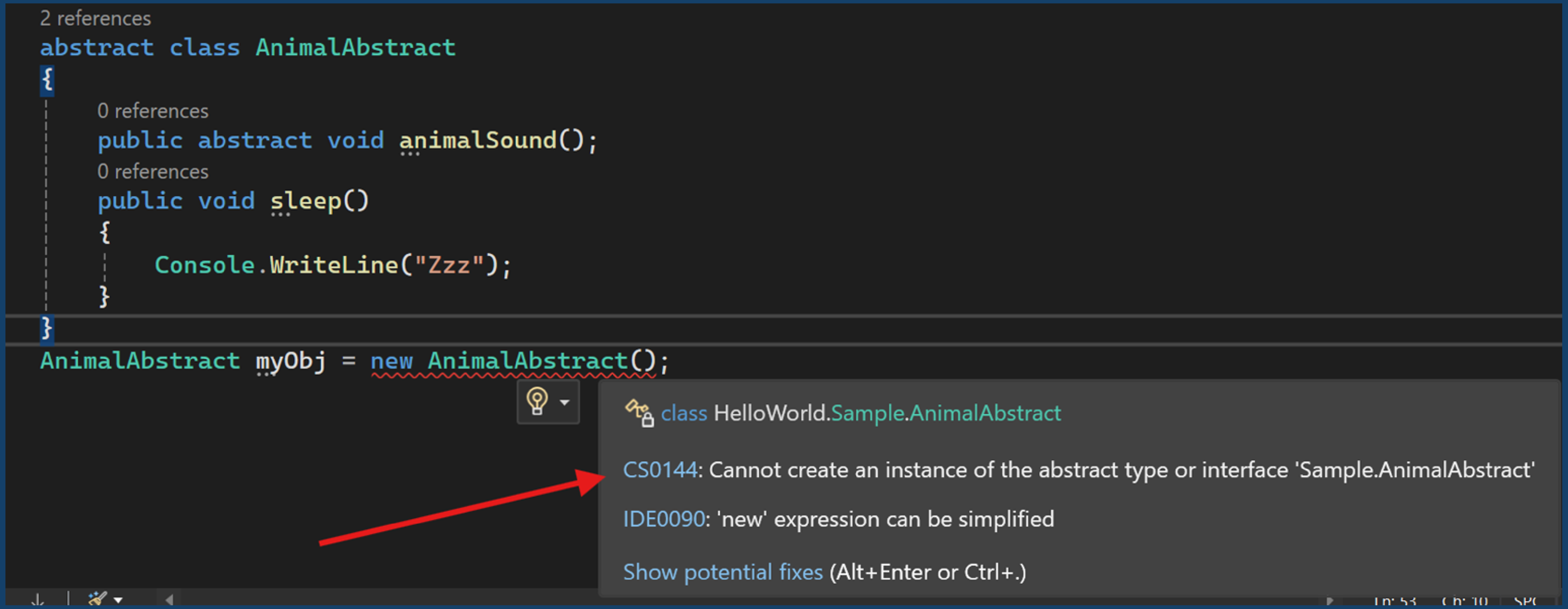
```
The animal makes a sound
The pig says: wee wee
The dog says: bow wow
```

9. Abstract

The abstract keyword is used for classes and methods:

Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the derived class (inherited from).
An abstract class can have both abstract and regular methods:



```
2 references
abstract class AnimalAbstract
{
    0 references
    public abstract void animalSound();
    0 references
    public void sleep()
    {
        Console.WriteLine("Zzz");
    }
}
AnimalAbstract myObj = new AnimalAbstract();
```

