Inheritance

Course Code: CSC 3115 Course Title: Object Oriented Programming 2

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	04	**Week No:**	04	**Semester:**	
Lecturer:					

Topics

1. Inheritance
2. HAS-A relationship
3. IS-A relationship
4. Single
5. Multilevel
6. Hierarchical Inheritance

Inheritance

Inheritance is ability for **derived class** to inherit all the members of the **base class**
Derived class also can be referred to as “child class” or “subclass”
Base class also can be referred to as “parent class” or “superclass”
Syntax to specify inheritance

```
class SubClass : SuperClass
```

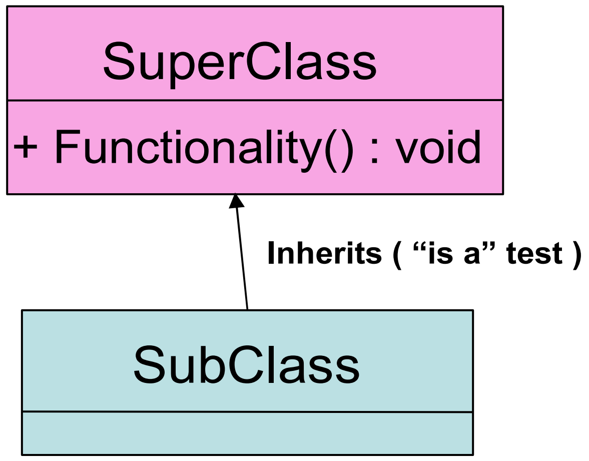
Inheritance allows to implement **Specialization** and **Generalization** concepts e.g.:

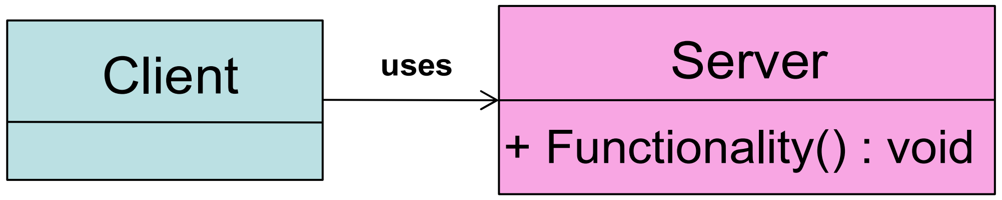
Radio button **is-a** specialized case of Control
Control is a **base** class
Radio button **derived** class

Inheritance

- C# supports “single inheritance” + multiple “interface implementation” inheritance (see interfaces)
- In "single inheritance," a common form of inheritance, classes have **only one** base class.
- Sometimes “implementation” inheritance is over-used. It leads to tight-coupling between parent and child. If coupling is “natural” then ok... if not, then not ok...
- Should satisfy the “**is a**” test...
- Alternative to inheritance is composition

Inheritance





- composition is dynamic

```
class Client
{
        // here we instanciate the ‘Server’ class
        Server s = ...
        public void MethodUsingServer()
        {
    // and use its instance methods from within the ‘Client’ class
    // thus ‘composing’ the client without inheritance from any ‘base’
                s.Functionality();
        }
    }
```

- inheritance is static

```
class SubClass : SuperClass
{ … }
```

Inheritance

- Keywords relevant to inheritance + polymorphism
  - **abstract** – class or method is not concrete; sub-classes are expected to provide the implementation
  - **virtual** – indicates a method can be overridden in the sub-class
  - **override** – override an implementation in the super-class; only abstract and virtual methods can be overridden
  - **new** – note the overloaded usage... indicates a method is a new implementation that should “hide” the implementation in the super-class

What is Inheritance?

