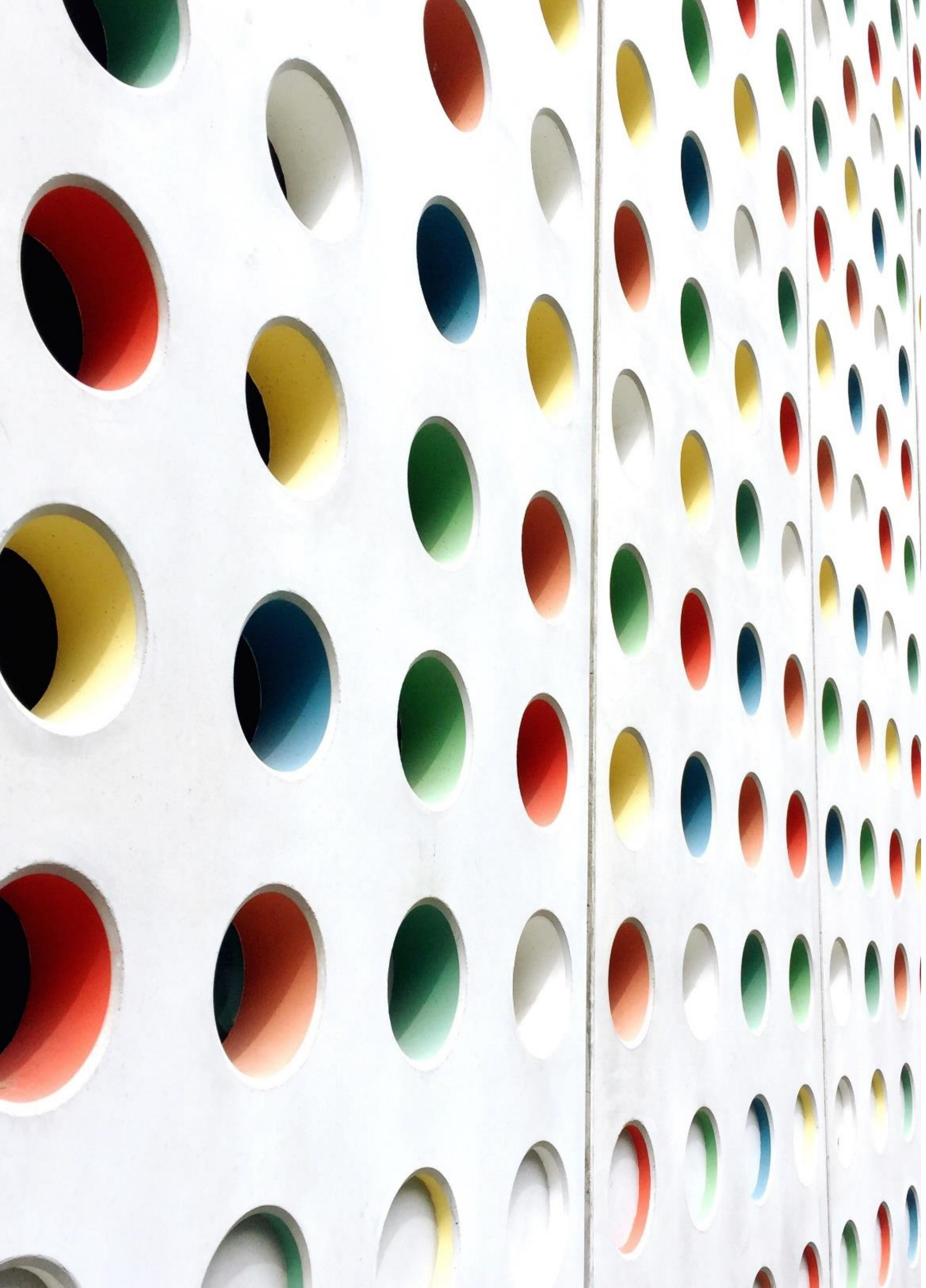


Chapter Three

Inheritance & Polymorphism

By: Sinodos G

Inheritance

- The process where one class acquires the properties (methods and fields) of another.
- Used to manage information in a hierarchical order.
- The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

extends Keyword

- **extends** is the keyword used to inherit the properties of a class.

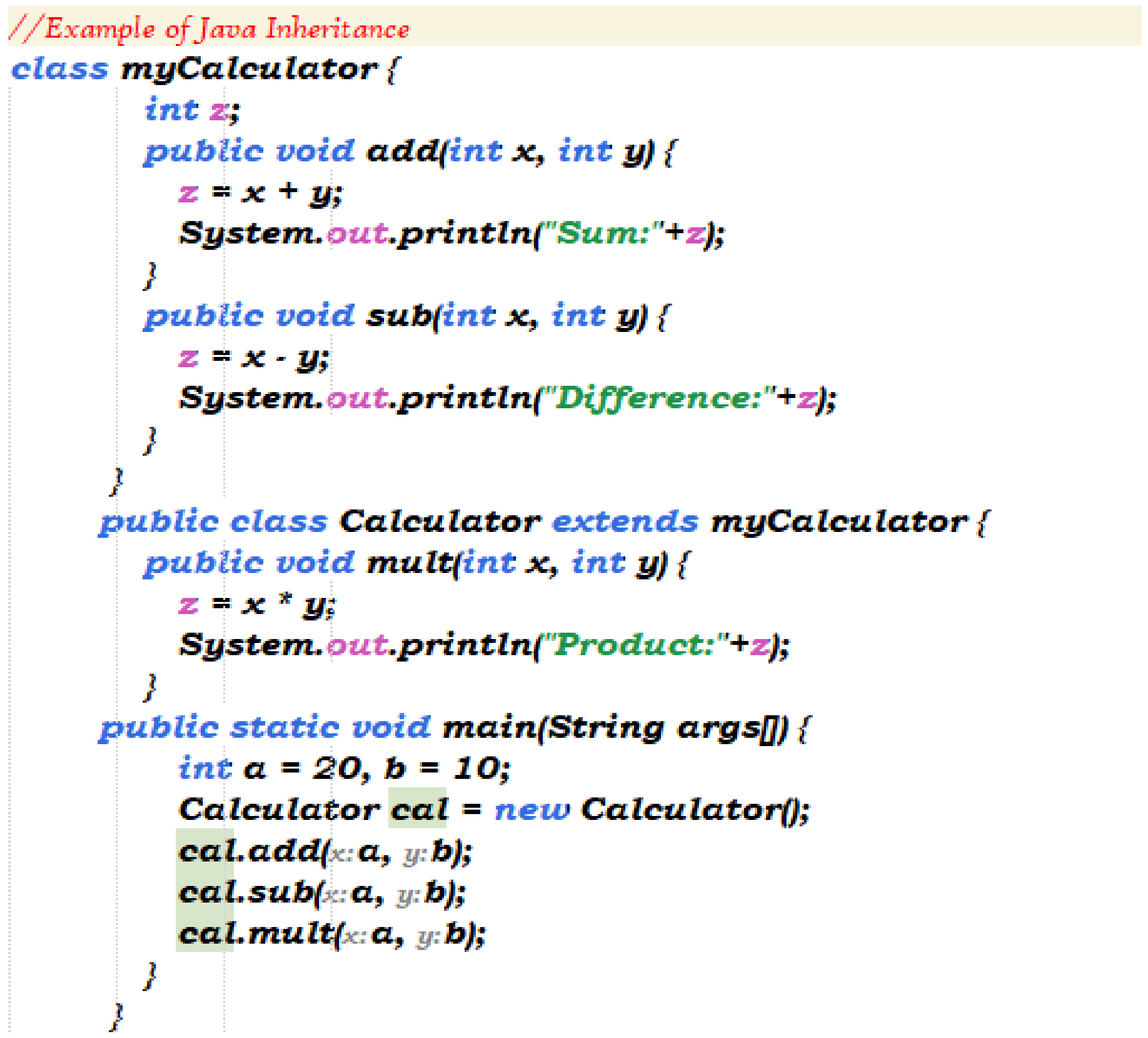
Syntax:

```
class Super {
        .....
        .....
    }
class Sub extends Super {
        .....
        .....
    }
```

Cont'd . . .

- It promotes code reusability and
- Allows a new class to extend [modify] functionality of an existing class.
  - Code Reusability
  - Subclass and Superclass Relationship
  - Extending Functionality
  - Single and Multiple Inheritance
  - Access Control
  - Method Overriding
  - Abstract Classes and Interfaces
  - Avoiding Code Duplication

Cont'd . . .



```
//Example of Java Inheritance
class myCalculator {
        int z;
        public void add(int x, int y) {
          z = x + y;
          System.out.println("Sum:"+z);
        }
        public void sub(int x, int y) {
          z = x - y;
          System.out.println("Difference:"+z);
        }
      }
      public class Calculator extends myCalculator {
        public void mult(int x, int y) {
          z = x * y;
          System.out.println("Product:"+z);
        }
      public static void main(String args[]) {
          int a = 20, b = 10;
          Calculator cal = new Calculator();
          cal.add(x:a, y:b);
          cal.sub(x:a, y:b);
          cal.mult(x:a, y:b);
        }
      }
```

- A subclass inherits all the members (fields, methods, and nested classes) from its superclass.

- Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

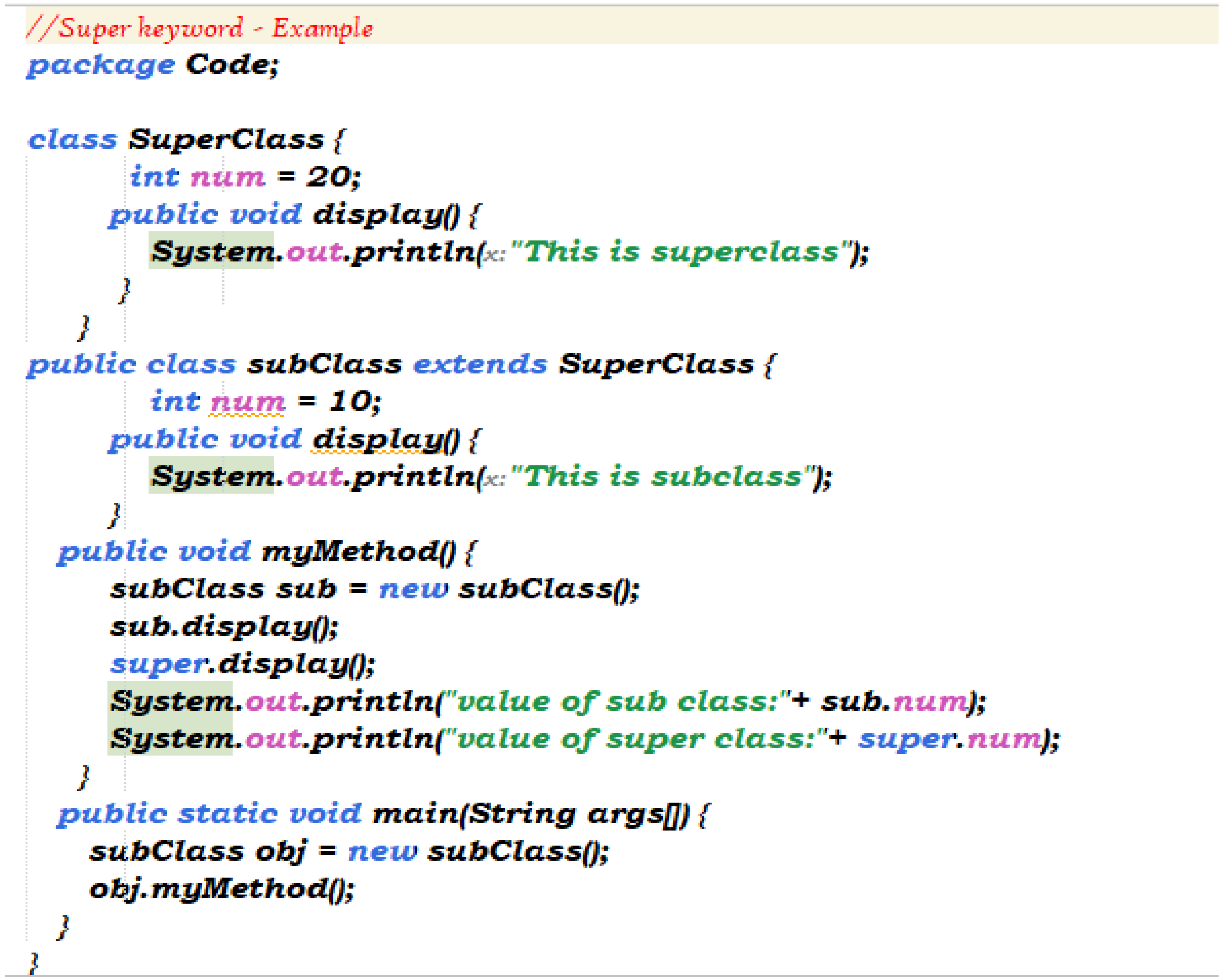
Cont'd . . .