class HelloWorld.Sample.AnimalAbstract

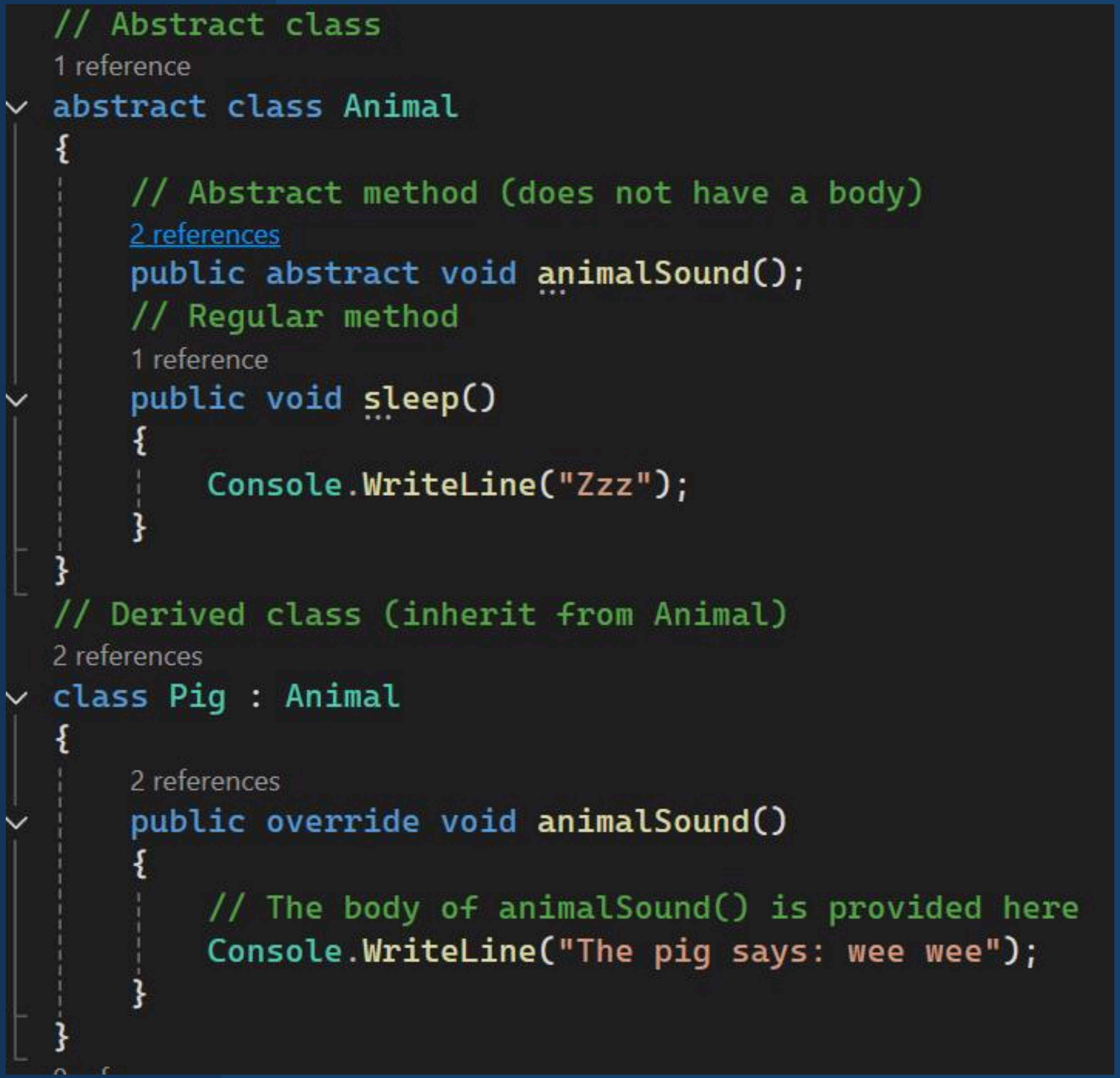
CS0144: Cannot create an instance of the abstract type or interface 'Sample.AnimalAbstract'