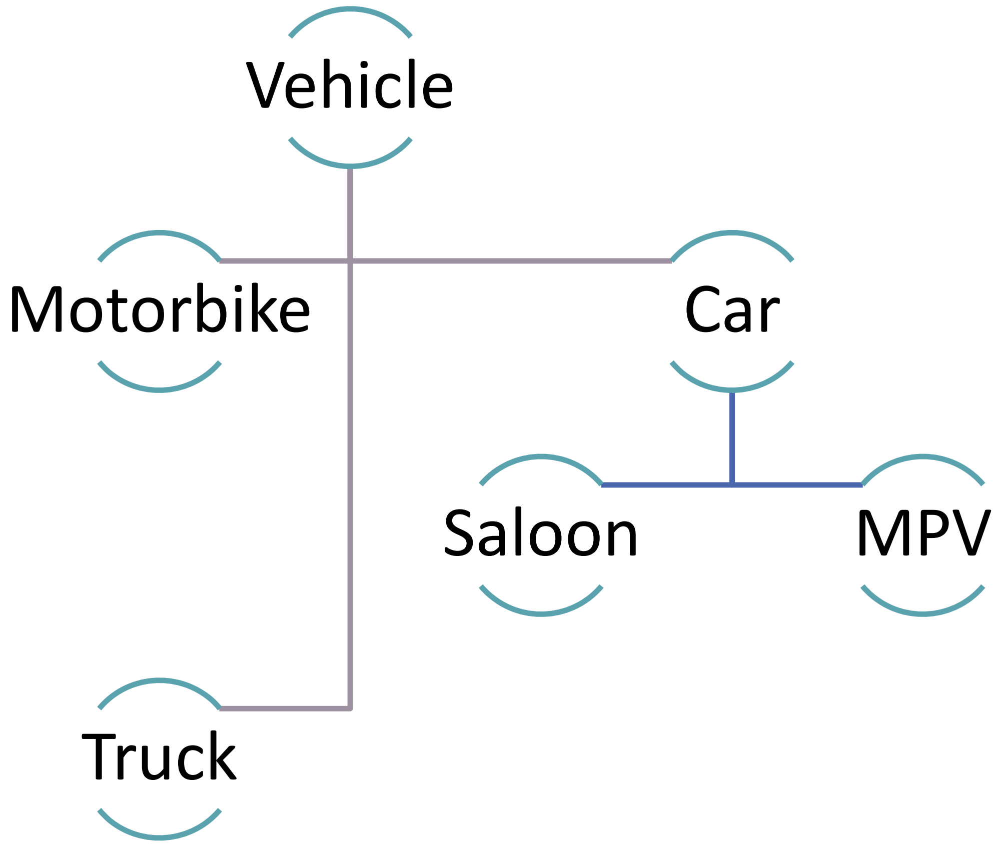
- A relationship between a more general class (called the **superclass or base class**) and a more specialised class(called the **subclass or derived class**).
- For example, a Cat is a specific type of Animal. Therefore the **Animal** class could be the **superclass/base class**, and the **Cat** class could be a **subclass** of Animal.

Why is Inheritance important?

- Inheritance is a fundamental concept in Object-Oriented programming.
- Many objects exist that share a lot of commonality between them. By using inheritance, subclasses **“inherit”** the **public** **variables and methods** of the **superclass,** in addition to their own variables and methods.
- Any private method or variables from the superclass will not be inherited.
- Inheritance helps to improve code re-use and minimise duplicate code among related classes.

Inheritance Hierarchies

- We can represent inheritance among objects via an inheritance hierarchy.



IS-A Relationship

- Use the **IS-A** test to verify that your inheritance hierarchy is valid.
- A Dog **IS-A** Animal – makes sense, therefore the Dog class can inherit from an Animal class.
- A Door **IS-A** Car– doesn’t make sense, so the Door class shouldn’t inherit from the Car class.
- An Animal **IS-A** Dog – doesn’t make sense. A Dog IS-A Animal, but an Animal is not a Dog.
- The IS-A test only works in **one direction** i.e. a Cat IS-A Animal, but an Animal is not a Cat.

HAS-A Relationships

- Use the **HAS-A** test to verify whether an object should contain another object as an instance variable. This is called object composition.
- A Car **IS-A** Engine – doesn’t make sense.
- A Car **HAS-A** Engine – This makes sense. So therefore our Car class would have an instance variable of type Engine.

```
public class Car{
        private Engine carEngine;
        public Car(){
                carEngine = new Engine();
        }
}
```