- **The super keyword**
  - similar to **this** keyword. Used to:-
    - **differentiate the members** of superclass from the members of subclass, if they have same names.
    - **invoke the superclass** constructor from subclass.

Cont'd . . .

Differentiate the members

- If a class is inheriting the properties of another class.
- if the members of the superclass have the names same as the sub class, to differentiate these variables we use super keyword.
- ***Example***
  - super.variable;
  - super.method();



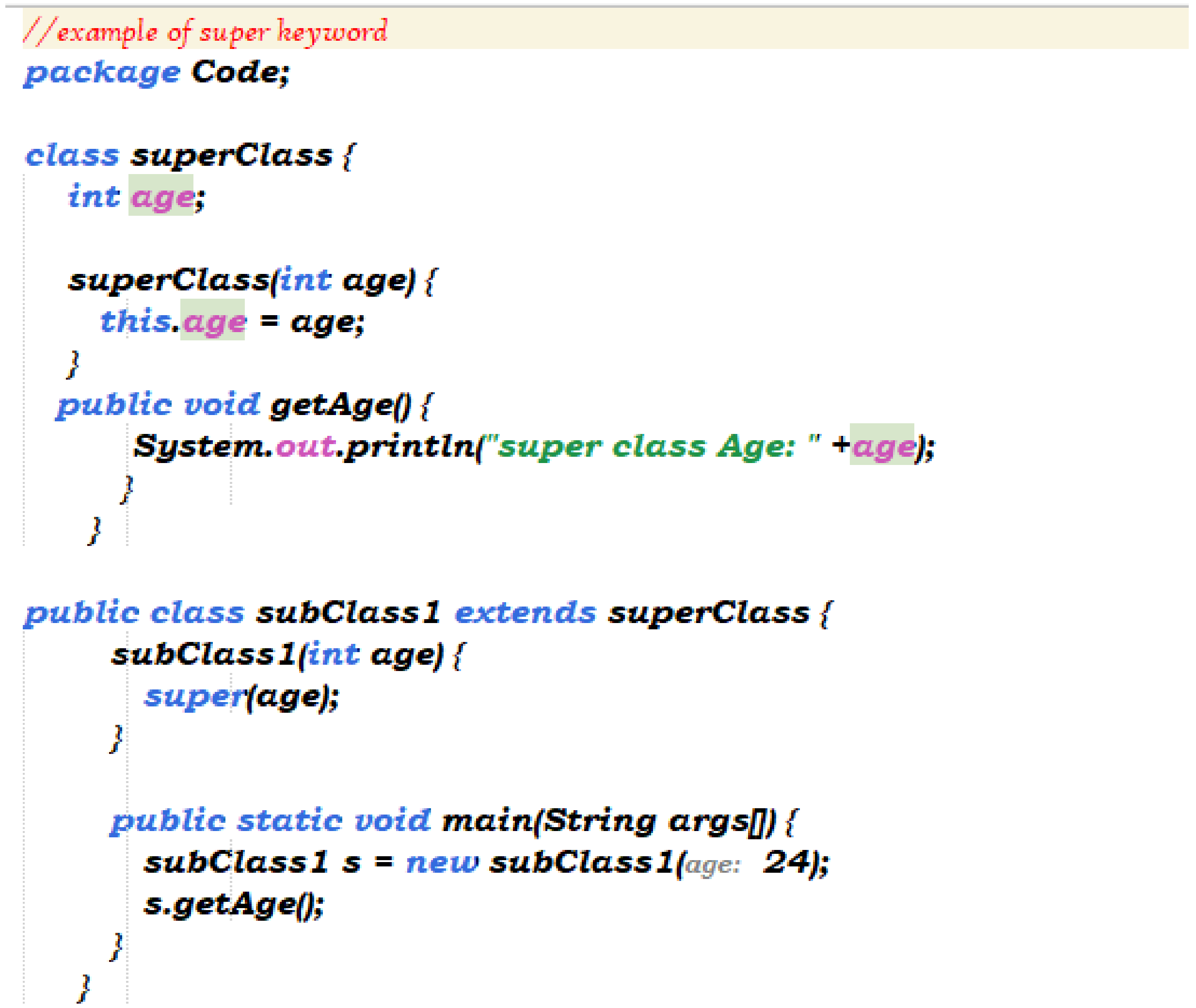
```
//Super keyword - Example
package Code;

class SuperClass {
        int num = 20;
      public void display() {
         System.out.println(x: "This is superclass");
      }
   }
public class subClass extends SuperClass {
         int num = 10;
      public void display() {
         System.out.println(x: "This is subclass");
      }
  public void myMethod() {
      subClass sub = new subClass();
      sub.display();
      super.display();
      System.out.println("value of sub class:"+ sub.num);
      System.out.println("value of super class:"+ super.num);
   }
  public static void main(String args[]) {
     subClass obj = new subClass();
     obj.myMethod();
  }
}
```

Cont'd . . .

- **Invoking Superclass Constructor**
  - If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the superclass.
  - But, to call a parameterized constructor of the superclass, you need to use the super keyword

Example:

super(values);



```java
//example of super keyword
package Code;

class superClass {
   int age;

   superClass(int age) {
     this.age = age;
   }
  public void getAge() {
        System.out.println("super class Age: " +age);
      }
    }

public class subClass1 extends superClass {
      subClass1(int age) {
        super(age);
      }

      public static void main(String args[]) {
        subClass1 s = new subClass1(age:  24);
        s.getAge();
      }
    }
```

IS-A Relationship

- IS-A is a way of saying: This object is a type of that object. Let us see how the **extends** keyword is used to achieve inheritance.

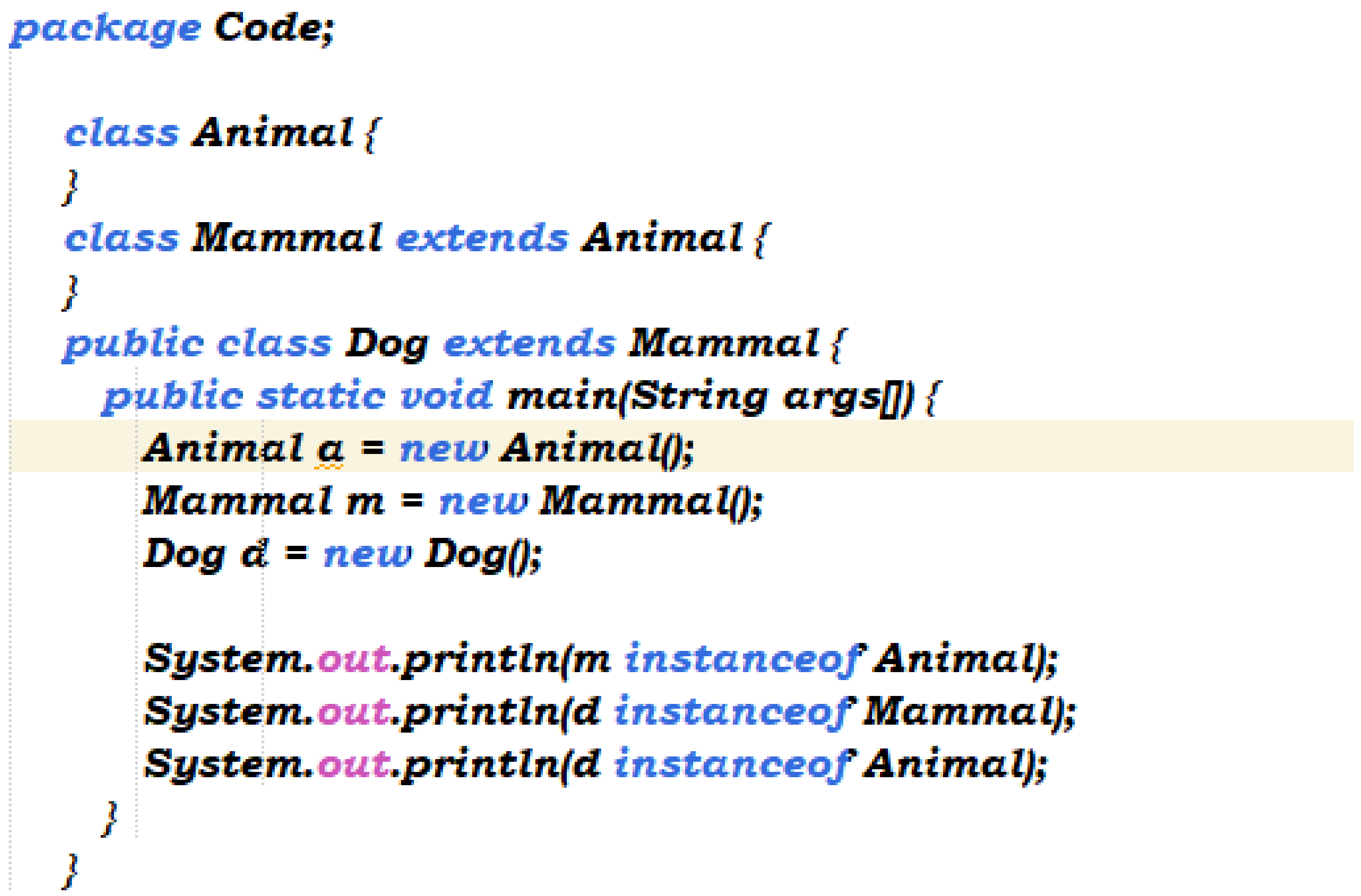
```
public class Animal {
- - -
}

public class Mammal extends Animal {
- - -
}

public class Reptile extends Animal {
- - -
}

public class Dog extends Mammal {
- - -
}
```

- Animal is the superclass of Mammal class.
- Animal is the superclass of Reptile class.
- Mammal and Reptile are subclasses of Animal class.
- Dog is the subclass of both Mammal and Animal classes.

the IS-A relationship, we can say −

Mammal IS-A Animal
Reptile IS-A Animal
Dog IS-A Mammal
Hence: Dog IS-A Animal as well

Cont’d . . .



```
package Code;

  class Animal {
  }
  class Mammal extends Animal {
  }
  public class Dog extends Mammal {
    public static void main(String args[]) {
      Animal a = new Animal();
      Mammal m = new Mammal();
      Dog d = new Dog();

      System.out.println(m instanceof Animal);
      System.out.println(d instanceof Mammal);
      System.out.println(d instanceof Animal);
    }
  }
```

Output

```
true
true
true
```

Cont'd . . .

implements keyword

- used with classes to inherit the properties of an interface.
- Interfaces can never be extended by a class.

```
public interface Animal {
}
public class Mammal implements Animal {
}
public class Dog extends Mammal {
}
```