IDE0090: 'new' expression can be simplified

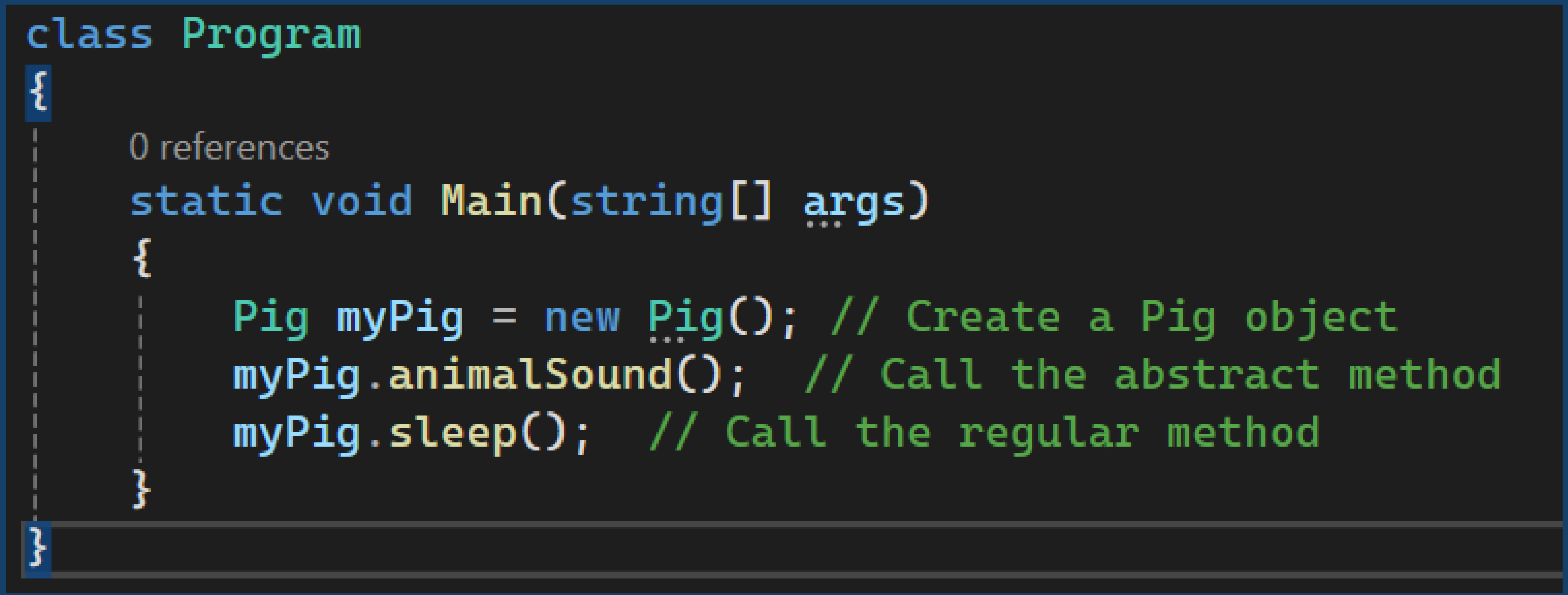
Show potential fixes (Alt+Enter or Ctrl+.)

9. Abstract

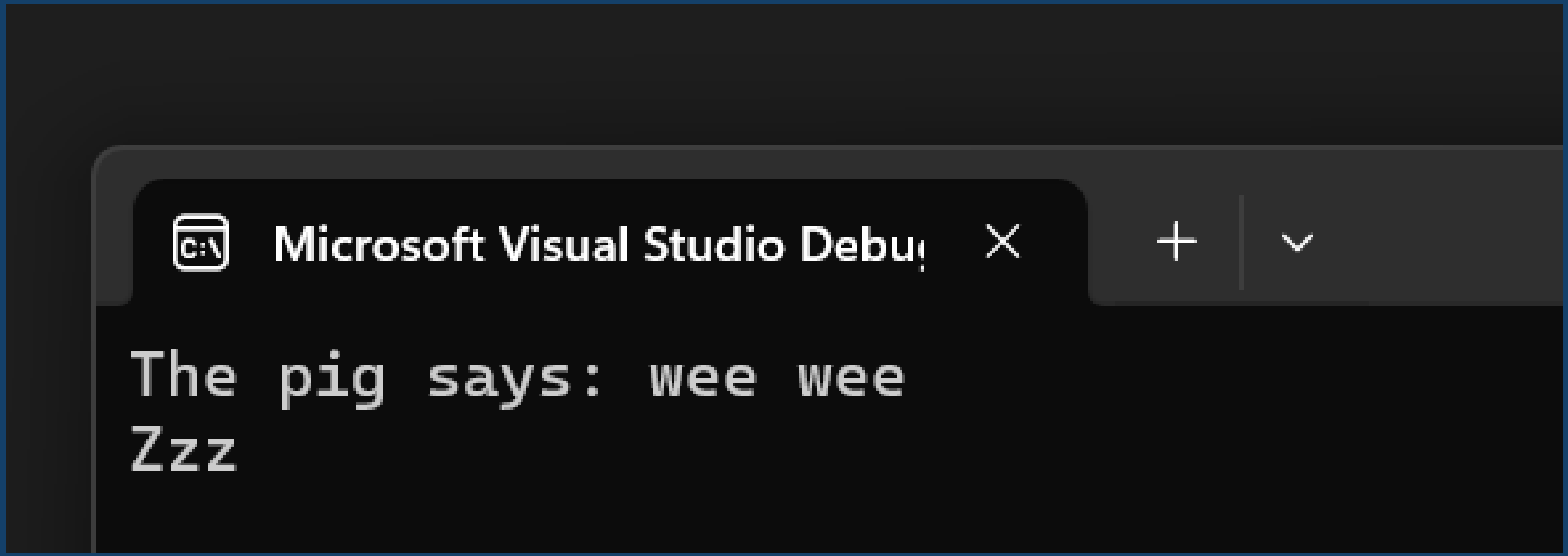
Example:



```
// Abstract class
1 reference
abstract class Animal
{
    // Abstract method (does not have a body)
    2 references
    public abstract void animalSound();
    // Regular method
    1 reference
    public void sleep()
    {
        Console.WriteLine("Zzz");
    }
}
// Derived class (inherit from Animal)
2 references
class Pig : Animal
{
    2 references
    public override void animalSound()
    {
        // The body of animalSound() is provided here
        Console.WriteLine("The pig says: wee wee");
    }
}
```



```
class Program
{
    0 references
    static void Main(string[] args)
    {
        Pig myPig = new Pig(); // Create a Pig object
        myPig.animalSound();  // Call the abstract method
        myPig.sleep();  // Call the regular method
    }
}
```