Inheritance in C#

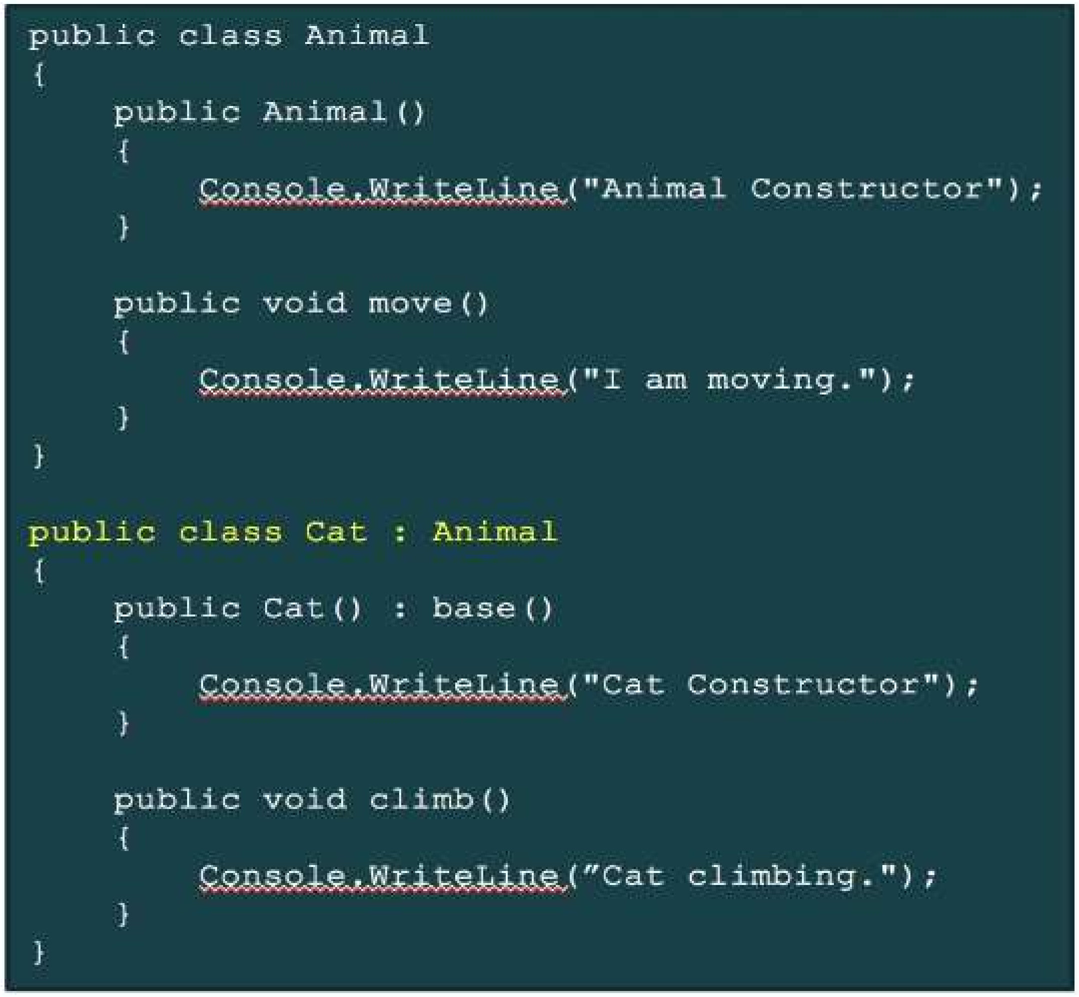
```
public class Cat : Animal
{
    public Cat() : base()
    {
        Console.WriteLine("Cat Constructor");
    }

    public void runUpATree()
    {
        Console.WriteLine("I am sleeping");
    }
}
```

- To declare a subclass in C#, we use the following notation

```
public class Subclass : BaseClass
```

- We give the subclass a name followed by a colon (:) and the name of the base class we want to inherit from.

C# Inheritance Ex. 1



```
public class Animal
{
    public Animal()
    {
        Console.WriteLine("Animal Constructor");
    }

    public void move()
    {
        Console.WriteLine("I am moving.");
    }
}

public class Cat : Animal
{
    public Cat() : base()
    {
        Console.WriteLine("Cat Constructor");
    }

    public void climb()
    {
        Console.WriteLine(”Cat climbing.");
    }
}
```

- Here the Cat class is inheriting all the public methods of the Animal class as well as it’s own methods.
- Now the following methods can be called on a Cat.

```
Cat cat = new Cat();

cat.move();

cat.climb();
```

C# Inheritance Ex. 2

Superclass

```
public class Bird{
    protected bool canFly;
    protected void setCanFly(bool cFly)
   {
        canFly = cFly;
    }

    public void message(){
        if (this.canFly){
           Console.WriteLine("This bird
           can fly");
        }
        else{
          Console.WriteLine("This bird
          cannot fly");
        }
    }
}
```