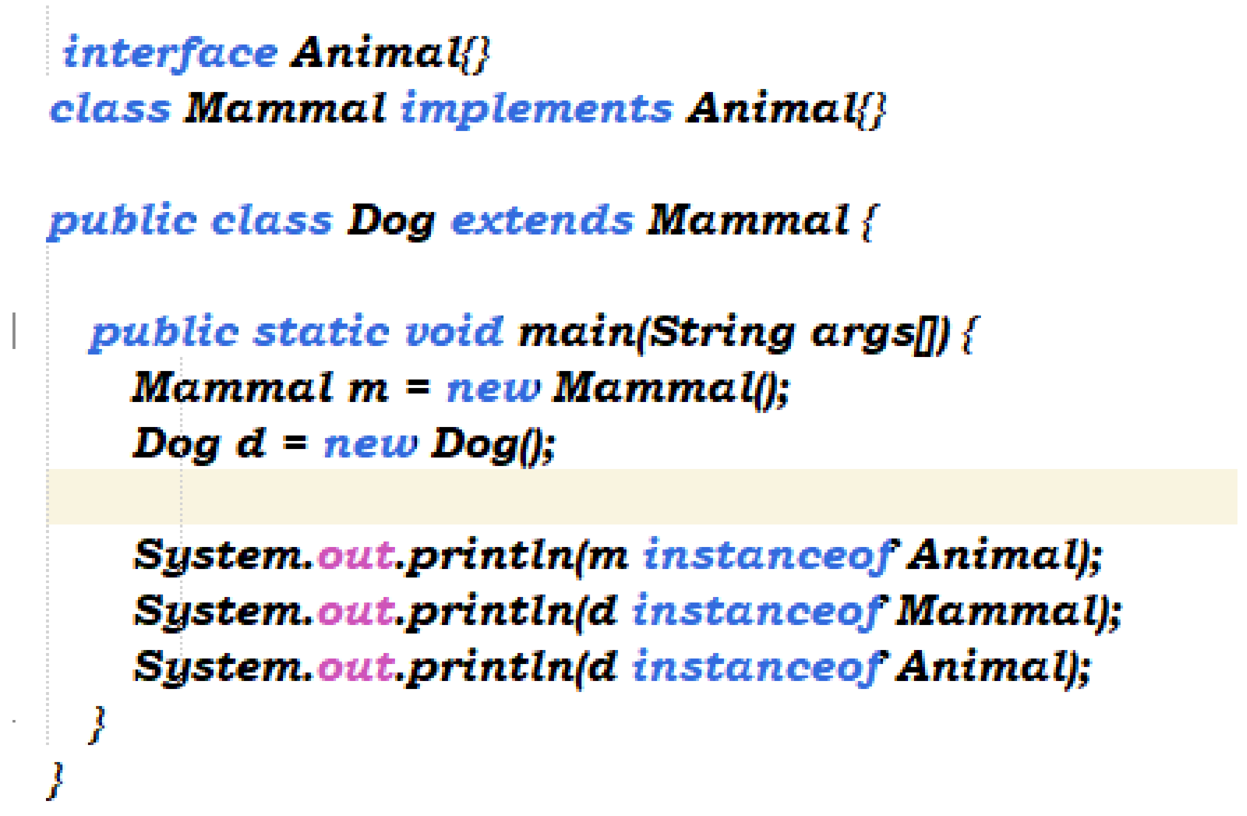
Instance Keyword

- Used to check and if a sub class is inherited from super class.

Example:

- it checks if Mammal is actually an Animal, and
- If dog is actually an Animal.

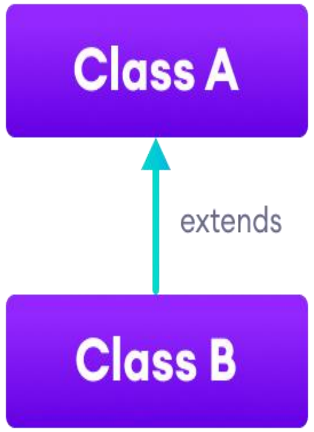
Cont'd . . .

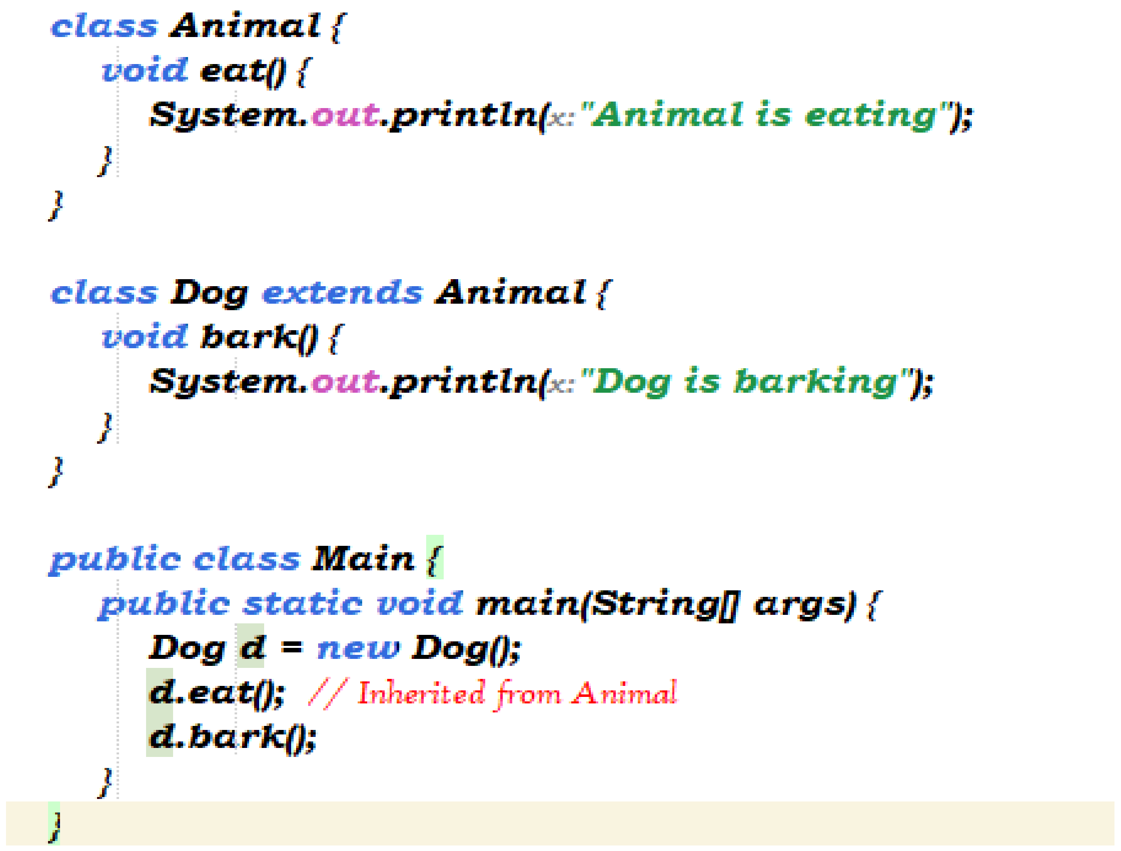


```
interface Animal{}
class Mammal implements Animal{}

public class Dog extends Mammal {

   public static void main(String args[]) {
      Mammal m = new Mammal();
      Dog d = new Dog();

      System.out.println(m instanceof Animal);
      System.out.println(d instanceof Mammal);
      System.out.println(d instanceof Animal);
   }
}
```

Types of Inheritance

- **Single Inheritance**
  - Occurs wen a class extends only one superclass



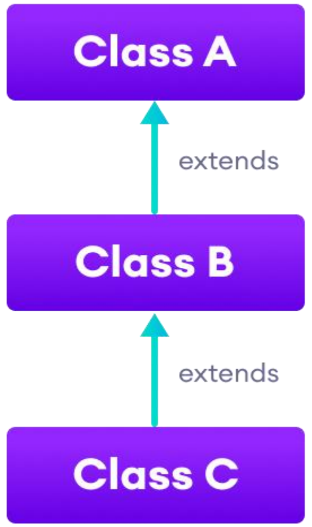


```
class Animal {
    void eat() {
        System.out.println(x: "Animal is eating");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println(x: "Dog is barking");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();  // Inherited from Animal
        d.bark();
    }
}
```

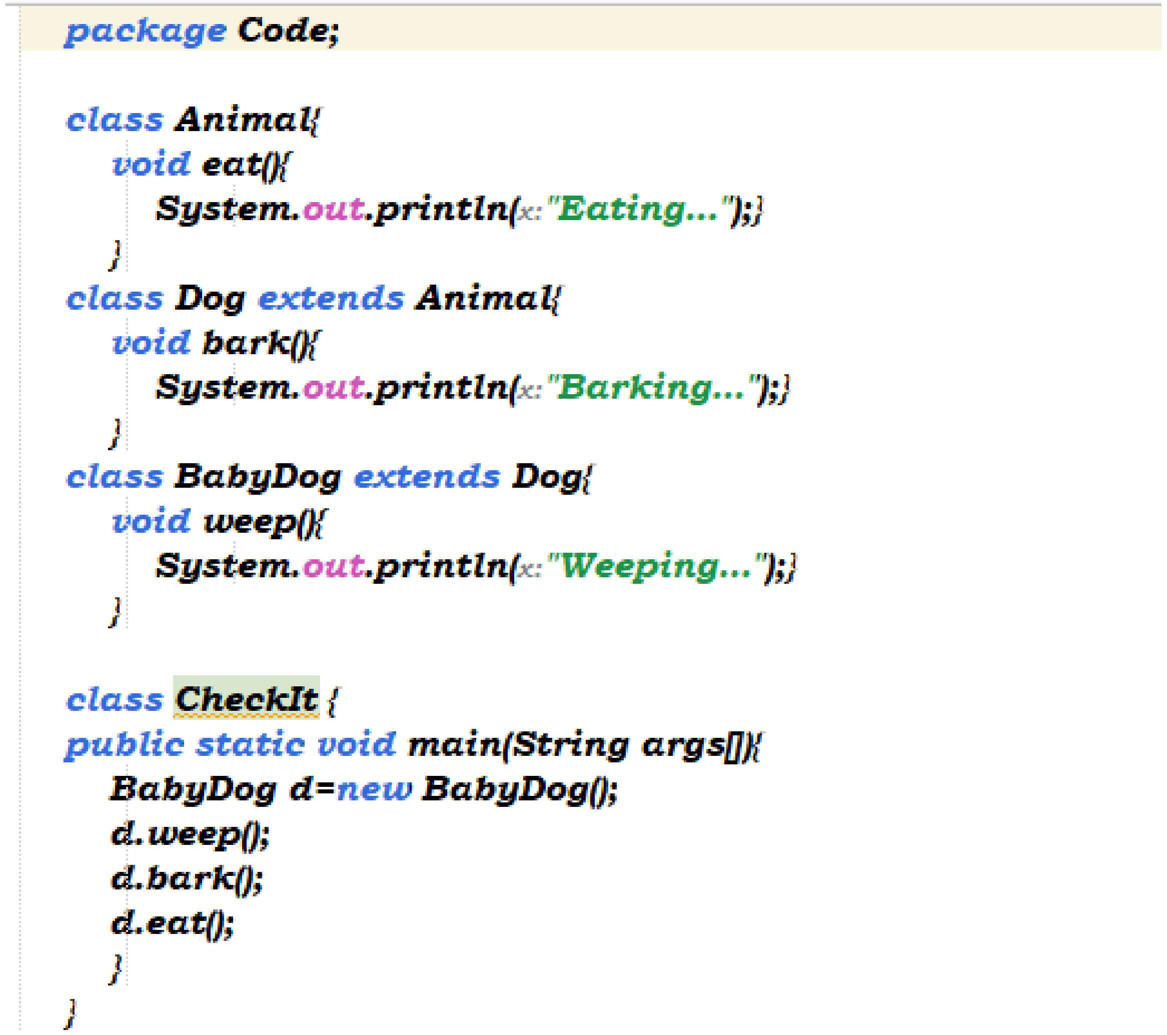
Cont'd . . .

- **Multilevel Inheritance**
  - a subclass extends from a superclass and then the same subclass acts as a superclass for another class.
  - It achieves multiple inheritance through interfaces
  - It allowing a class to implement multiple interfaces.