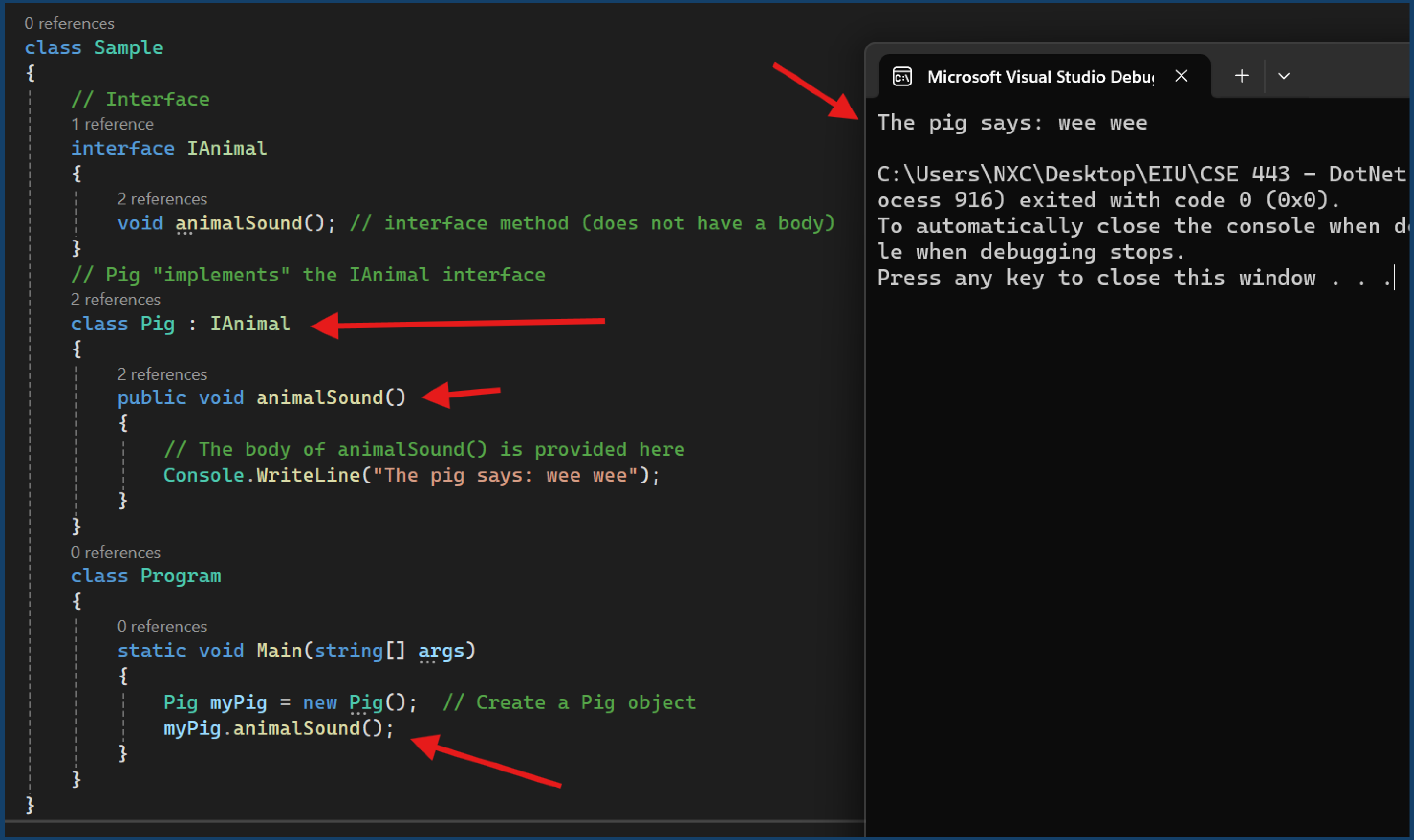


```
The pig says: wee wee
Zzz
```

9. Interface

Another way to achieve abstraction in C#, is with interfaces.

An interface is a completely "abstract class", which can only contain abstract methods and properties (with empty bodies):

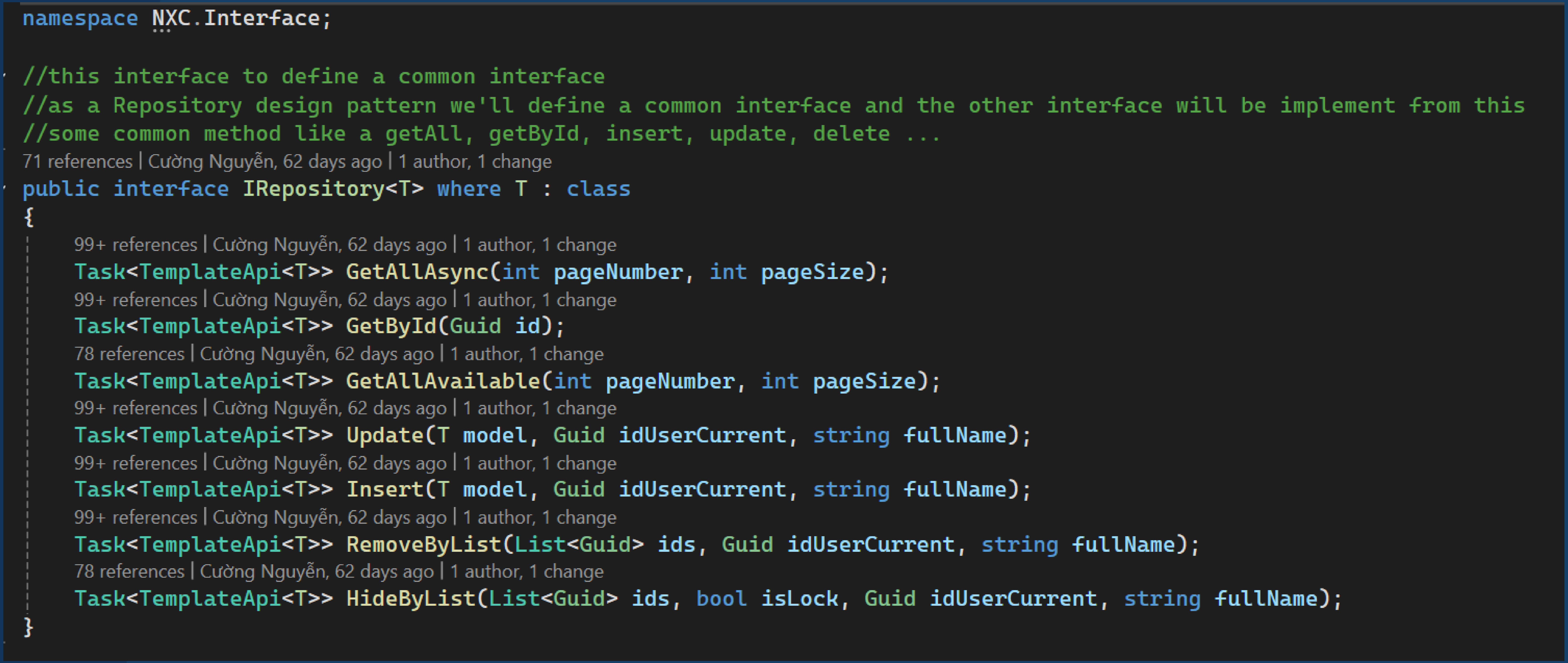


```
0 references
class Sample
{
    // Interface
    1 reference
    interface IAnimal
    {
        2 references
        void animalSound(); // interface method (does not have a body)
    }
    // Pig "implements" the IAnimal interface
    2 references
    class Pig : IAnimal
    {
        2 references
        public void animalSound()
        {
            // The body of animalSound() is provided here
            Console.WriteLine("The pig says: wee wee");
        }
    }
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Pig myPig = new Pig();  // Create a Pig object
            myPig.animalSound();
        }
    }
}
```

```
The pig says: wee wee

C:\Users\NXC\Desktop\EIU\CSE 443 - DotNet
ocess 916) exited with code 0 (0x0).
To automatically close the console when d
le when debugging stops.
Press any key to close this window . . .
```