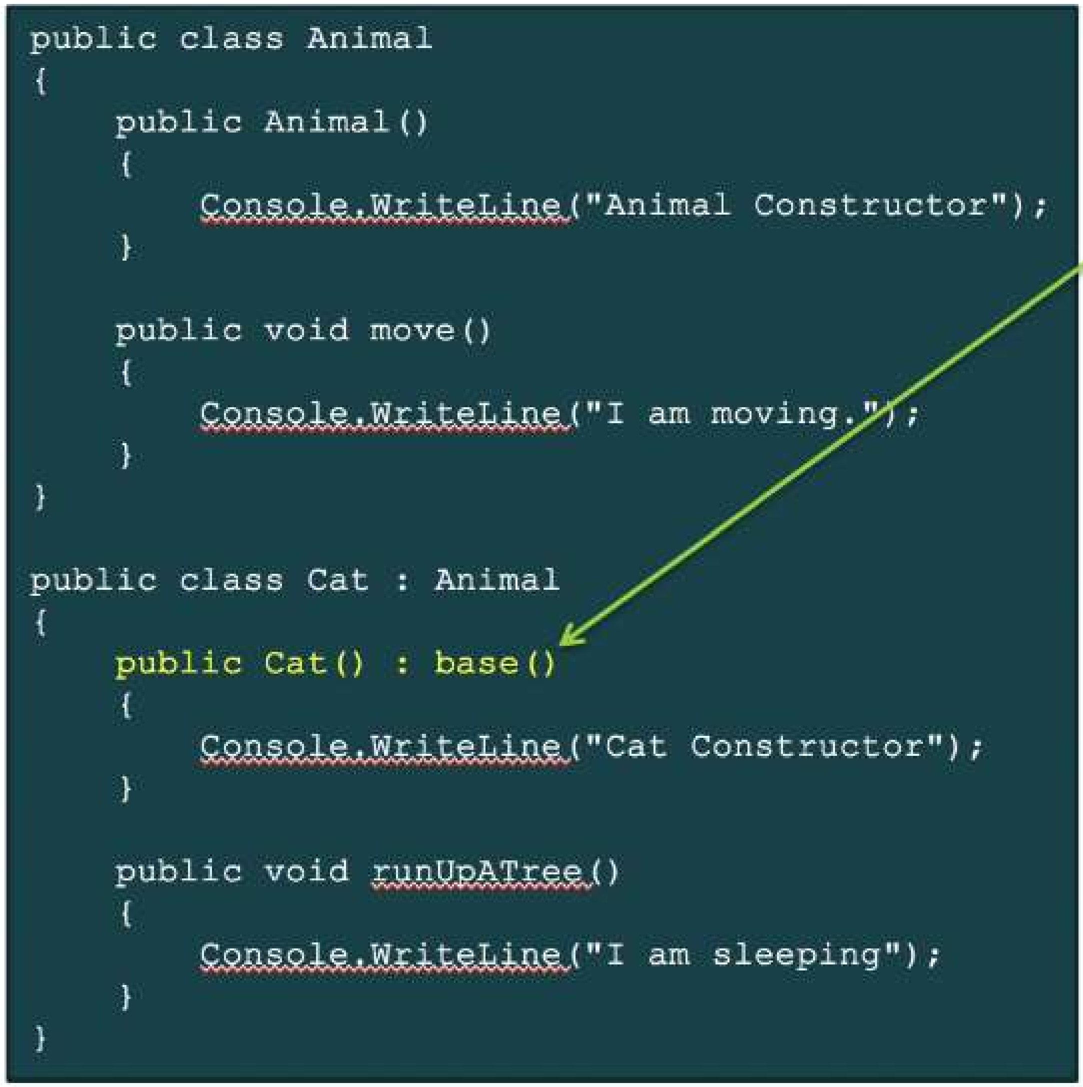
Subclass

```
public class Robin: Bird
{
    public Robin(){
        setCanFly(true);
    }
}

class birdTest{
    static void Main(string[] args)
        {
            Robin robin1 = new Robin();
            robin1.message();
            Console.Read();
        }
}
```

Calling Superclass Constructors

- When creating subclass objects, it is important that the superclass constructor of that object is called in the subclass constructor.
- The reasoning behind this is in order to use the subclass, we want to ensure that the superclass and any instance variables or set up logic that it may have is fully initialised before continuing with the more specific subclass.
- For example, when a Cat object is being constructed we want to call the constructor of the superclass Animal first, before setting up our Cat.

Constructing Cat subclass Ex



```
public class Animal
{
    public Animal()
    {
        Console.WriteLine("Animal Constructor");
    }

    public void move()
    {
        Console.WriteLine("I am moving.");
    }
}

public class Cat : Animal
{
    public Cat() : base()
    {
        Console.WriteLine("Cat Constructor");
    }

    public void runUpATree()
    {
        Console.WriteLine("I am sleeping");
    }
}
```

- The keyword **base** is used for calling the superclass constructor.
- Therefore whenever we construct a Cat, the Animal constructor is called first, then the Cat constructor.

Constructing Subclass Ex. 2

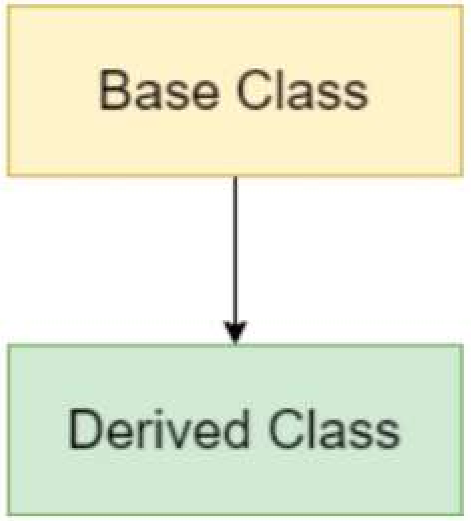
- If our Animal constructor requires a name parameter, then whenever we construct a Cat object, we want our animal's name to be initialised.

```
public class Animal
{
    private String Name;
    public Animal(String name)
    {
         Name = name;
    }

    public void move()
    {
        Console.WriteLine("I am moving.");
    }
}

public class Cat : Animal
{
    public Cat(String catName) : base(catName)
    {
        Console.WriteLine("Cat Constructor");
    }
}
```