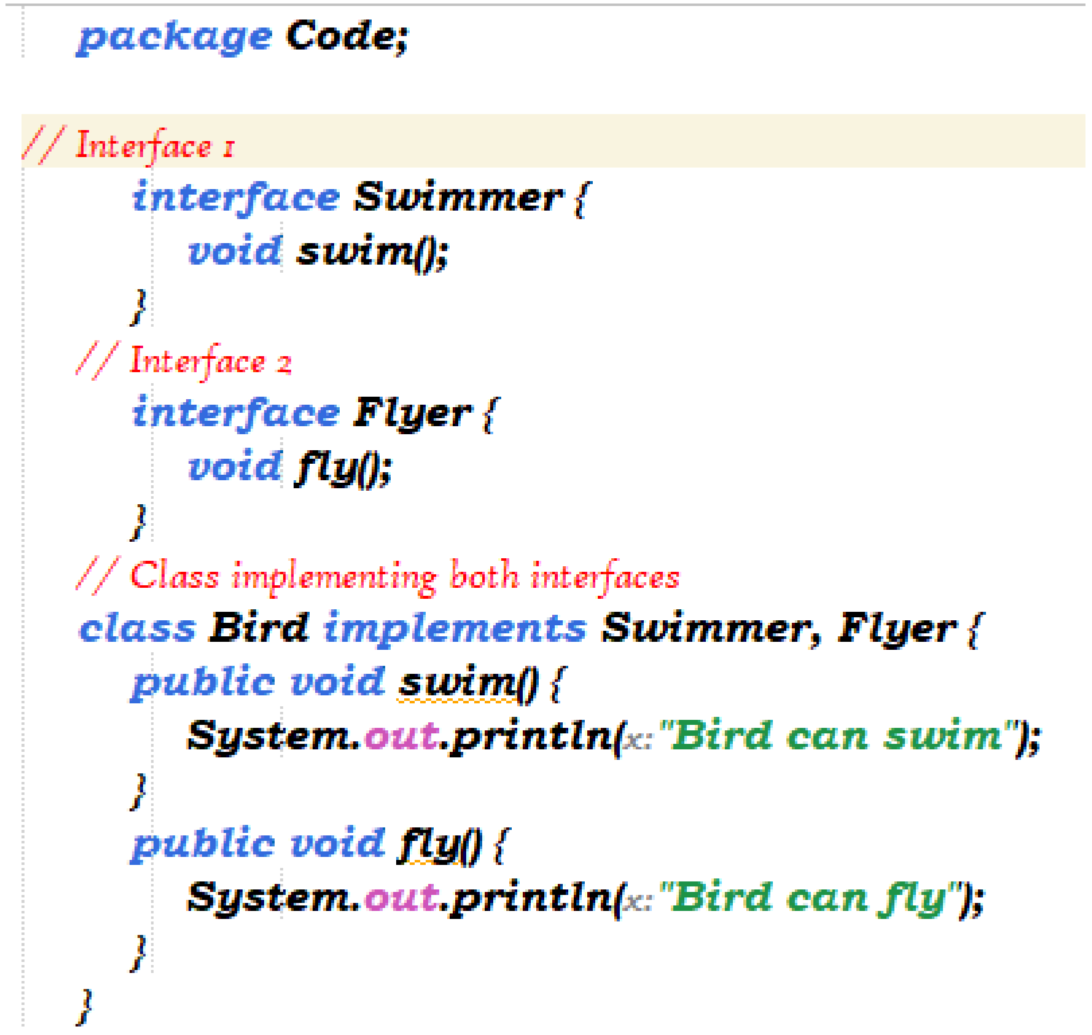
- ***Example:-***



Example:



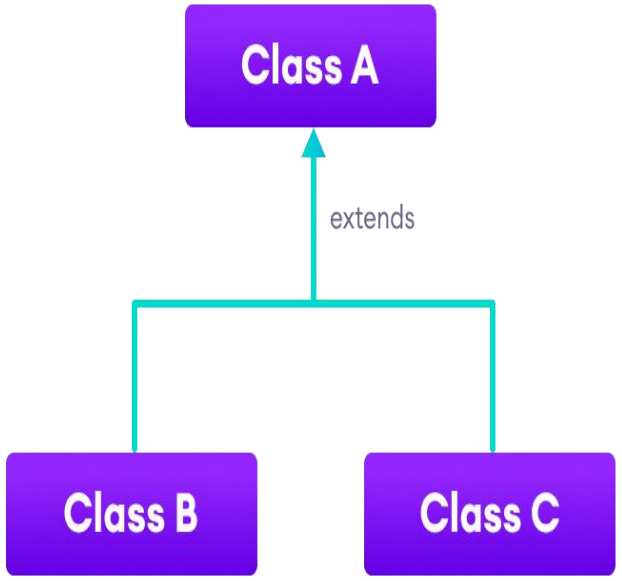
```
package Code;

class Animal{
    void eat(){
        System.out.println(x: "Eating...");}
    }
class Dog extends Animal{
    void bark(){
        System.out.println(x: "Barking...");}
    }
class BabyDog extends Dog{
    void weep(){
        System.out.println(x: "Weeping...");}
    }

class CheckIt {
public static void main(String args[]){
    BabyDog d=new BabyDog();
    d.weep();
    d.bark();
    d.eat();
    }
}
```

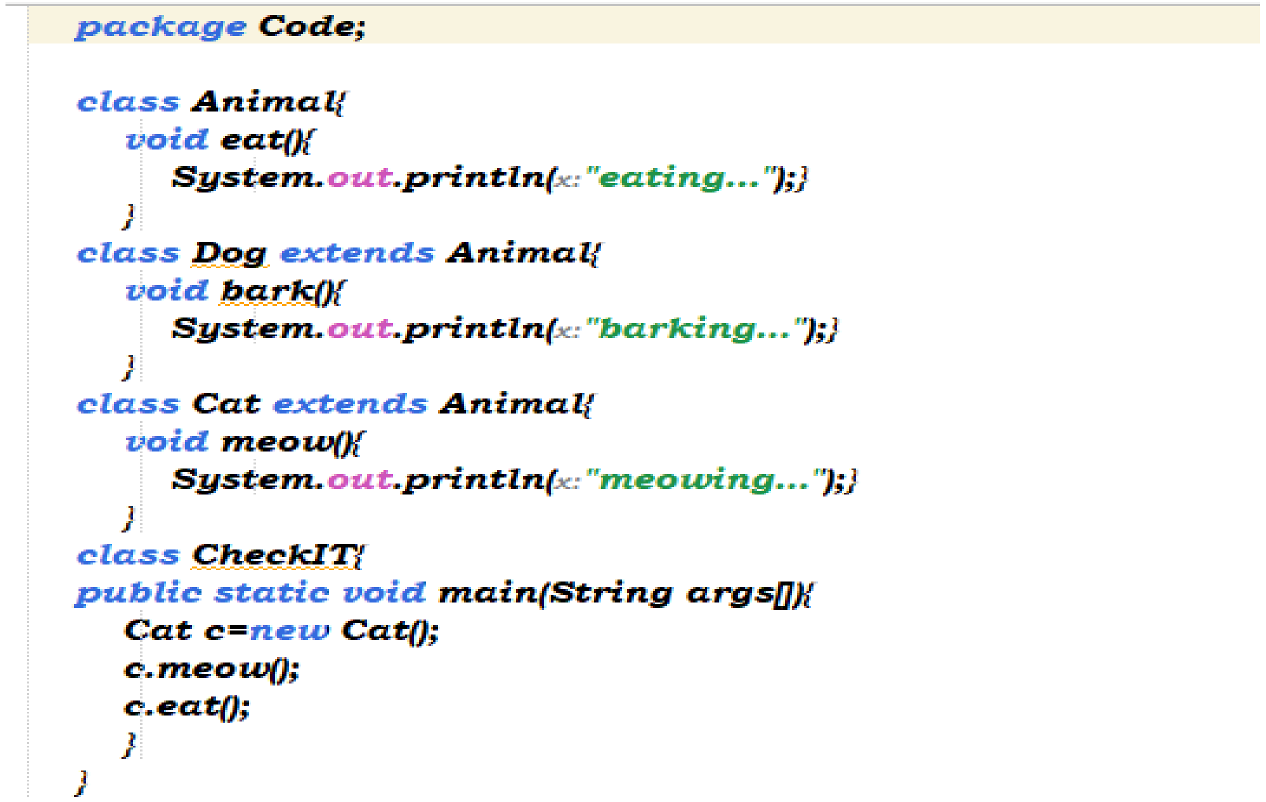


```
package Code;

// Interface 1
    interface Swimmer {
        void swim();
    }
// Interface 2
    interface Flyer {
        void fly();
    }
// Class implementing both interfaces
class Bird implements Swimmer, Flyer {
    public void swim() {
        System.out.println(x: "Bird can swim");
    }
    public void fly() {
        System.out.println(x: "Bird can fly");
    }
}
```

Hierarchical Inheritance

- When two or more classes inherits a single class, it is known as *hierarchical inheritance.*
- ***Example:*** Dog and Cat classes inherits the Animal class

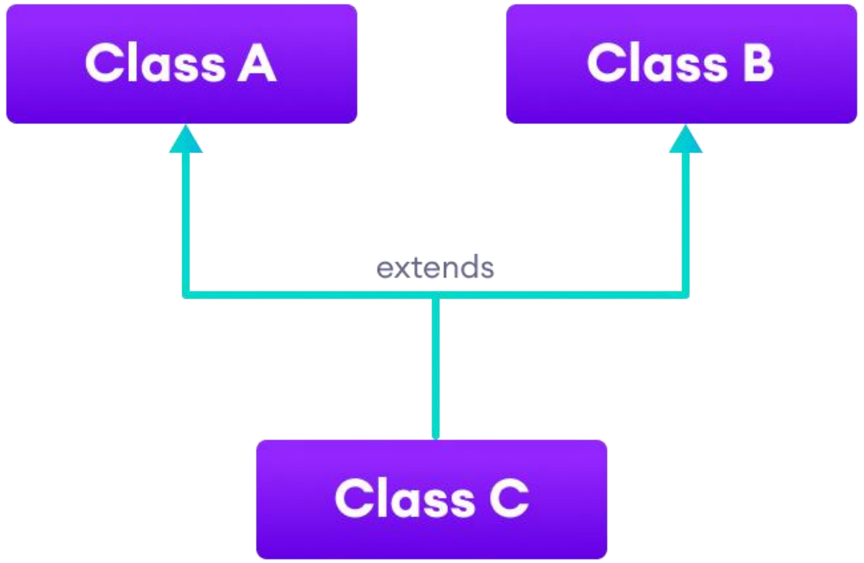




```
package Code;

class Animal{
    void eat(){
        System.out.println(x: "eating...");}
    }
class Dog extends Animal{
    void bark(){
        System.out.println(x: "barking...");}
    }
class Cat extends Animal{
    void meow(){
        System.out.println(x: "meowing...");}
    }
class CheckIT{
public static void main(String args[]){
    Cat c=new Cat();
    c.meow();
    c.eat();
    }
}
```

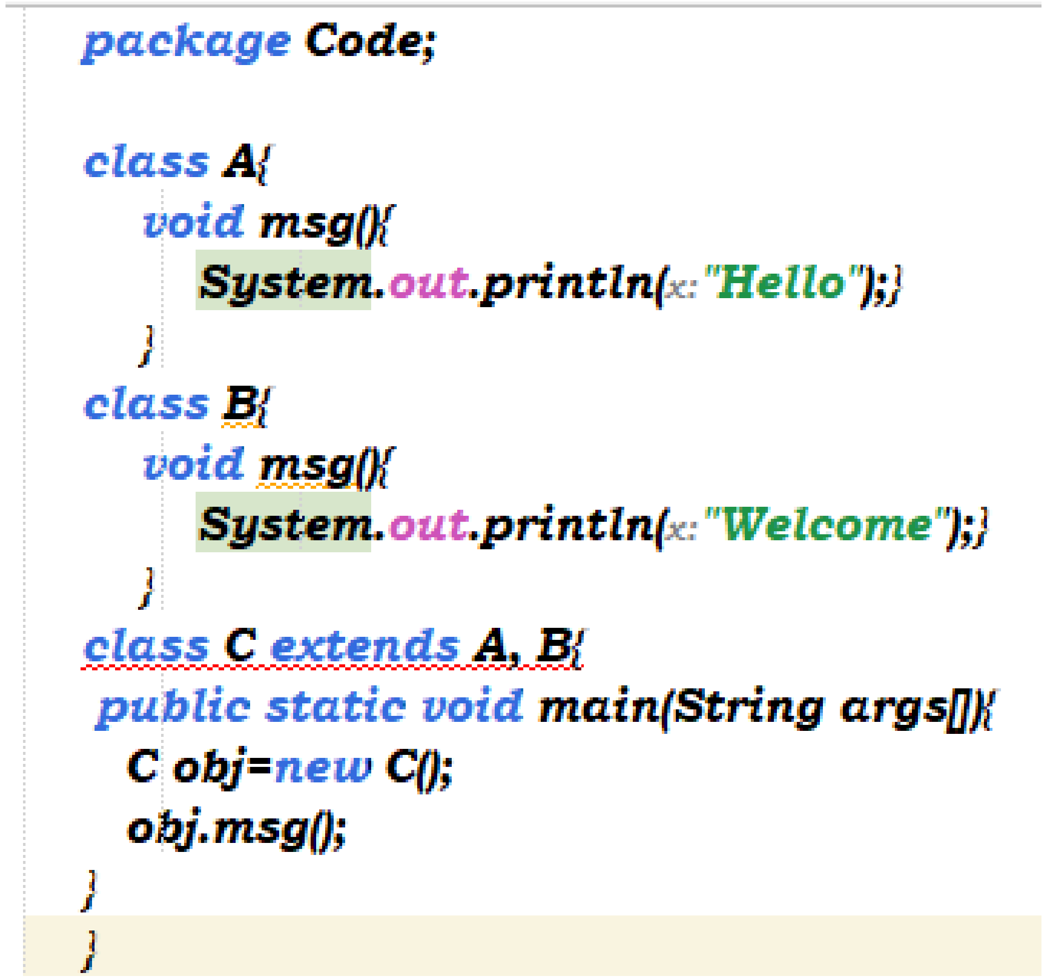
Cont'd . . .