9. Interface

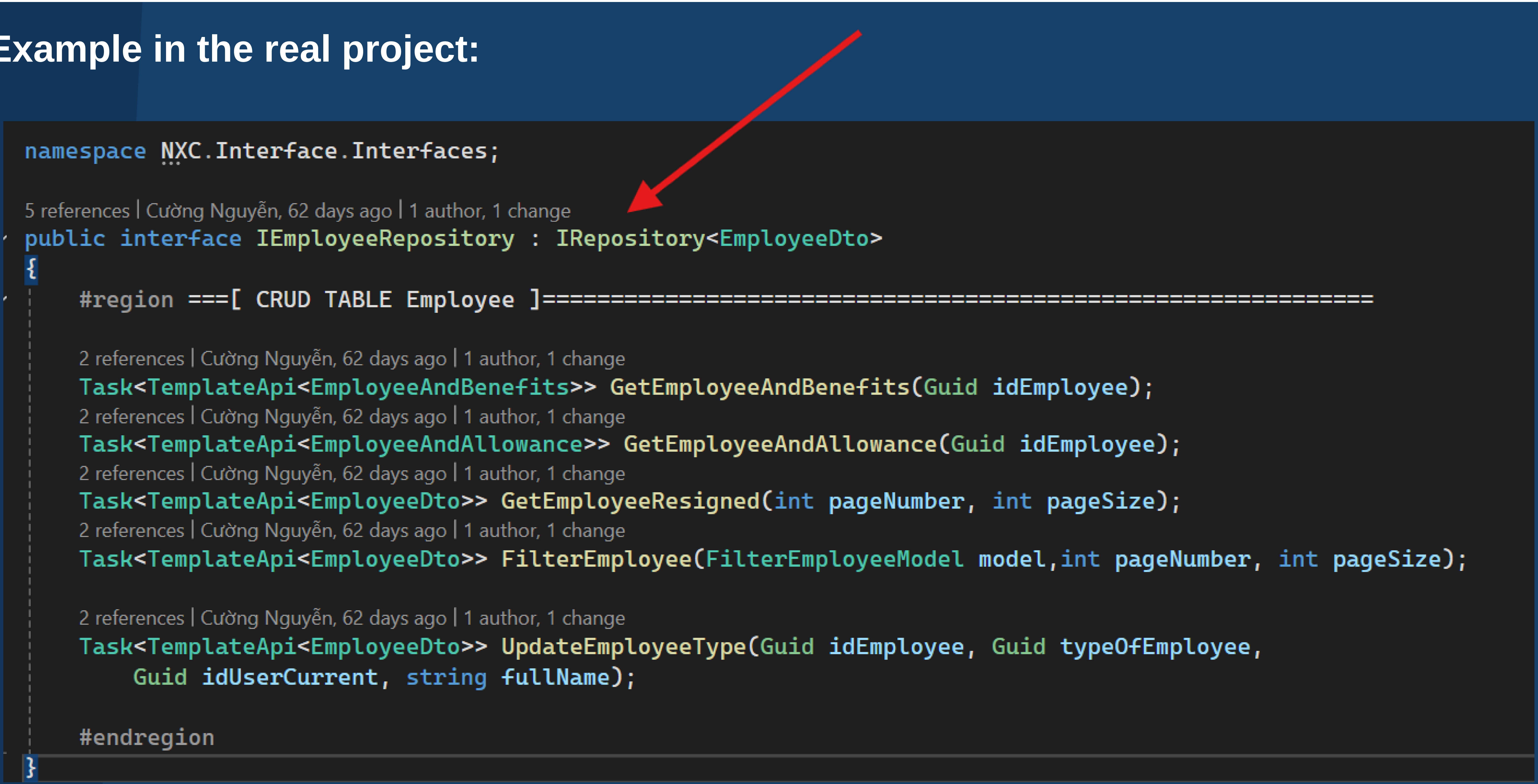
Example in the real project:



```
namespace NXC.Interface;

//this interface to define a common interface
//as a Repository design pattern we'll define a common interface and the other interface will be implement from this
//some common method like a getAll, getById, insert, update, delete ...
71 references | Cường Nguyễn, 62 days ago | 1 author, 1 change
public interface IRepository<T> where T : class
{
    99+ references | Cường Nguyễn, 62 days ago | 1 author, 1 change
    Task<TemplateApi<T>> GetAllAsync(int pageNumber, int pageSize);
    99+ references | Cường Nguyễn, 62 days ago | 1 author, 1 change
    Task<TemplateApi<T>> GetById(Guid id);
    78 references | Cường Nguyễn, 62 days ago | 1 author, 1 change
    Task<TemplateApi<T>> GetAllAvailable(int pageNumber, int pageSize);
    99+ references | Cường Nguyễn, 62 days ago | 1 author, 1 change
    Task<TemplateApi<T>> Update(T model, Guid idUserCurrent, string fullName);
    99+ references | Cường Nguyễn, 62 days ago | 1 author, 1 change
    Task<TemplateApi<T>> Insert(T model, Guid idUserCurrent, string fullName);
    99+ references | Cường Nguyễn, 62 days ago | 1 author, 1 change
    Task<TemplateApi<T>> RemoveByList(List<Guid> ids, Guid idUserCurrent, string fullName);
    78 references | Cường Nguyễn, 62 days ago | 1 author, 1 change
    Task<TemplateApi<T>> HideByList(List<Guid> ids, bool isLock, Guid idUserCurrent, string fullName);
}
```