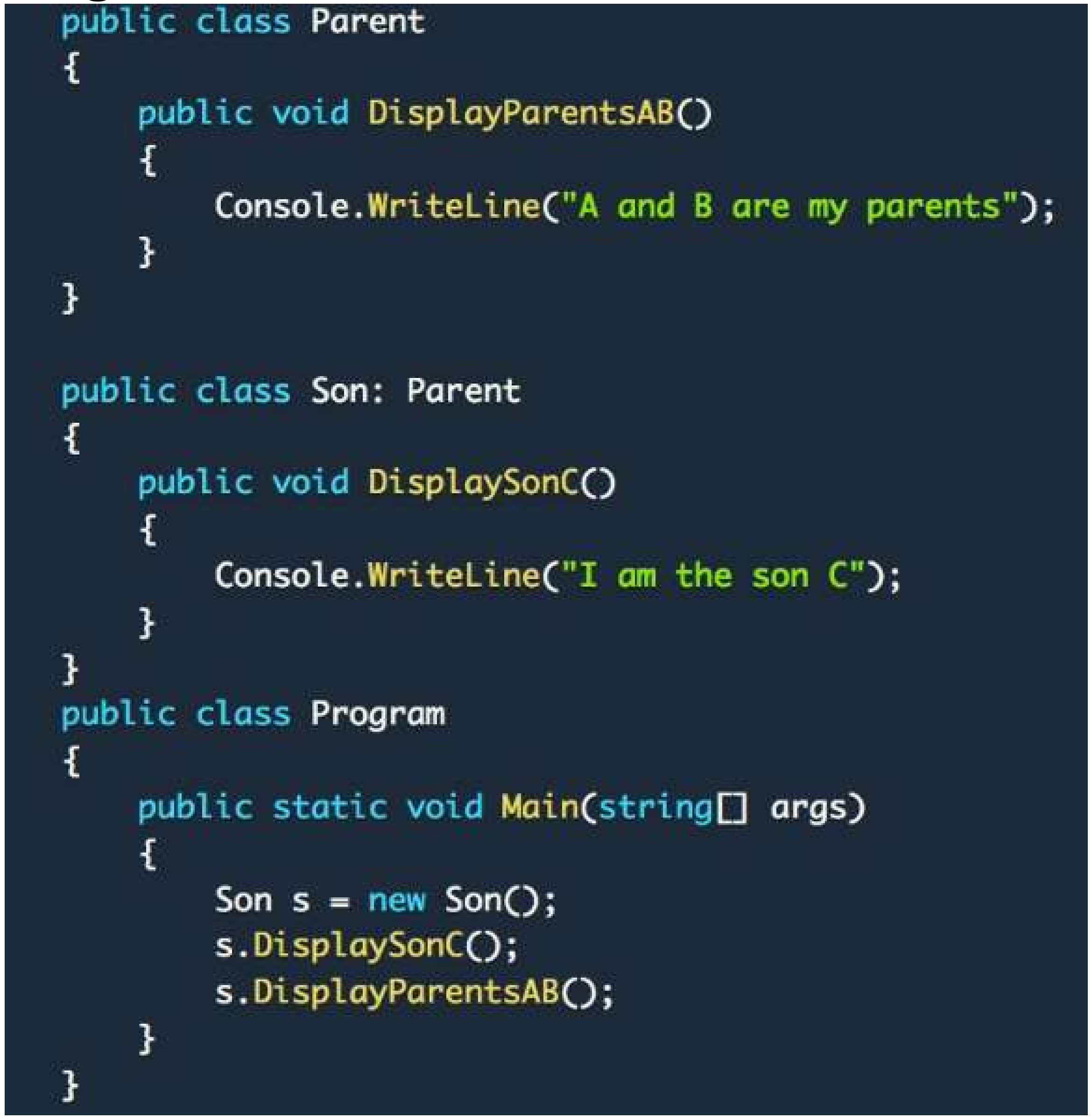
- We can achieve this by adding a parameter to our Cat constructor called catName, and then pass this to the superclass constructor using base(catName).

Single Inheritance

In this type of inheritance, the derived class inherits properties and behavior from a single base class. It's like a child inherits the traits of his/her parents.