- ***Multiple Inheritance***
  - Multiple inheritance is not supported in java.
  - Used to reduce the complexity and simplify the language
  - Example:
  - Consider ***A, B, and C*** are three classes.
    - The C class inherits A and B classes.
    - If A and B classes have the same method and you call it from child class object, there will be ***ambiguity*** to call the method of A or B class.



public class c extends A,B{}

Cont'd . . .



```
package Code;

class A{
    void msg(){
        System.out.println(x: "Hello");}
    }
class B{
    void msg(){
        System.out.println(x: "Welcome");}
    }
class C extends A, B{
 public static void main(String args[]){
   C obj=new C();
   obj.msg();
}
}
```

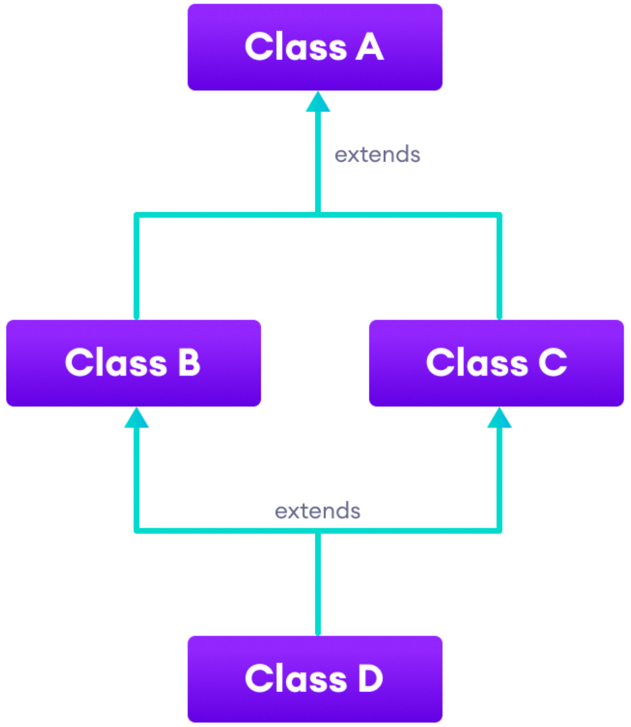
- Java doesn't support multiple inheritance.
- However, we can achieve multiple inheritance using interfaces.

Example:

class Language extends Frontend implements Backend {

Hybrid Inheritance

- Hybrid inheritance is a combination of two or more types of inheritance.



Example:

*public class **Child** extends **HumanBody** implements **Male, Female** {}*

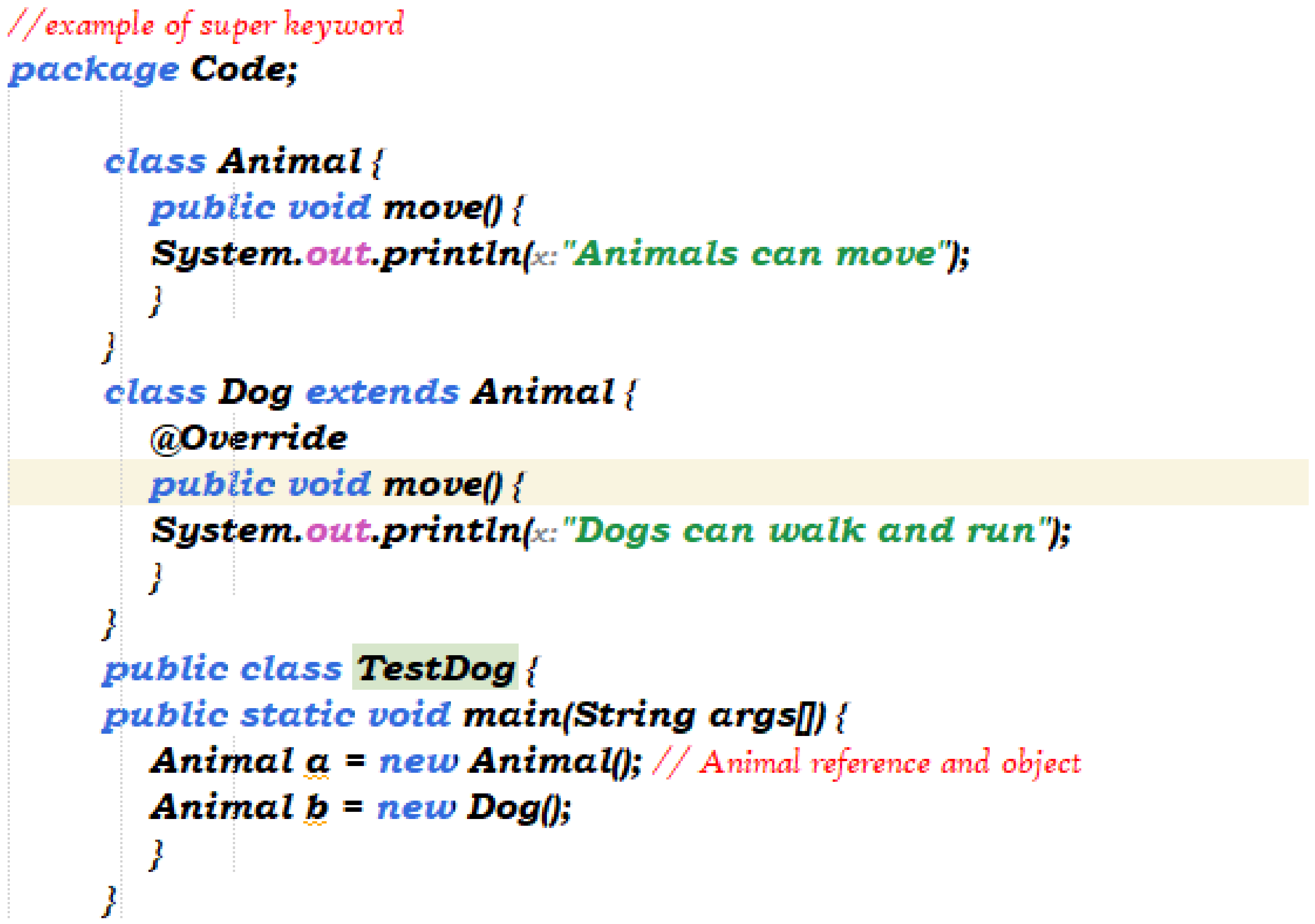
Overriding

Overriding

- means to override the functionality of an existing method.
- If a class inherits a method from its superclass, then there is a chance to override the method provided that it is not marked final.

- ***Benefit of overriding***
  - ability to define a behavior that's specific to the subclass type, which means a subclass can implement a parent class method based on its requirement.

Cont’d . . .



```
//example of super keyword
package Code;

    class Animal {
        public void move() {
        System.out.println(x:"Animals can move");
        }
    }
    class Dog extends Animal {
        @Override
        public void move() {
        System.out.println(x:"Dogs can walk and run");
        }
    }
    public class TestDog {
    public static void main(String args[]) {
        Animal a = new Animal(); // Animal reference and object
        Animal b = new Dog();
        }
    }
```

Rules for Method Overriding

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
- The access level cannot be more restrictive than the overridden method's access level.
  - For example: If the superclass method is declared public then the overridding method in the sub class cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited, then it cannot be overridden.

Cont'd . . .

- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.
- An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method.
- The overriding method can throw narrower or fewer exceptions than the overridden method.
- Constructors cannot be overridden.

Cont'd . . .

super keyword

- When invoking a superclass version of an overridden method the super keyword is used.

```
class Animal {
      public void move() {
         System.out.println("Animals can move");
         }
      }
class Dog extends Animal {
      public void move() {
      super.move(); // invokes the super class method
      System.out.println("Dogs can walk and run");
   }
}
```

```
public class TestDog {
public static void main(String args[]) {
 Animal b = new Dog();
// Animal reference but Dog object
b.move();
// runs the method in Dog class
} }
```

Cont'd . . .

Polymorphism

- the ability of an object to take on many forms.
- occurs when a parent class reference is used to refer to a child class object.
- all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.
- Only possible to access an object through a reference variable.
- A reference variable
  - Once declared cannot be changed.
  - Can reassigned to other objects provided that it is not declared final.
  - A reference variable can be declared as a class or interface type.

Cont'd . . .

Example:

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

- ***Following are true for the above examples***
  - A Deer IS-A Animal
  - A Deer IS-A Vegetarian
  - A Deer IS-A Deer
  - A Deer IS-A Object
- Now, the Deer class is considered to be polymorphic since this has multiple inheritance.

```
Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;
```

- All the reference variables ***d, a, v, o*** refer to the same Deer object in the heap.

Cont'd . . .

- There are two types of polymorphism in Java:
  - compile-time (static) polymorphism
  - runtime (or dynamic) polymorphism.
- **Compile-time Polymorphism:**
  - Method Overloading
  - occurs when two or more methods in the same class have the same name but different parameters.
  - The compiler determines which method to invoke based on the method signature.