9. Interface

Example in the real project:



```
namespace NXC.Interface.Interfaces;

5 references | Cường Nguyễn, 62 days ago | 1 author, 1 change
public interface IEmployeeRepository : IRepository<EmployeeDto>
{
    #region ===[ CRUD TABLE Employee ]=============================================================

    2 references | Cường Nguyễn, 62 days ago | 1 author, 1 change
    Task<TemplateApi<EmployeeAndBenefits>> GetEmployeeAndBenefits(Guid idEmployee);
    2 references | Cường Nguyễn, 62 days ago | 1 author, 1 change
    Task<TemplateApi<EmployeeAndAllowance>> GetEmployeeAndAllowance(Guid idEmployee);
    2 references | Cường Nguyễn, 62 days ago | 1 author, 1 change
    Task<TemplateApi<EmployeeDto>> GetEmployeeResigned(int pageNumber, int pageSize);
    2 references | Cường Nguyễn, 62 days ago | 1 author, 1 change
    Task<TemplateApi<EmployeeDto>> FilterEmployee(FilterEmployeeModel model,int pageNumber, int pageSize);

    2 references | Cường Nguyễn, 62 days ago | 1 author, 1 change
    Task<TemplateApi<EmployeeDto>> UpdateEmployeeType(Guid idEmployee, Guid typeOfEmployee,
        Guid idUserCurrent, string fullName);

    #endregion
}
```

Avoid In OOP

In OOP (Object-Oriented Programming), while there are many benefits such as code reuse, ease of maintenance, and scalability, improper use can lead to significant issues. Below are some things to avoid when working with OOP:

1. Overuse of Inheritance
Avoid creating complex inheritance relationships. Use **composition** or **interfaces** when appropriate instead of excessive inheritance.
2. Misusing Polymorphism
Use polymorphism only when necessary and ensure its purpose is clear to avoid confusion with too many virtual or override methods
.3. Failing to Apply SOLID Principles
Ignoring SOLID principles can lead to rigid, error-prone systems. Follow Single Responsibility and Open/Closed Principles for maintainable designs.

Avoid In OOP

4. Lack of Encapsulation

Avoid using public fields. Ensure data and logic are encapsulated and accessed through controlled methods (properties and methods).

5. Excessive Class Creation

Don't break classes down excessively. Keep the number of classes reasonable to maintain clarity and manageability.

6. Misusing Inheritance Instead of Composition

Prefer composition for flexibility instead of forcing inheritance where it’s not necessary.

Avoid In OOP

7. Poor Error and Exception Handling

Handle errors and exceptions properly. Avoid generic try-catch blocks without specific handling and proper logging.

8. Violating the "Tell, Don’t Ask" Principle

Let objects perform actions instead of querying their state and processing externally.

9. God Objects

Avoid creating objects with too many responsibilities. Follow the Single Responsibility Principle.