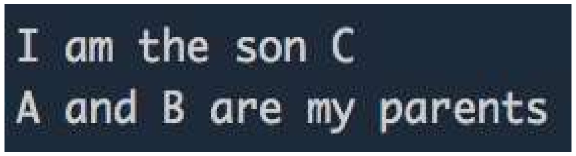


Single Inheritance



```
public class Parent
{
    public void DisplayParentsAB()
    {
        Console.WriteLine("A and B are my parents");
    }
}

public class Son: Parent
{
    public void DisplaySonC()
    {
        Console.WriteLine("I am the son C");
    }
}
public class Program
{
    public static void Main(string[] args)
    {
        Son s = new Son();
        s.DisplaySonC();
        s.DisplayParentsAB();
    }
}
```

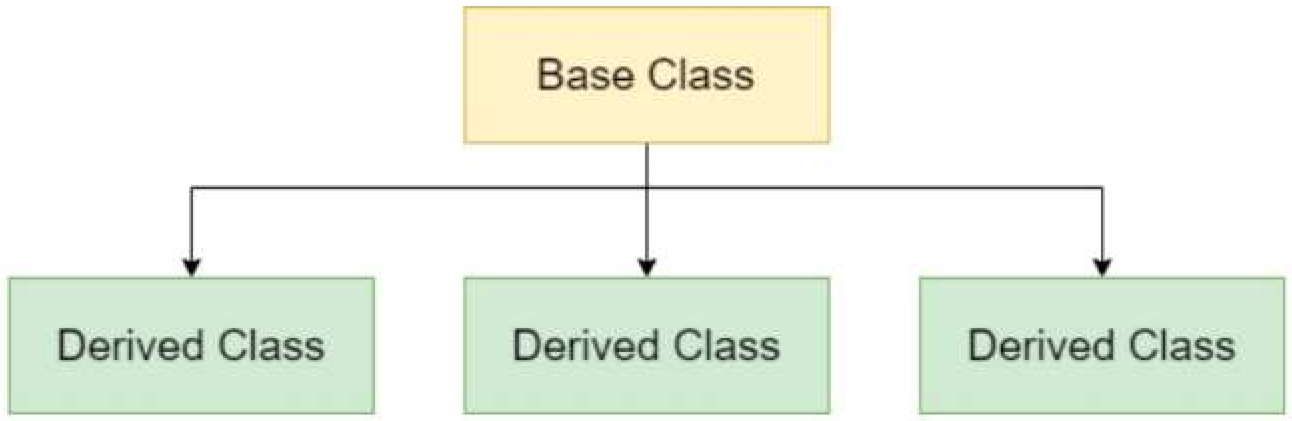
Output:

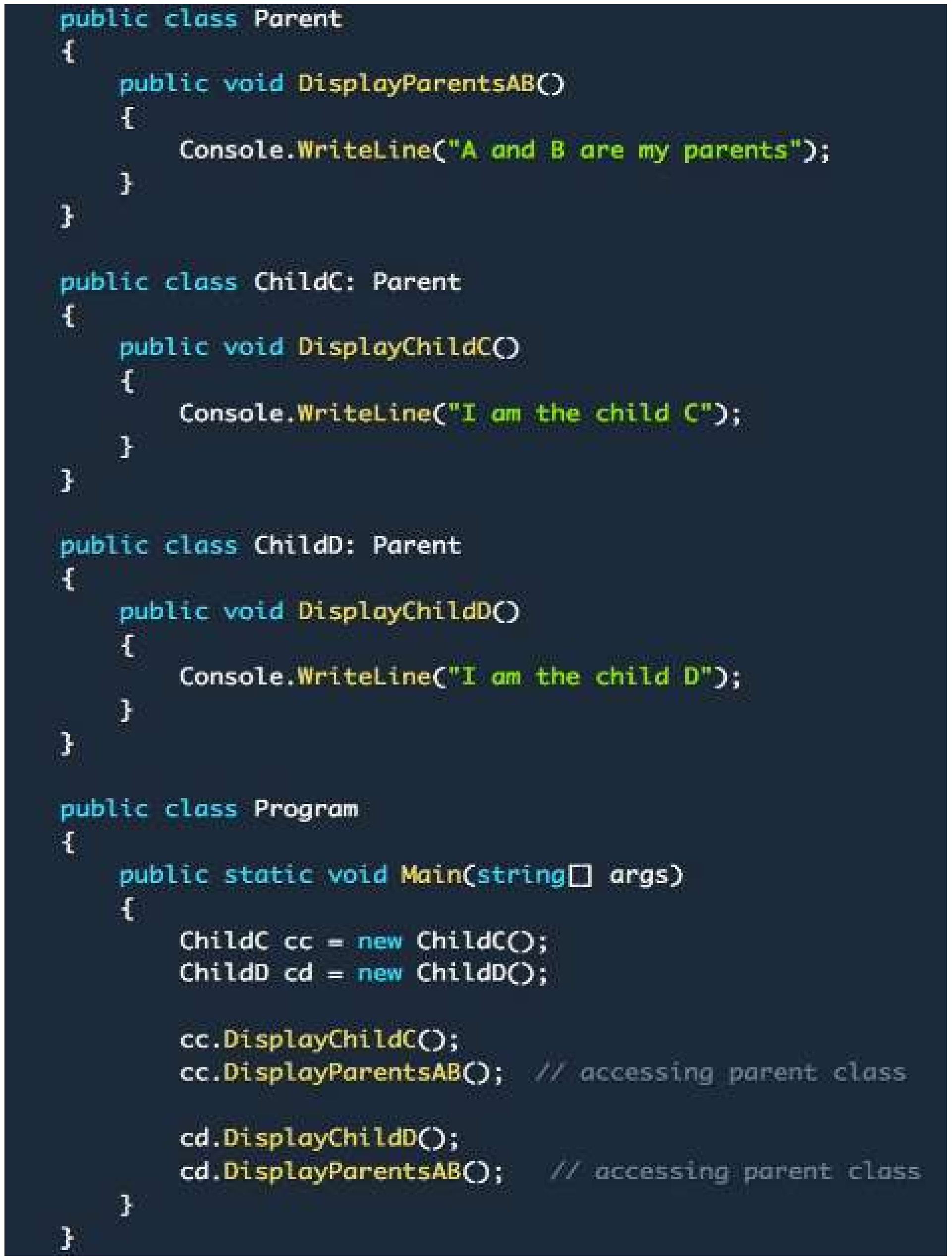


```
I am the son C
A and B are my parents
```