Example

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }
}
```

Cont'd . . .

- ***Runtime Polymorphism***
  - Method Overriding
  - occurs when a subclass provides a specific implementation for a method that is already defined in its superclass

Example:

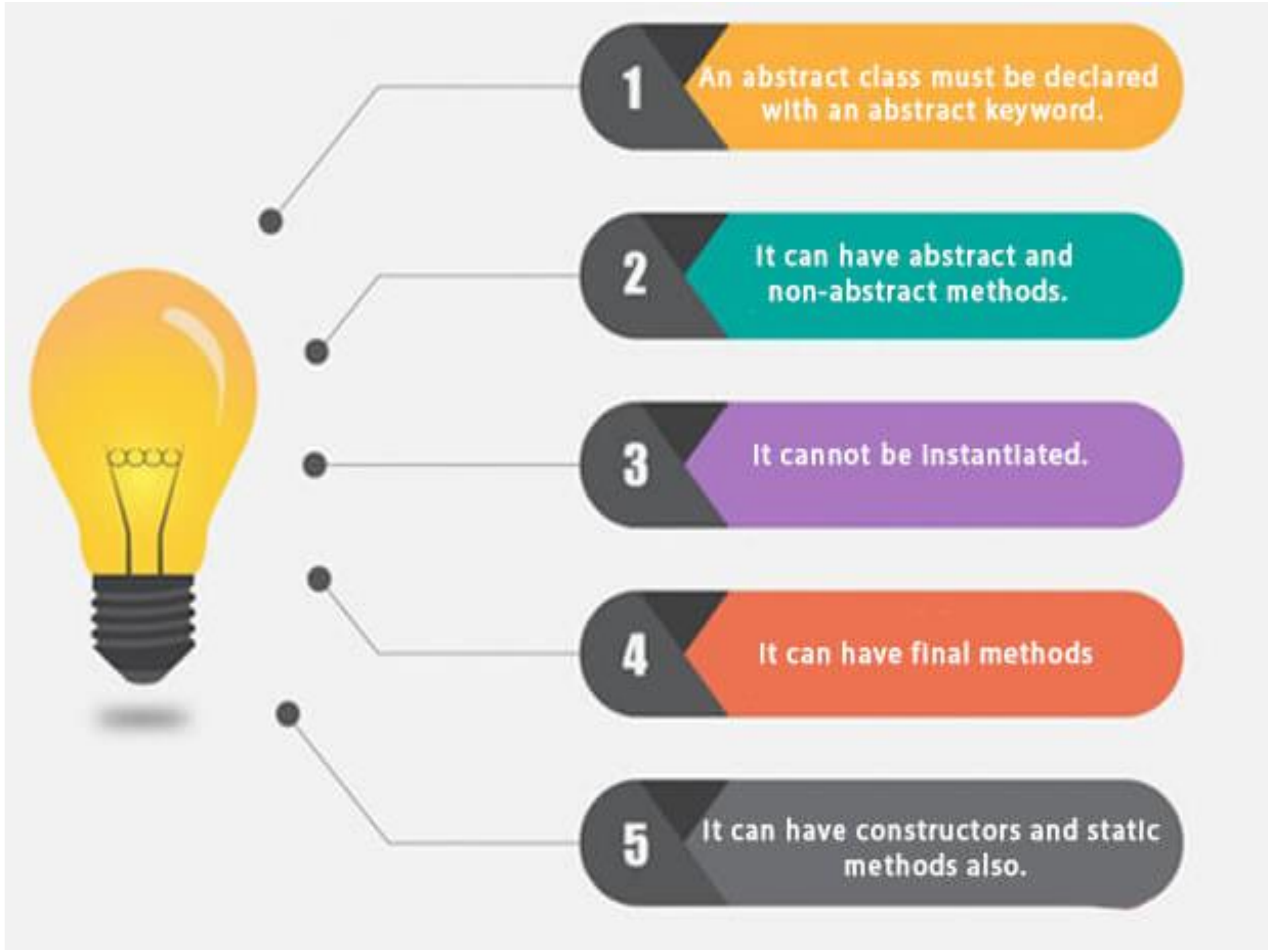
```
cass Animal {
    void makeSound() {
        System.out.println("Generic animal sound");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Bark! Bark!");
    }
}
```

Abstraction

- The process of hiding certain details and showing only essential information to the user.
- a process of hiding the implementation details and showing only functionality to the user.
- ***Example:-***
  - Sending SMS where you type the text and send the message.
  - You don't know the internal processing about the message delivery.
- Abstraction - focus on what the object does instead of how it does it.
- It can have abstract and non-abstract methods ***(method with the body)***.
  - ***Example:-*** public void get();
- There are two ways to achieve abstraction in java
  - **Abstract classes and**
  - **Interfaces.**

Abstract Class

- A class which is declared with the **abstract keyword** is known as an abstract class
- It can have abstract and non-abstract methods.
- It needs to be extended and its method implemented. It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.


1
An abstract class must be declared with an abstract keyword.
2
It can have abstract and non-abstract methods.
3
It cannot be instantiated.
4
It can have final methods
5
It can have constructors and static methods also.

Cont'd . . .

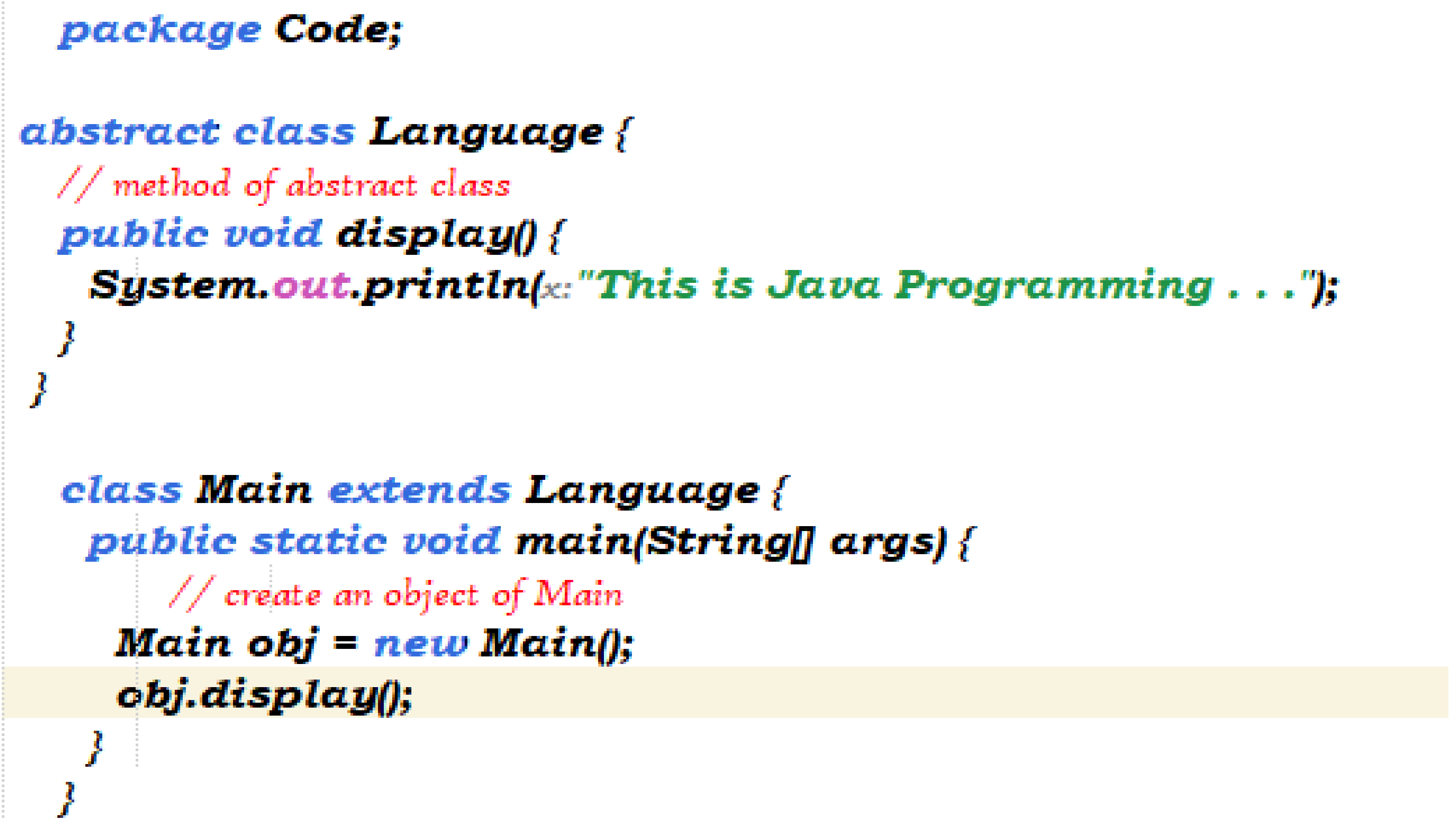
- If a class has at least one abstract method, then the class must be declared abstract.
  - If a class is declared abstract, it cannot be instantiated.
- To use an abstract class:-
  - you have to inherit it from another class, provide implementations to the abstract methods in it.
  - If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.
- Abstraction promotes:-
  - code reusability,
  - maintainability, and
  - flexibility in software development.

Syntax:

public abstract class ***Employee*** *{ }*

Cont’d . . .

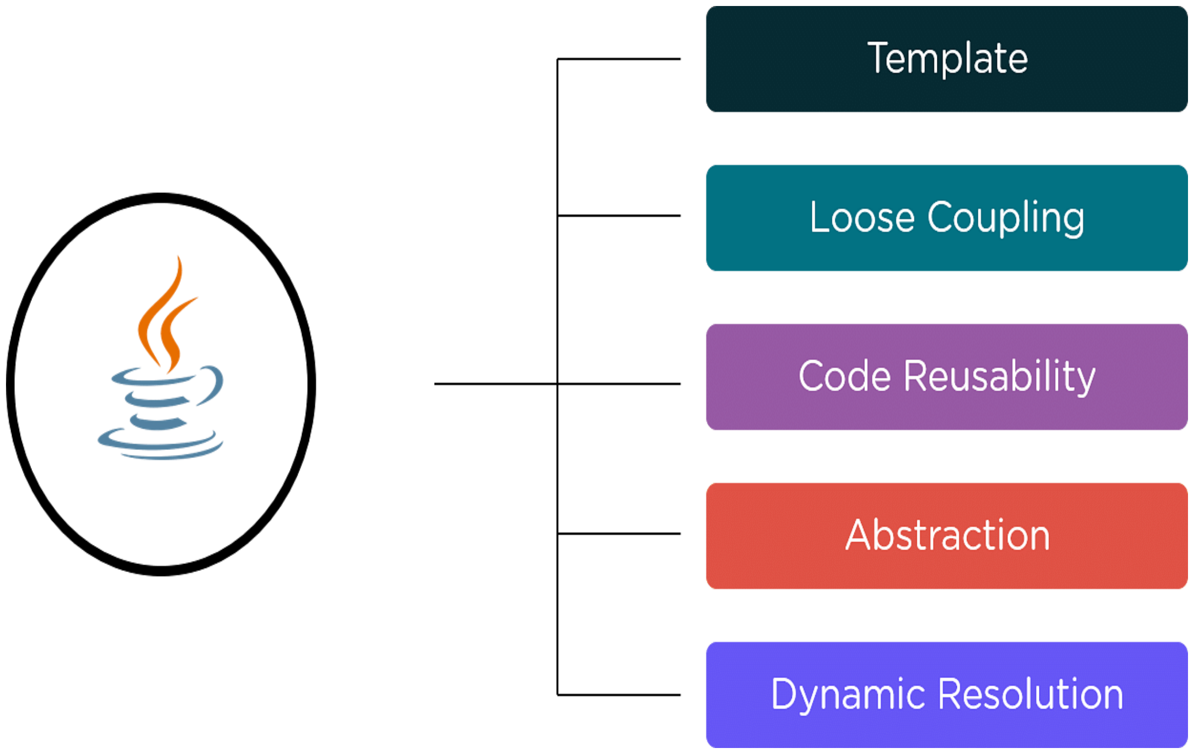
- **Example:**



```
  package Code;

abstract class Language {
  // method of abstract class
  public void display() {
    System.out.println(x: "This is Java Programming . . .");
  }
 }