Avoid In OOP

10. Overuse of Getter and Setter Methods

Use getter and setter methods only when needed to preserve encapsulation.

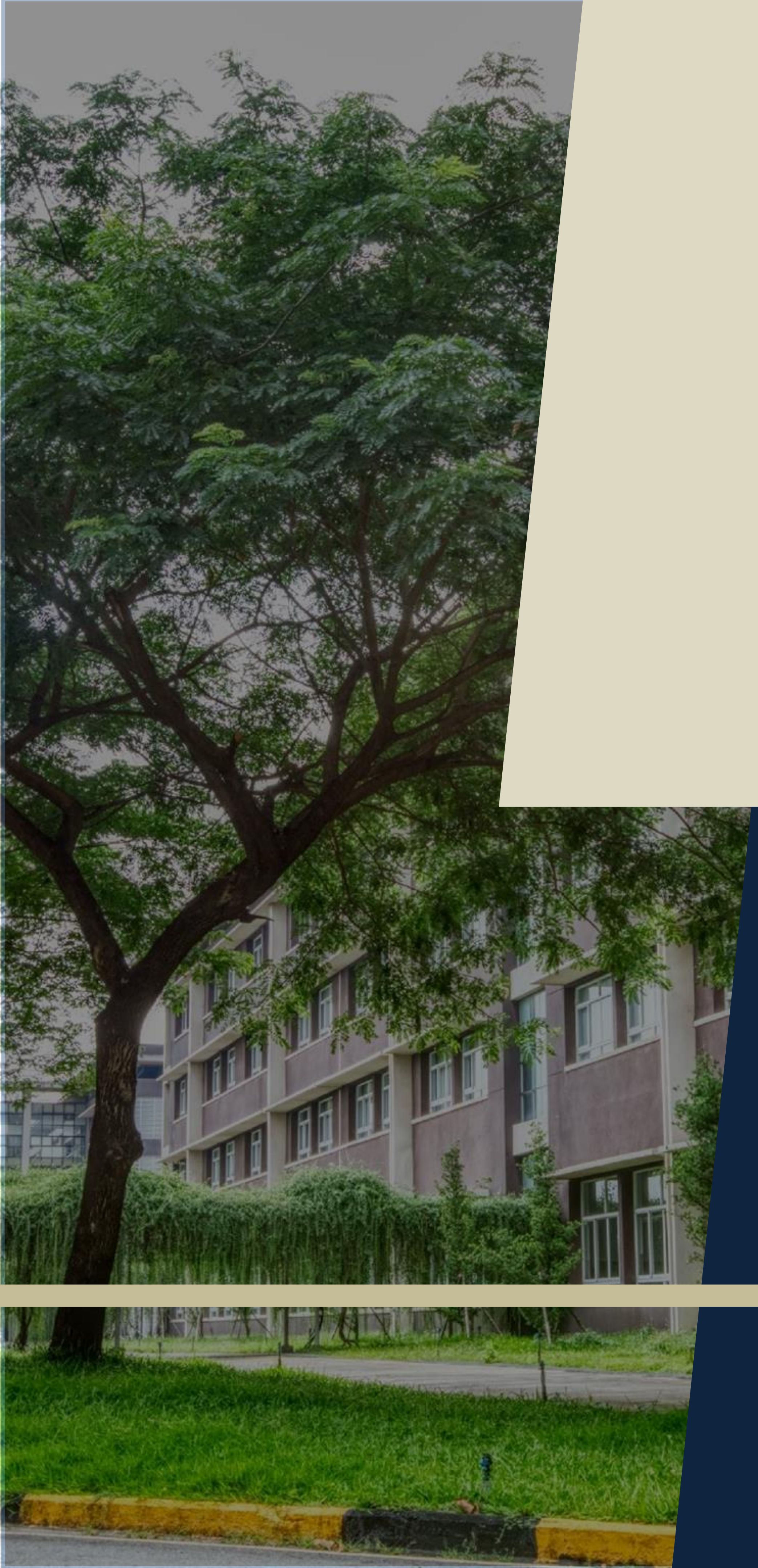
11. Poor Memory Management

Manage memory and resources properly using IDisposable to ensure resources are released when no longer needed.

TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG
EASTERN
INTERNATIONAL
UNIVERSITY

Start your future at EIU

Q&A



EIU
TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG
EASTERN
INTERNATIONAL
UNIVERSITY

Start your future at EIU

THANK YOU