Hierarchical Inheritance

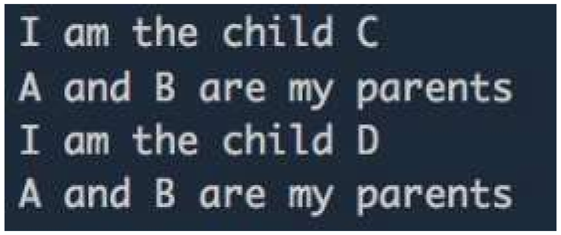
In this type of inheritance, the multiple classes derives from one base class. It's like having multiple kids, all inheriting traits from parent, but in their own different ways.





```
public class Parent
{
    public void DisplayParentsAB()
    {
        Console.WriteLine("A and B are my parents");
    }
}

public class ChildC: Parent
{
    public void DisplayChildC()
    {
        Console.WriteLine("I am the child C");
    }
}

public class ChildD: Parent
{
    public void DisplayChildD()
    {
        Console.WriteLine("I am the child D");
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        ChildC cc = new ChildC();
        ChildD cd = new ChildD();

        cc.DisplayChildC();
        cc.DisplayParentsAB();  // accessing parent class

        cd.DisplayChildD();
        cd.DisplayParentsAB();   // accessing parent class
    }
}
```

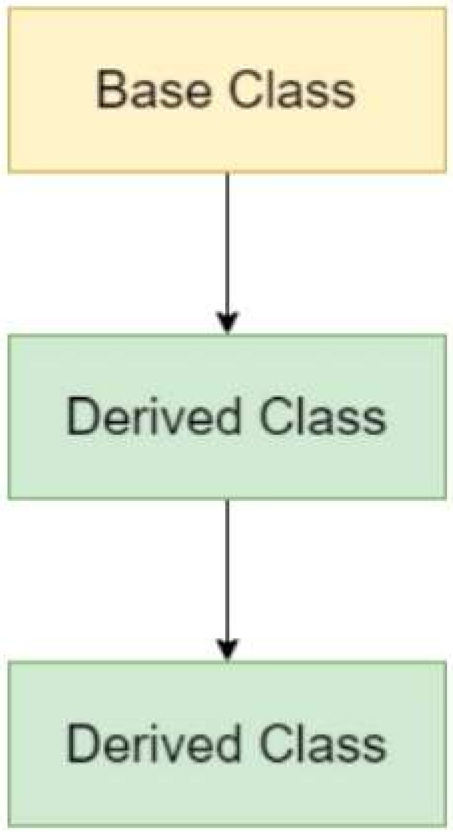
Output:

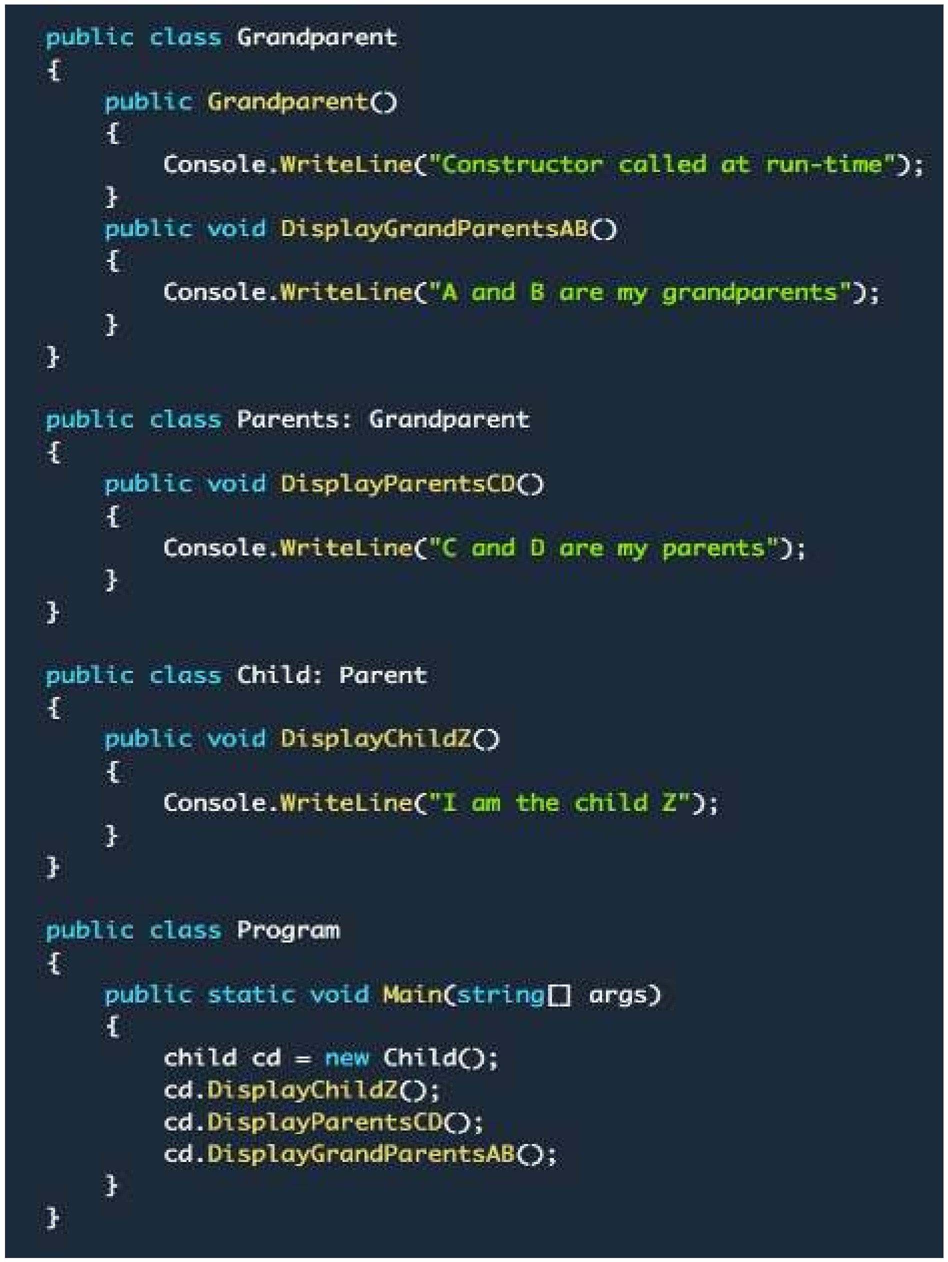


```
I am the child C
A and B are my parents
I am the child D
A and B are my parents
```

Multilevel Inheritance

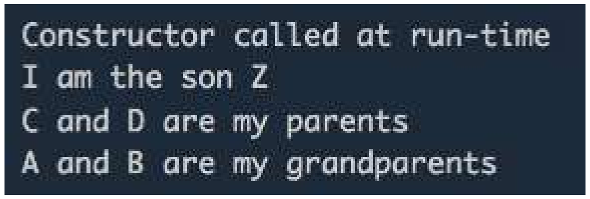
In this type of inheritance, a class inherits another derived/child class which in turn inherits another class. It's like a child inherits the traits of his/her parents, and parents inherit the traits of their grandparents.





```
public class Grandparent
{
    public Grandparent()
    {
        Console.WriteLine("Constructor called at run-time");
    }
    public void DisplayGrandParentsAB()
    {
        Console.WriteLine("A and B are my grandparents");
    }
}

public class Parents: Grandparent
{
    public void DisplayParentsCD()
    {
        Console.WriteLine("C and D are my parents");
    }
}

public class Child: Parent
{
    public void DisplayChildZ()
    {
        Console.WriteLine("I am the child Z");
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        child cd = new Child();
        cd.DisplayChildZ();
        cd.DisplayParentsCD();
        cd.DisplayGrandParentsAB();
    }
}
```

Output:



```
Constructor called at run-time
I am the son Z
C and D are my parents
A and B are my grandparents
```

Thank You

Books

- C# 4.0 The Complete Reference; Herbert Schildt; McGraw-Hill Osborne Media; 2010
- Head First C# by Andrew Stellman
- Fundamentals of Computer Programming with CSharp – Nakov v2013

References

MSDN Library; URL: http://msdn.microsoft.com/library

C# Language Specification; URL: http://download.microsoft.com/download/0/B/D/0BDA894F-2CCD-4C2C-B5A7-4EB1171962E5/CSharp%20Language%20Specixfication.doc

C# 4.0 The Complete Reference; Herbert Schildt; McGraw-Hill Osborne Media; 2010