  class Main extends Language {
    public static void main(String[] args) {
        // create an object of Main
      Main obj = new Main();
      obj.display();
    }
  }
```

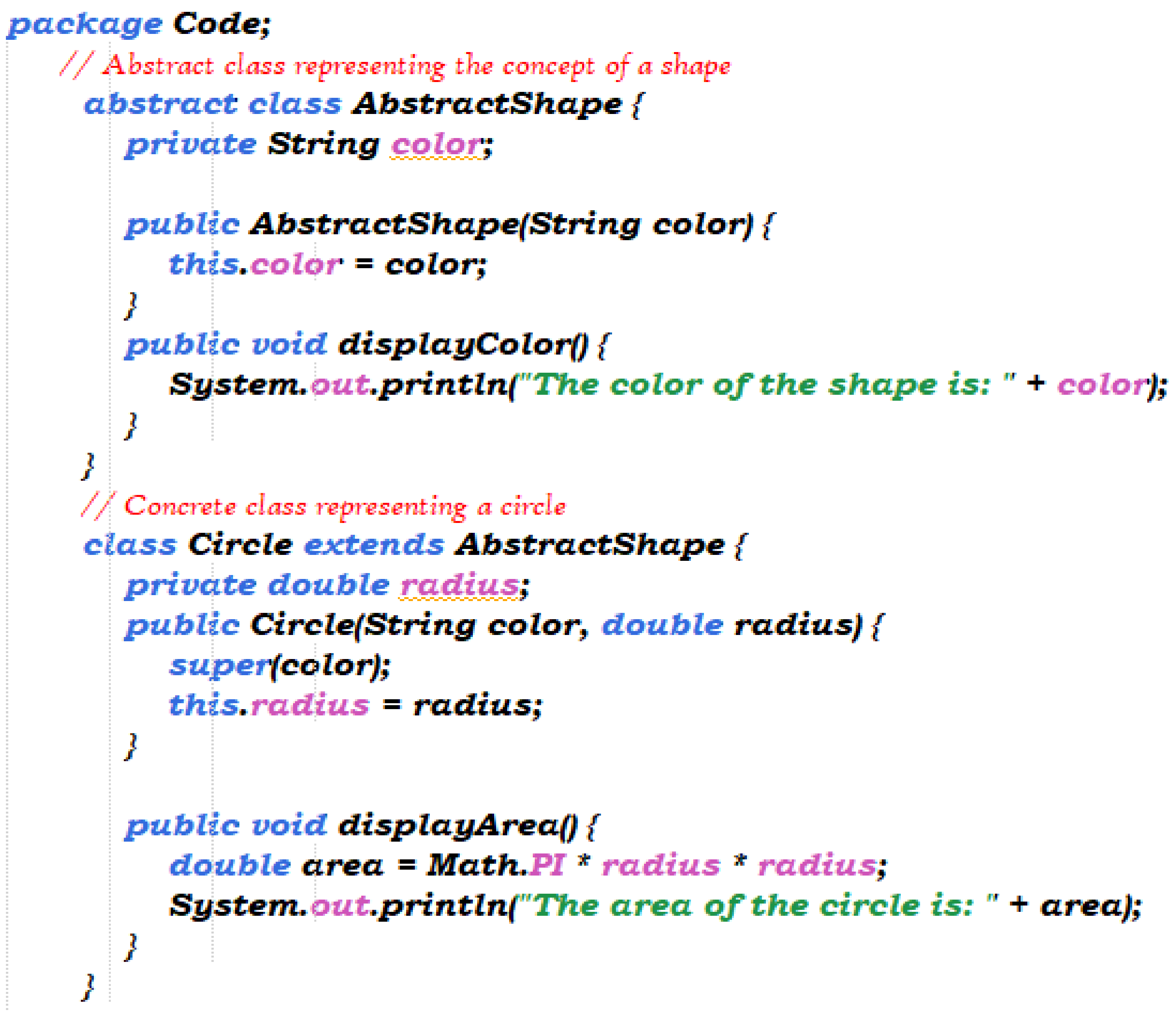
Cont'd . . .

Features of Abstract Class

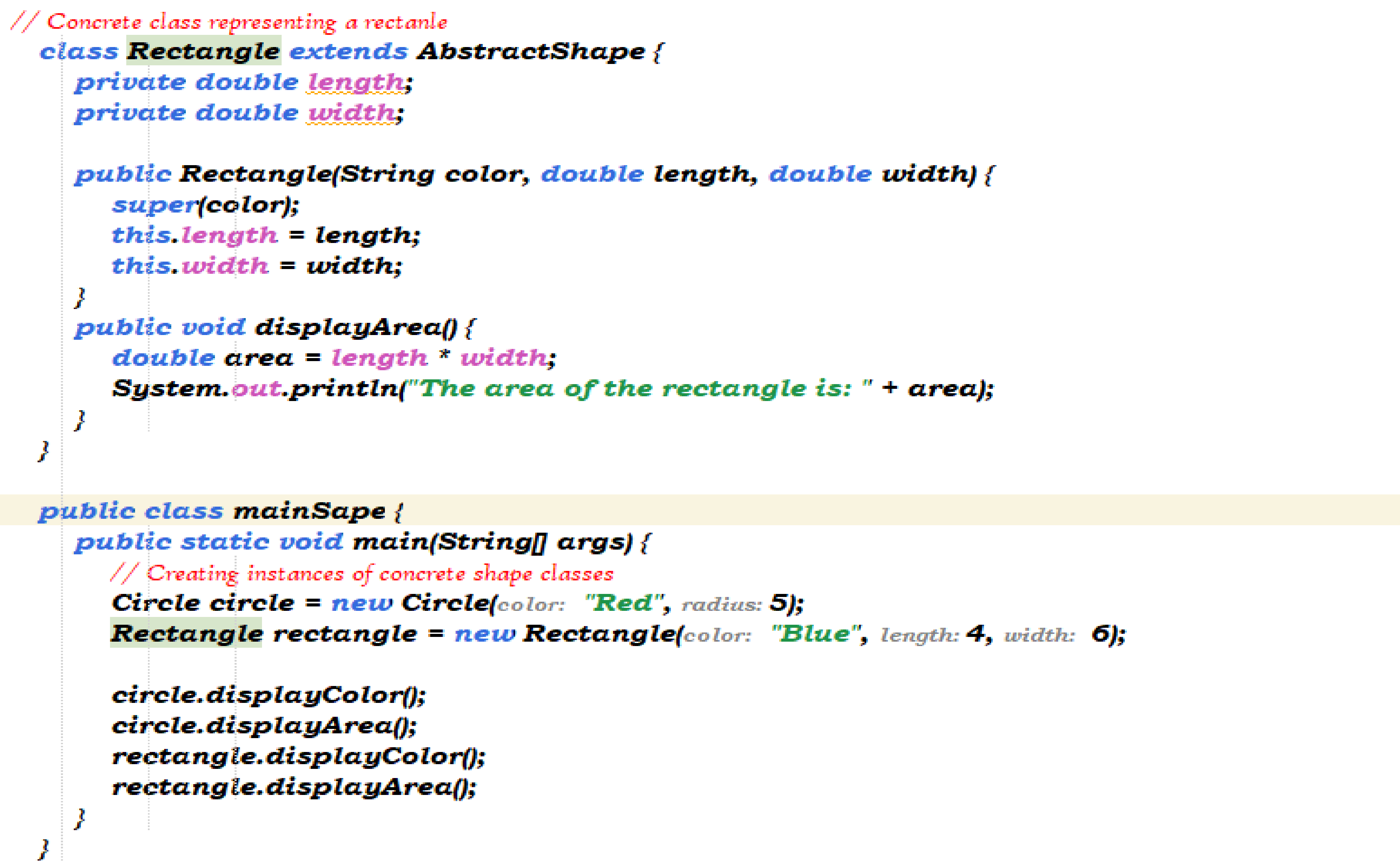


- ***Template:*** presents the end-user with a template that explains the methods involved.
- **Loose Coupling:** Data abstraction in Java enables loose coupling, by reducing the dependencies at an exponential level.
- **Code Reusability:** avoids the process of writing the same code again.
- **Abstraction:** helps the developers hide the code complications from the end-user
- **Dynamic Resolution:** Using the support of dynamic method resolution, developers can solve multiple problems

Example 2



```
package Code;
  // Abstract class representing the concept of a shape
    abstract class AbstractShape {
      private String color;

      public AbstractShape(String color) {
          this.color = color;
      }
      public void displayColor() {
          System.out.println("The color of the shape is: " + color);
      }
    }
  // Concrete class representing a circle
    class Circle extends AbstractShape {
      private double radius;
      public Circle(String color, double radius) {
          super(color);
          this.radius = radius;
      }

      public void displayArea() {
          double area = Math.PI * radius * radius;
          System.out.println("The area of the circle is: " + area);
      }
    }
```



```
// Concrete class representing a rectanle
class Rectangle extends AbstractShape {
    private double length;
    private double width;

    public Rectangle(String color, double length, double width) {
        super(color);
        this.length = length;
        this.width = width;
    }
    public void displayArea() {
        double area = length * width;
        System.out.println("The area of the rectangle is: " + area);
    }
}

public class mainSape {
    public static void main(String[] args) {
        // Creating instances of concrete shape classes
        Circle circle = new Circle(color: "Red", radius: 5);
        Rectangle rectangle = new Rectangle(color: "Blue", length: 4, width: 6);

        circle.displayColor();
        circle.displayArea();
        rectangle.displayColor();
        rectangle.displayArea();
    }
}
```

Abstract Methods

- A method which is declared as abstract and does not have implementation is known as an abstract method.
- can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).
- Used If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as an abstract.
  - abstract keyword is used to declare the method as abstract.
  - An abstract method contains a method signature, but no method body.
  - Instead of curly braces, an abstract method will have a semoicolon (;)

Cont'd . . .

- An abstract class can have both abstract and regular methods:

```
abstract class Animal {
  public abstract void animalSound();
  public void sleep() {
    System.out.println("kjfkdfjdfjklfjZzz");
  }
}
```

It is not possible to create an object of the Animal class:

Example:

```
Animal abj = new Animal(); // will generate an error
```

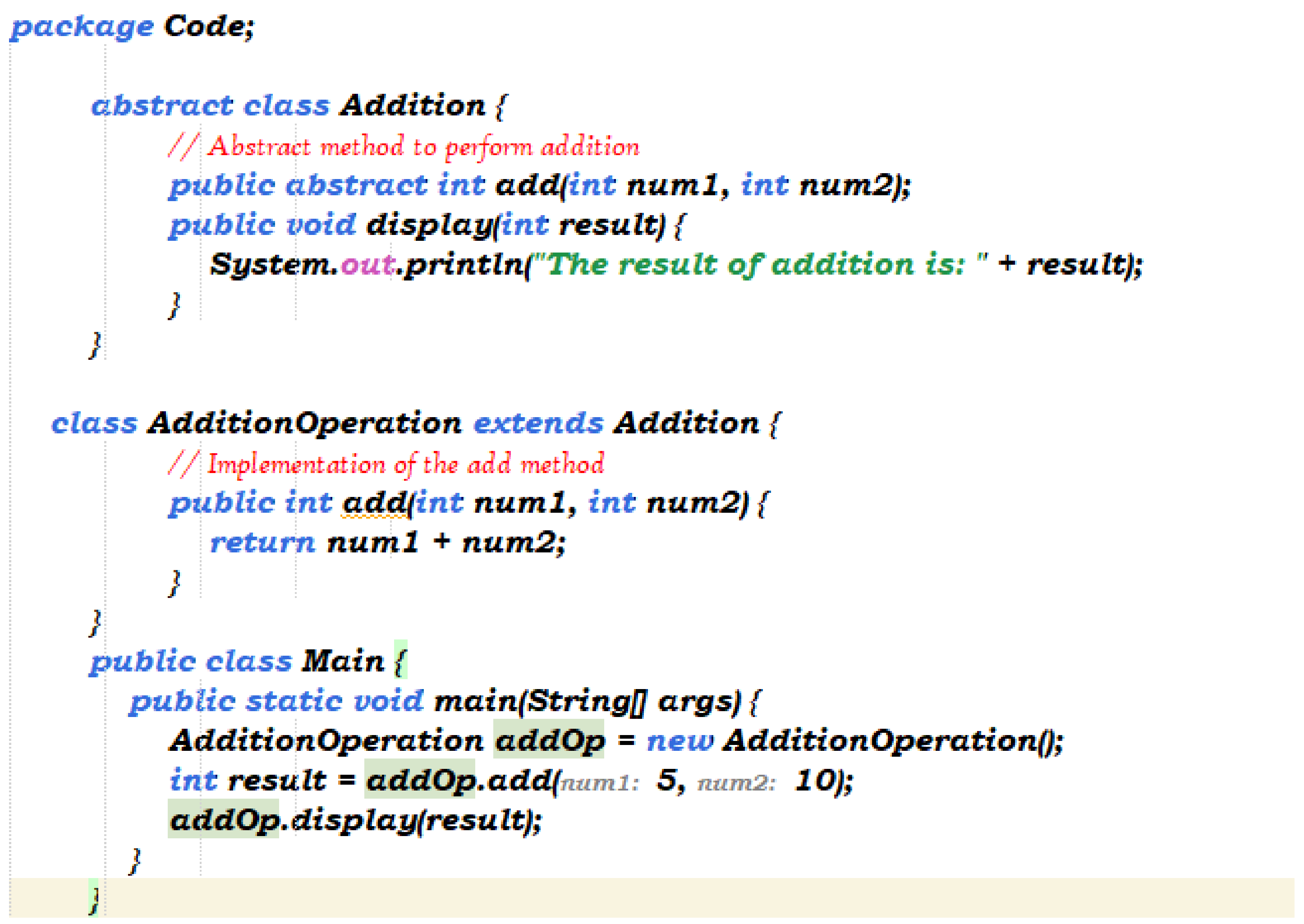
- To access the abstract class, it must be inherited from another class.

Example:

```
// Abstract class
abstract class Animal {
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep() {
    System.out.println("ZzzQqq");
  }
}

  // Subclass (inherit from Animal)
  class Pig extends Animal {
    public void animalSound() {
      // The body of animalSound() is provided here
      System.out.println("The pig says: wee wee");
    }
  }
```

```
class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig();
// Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Example2



```
package Code;

    abstract class Addition {
        // Abstract method to perform addition
        public abstract int add(int num1, int num2);
        public void display(int result) {
            System.out.println("The result of addition is: " + result);
        }
    }

  class AdditionOperation extends Addition {
        // Implementation of the add method
        public int add(int num1, int num2) {
            return num1 + num2;
        }
    }
    public class Main {
      public static void main(String[] args) {
        AdditionOperation addOp = new AdditionOperation();
        int result = addOp.add(num1: 5, num2: 10);
        addOp.display(result);
      }
    }
```

Interface

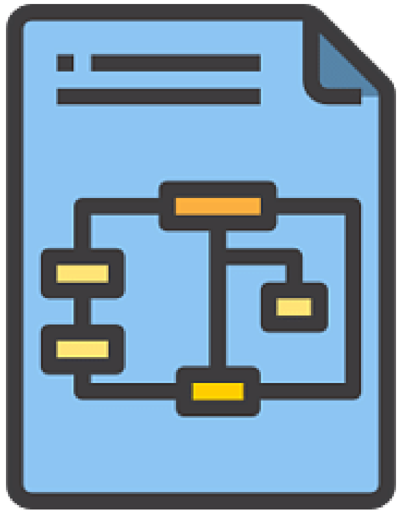
- a boundary between the method and the class implementing it.
- holds the method signature in it, but never the implementation.
- It is a fully abstract class and used to achieve abstraction.
- It includes a group of abstract methods (methods without a body).
- We use the interface keyword to create an interface

//Syntax:

```
interface <Class_Name>{
//Method_Signatures;
}
```

Example:

```
interface Language {
    public void getType();
    public void getVersion();
}
```



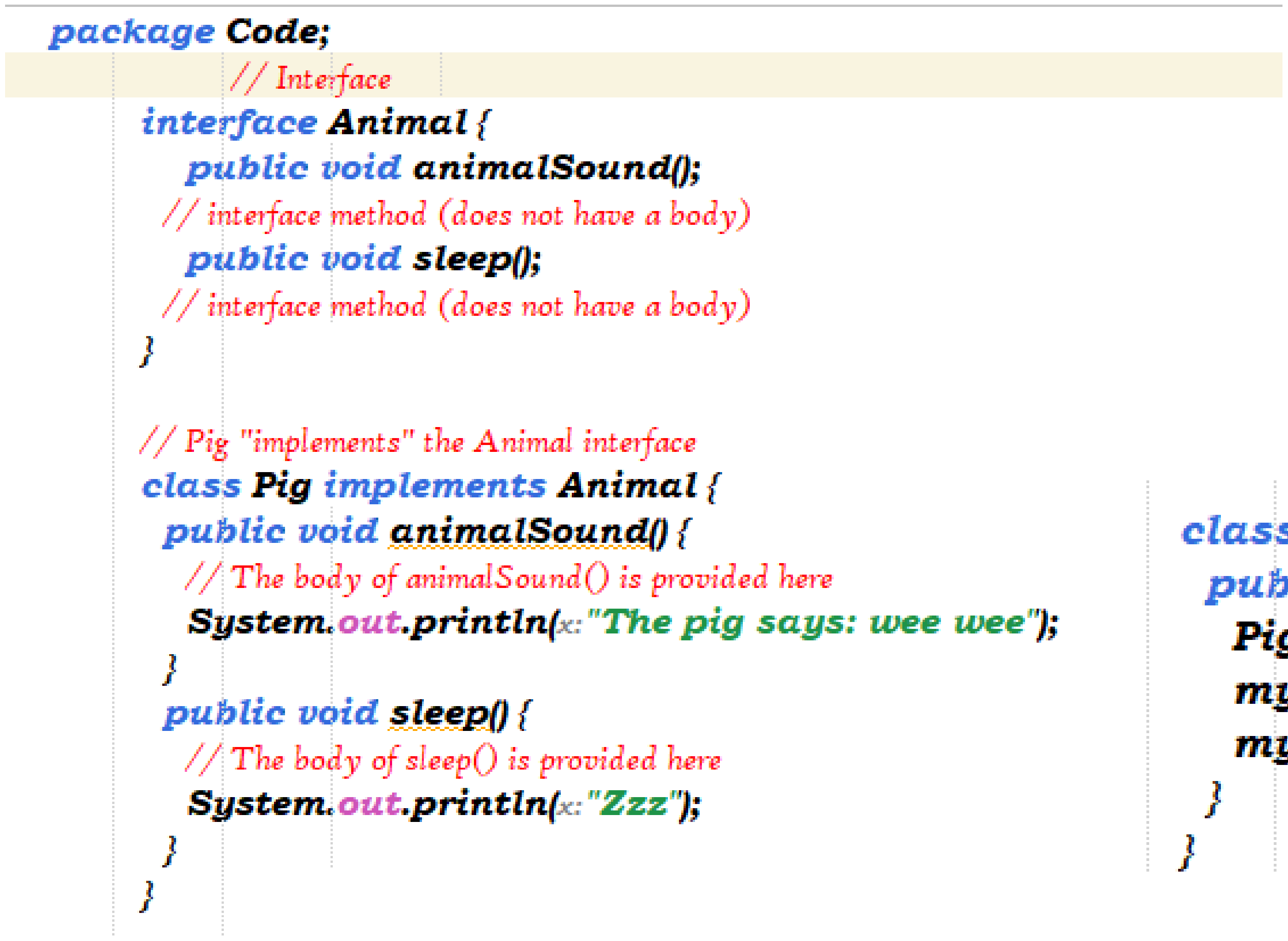
Cont'd . . .

- ***Advantages of Interface***
  - ***To achieve security:***
    - hide certain details and only show the important details of an object (interface).
  - **For multiple inheritance**
    - Java does not support "multiple inheritance" (a class can only inherit from one superclass)
    - However, it can be achieved with interfaces, because the class can **implement** multiple interfaces.
    - **Note:** To implement multiple interfaces, separate them with a comma

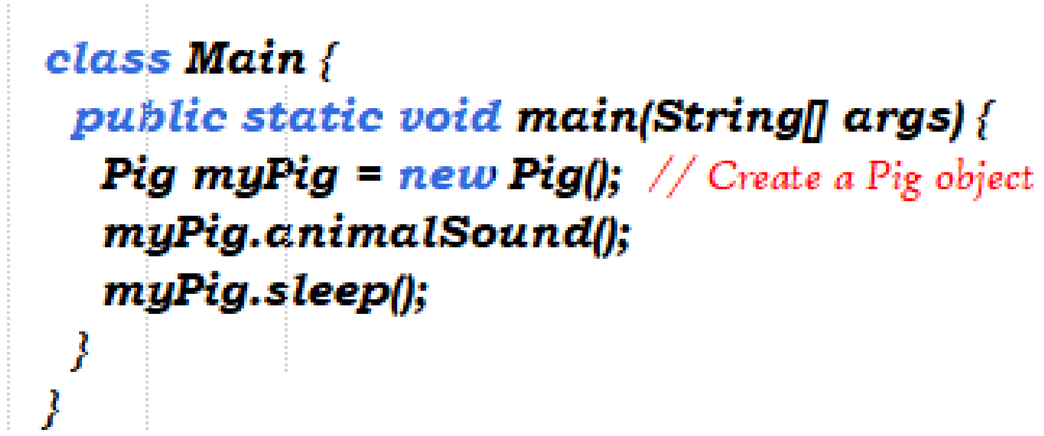
Implementing an Interface

- Like abstract classes:-
  - we can’t create objects of interfaces.
  - To use an interface, other classes must implement it.
  - We use the ***implements*** keyword to implement an interface.
- On the implementation of an interface, you must **override** all of its methods
- Interface methods are by default abstract and public
- Interface attributes are by default public, static and final
- An interface can’t contain a constructor (as it cannot be used to create objects)

Example:



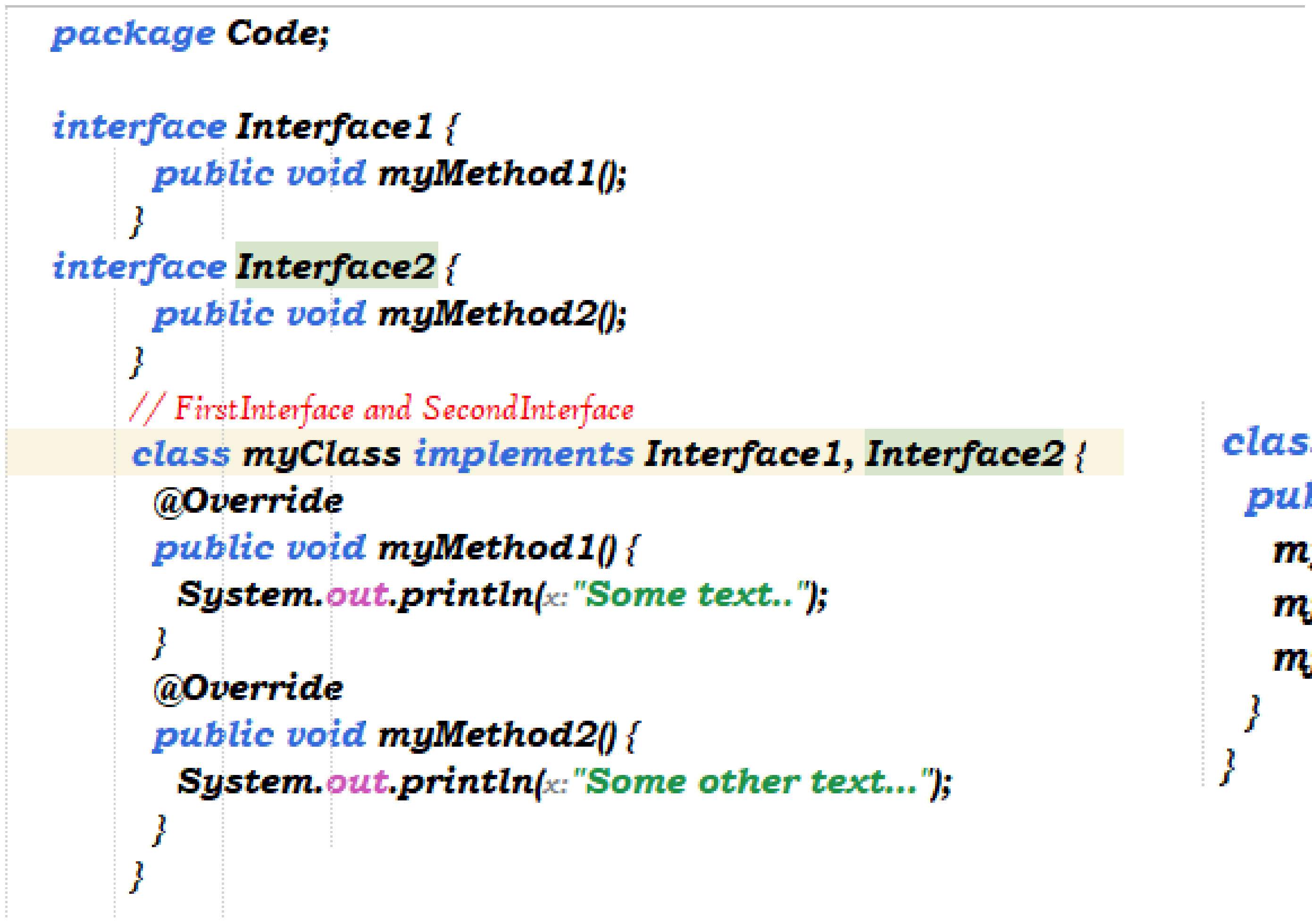
```
package Code;
            // Interface
    interface Animal {
        public void animalSound();
      // interface method (does not have a body)
        public void sleep();
      // interface method (does not have a body)
    }

    // Pig "implements" the Animal interface
    class Pig implements Animal {
      public void animalSound() {
        // The body of animalSound() is provided here
        System.out.println(x: "The pig says: wee wee");
      }
      public void sleep() {
        // The body of sleep() is provided here
        System.out.println(x: "Zzz");
      }
    }
```



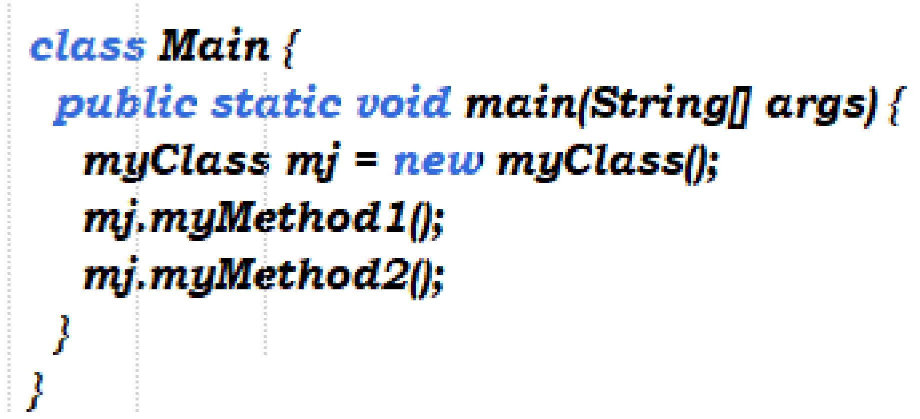
```
class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Multiple interface

- To implement multiple interfaces, separate them with a comma



```
package Code;

interface Interface1 {
        public void myMethod1();
    }
interface Interface2 {
        public void myMethod2();
    }
    // FirstInterface and SecondInterface
    class myClass implements Interface1, Interface2 {
        @Override
        public void myMethod1() {
          System.out.println(x: "Some text..");
        }
        @Override
        public void myMethod2() {
          System.out.println(x: "Some other text...");
        }
    }
```



```
class Main {
  public static void main(String[] args) {
    myClass mj = new myClass();
    mj.myMethod1();
    mj.myMethod2();
  }
}
```

Question

End of Chapter 3

Next → Chapter 4