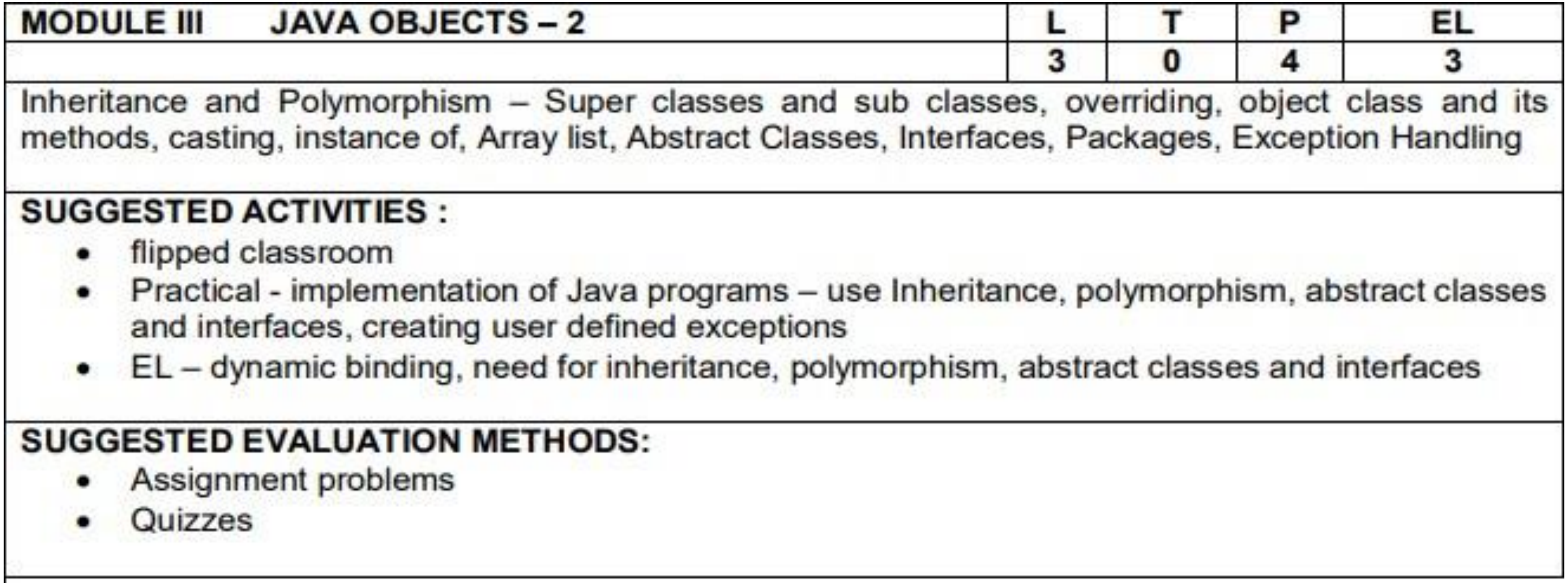
CS6308- Java Programming

V P Jayachitra
Assistant Professor
Department of Computer Technology
MIT Campus
Anna University

Syllabus



MODULE III JAVA OBJECTS – 2	L	T	P	EL
	3	0	4	3

Inheritance and Polymorphism – Super classes and sub classes, overriding, object class and its methods, casting, instance of, Array list, Abstract Classes, Interfaces, Packages, Exception Handling

SUGGESTED ACTIVITIES :

- flipped classroom
- Practical - implementation of Java programs – use Inheritance, polymorphism, abstract classes and interfaces, creating user defined exceptions
- EL – dynamic binding, need for inheritance, polymorphism, abstract classes and interfaces

SUGGESTED EVALUATION METHODS:

- Assignment problems
- Quizzes

Abstract class

- A class that is declared using “**abstract**” keyword is known as abstract class.
- Abstract class can have **abstract methods**(methods without body) as well as **concrete methods** (regular methods with body).
- A normal class(non-abstract class) cannot have abstract methods.
- An abstract class can not be **instantiated**, which means you are not allowed to create an **object** of it.

Why an abstract class?

- A class `Animal` that has a method `sound()` and the subclasses of it like `Dog`, `Lion`, `Horse`, `Cat` etc.
- The animal sound differs from one animal to another, there is no point to implement this method in parent class.
- Every child class must override this method to give its own implementation details, like `Lion` class say “Roar” in this method and `Dog` class say “Woof”.
- All the animal child classes will and should override this method, then there is no point to implement this method in parent class.
- Thus, making this method abstract would be the good choice as by making this method abstract will force all the sub classes to implement this method(otherwise will get compilation error)
- The `Animal` class has an abstract method, must need to declare this class abstract.

Abstract class declaration

An abstract class outlines the methods but not necessarily implements all the methods.

```
//abstract class declaration using abstract keyword
abstract class MyClass {
//abstract method
   abstract void methodA();
//Concrete method
    void methodB(){
        //implementation
     }
}
```

Abstract class

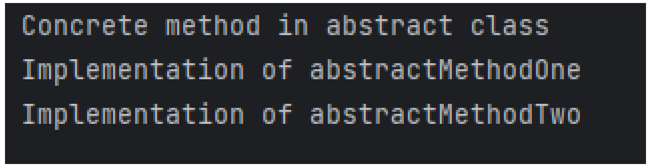
```
 //abstract parent class
 abstract class Animal{
   //abstract method
   public abstract void sound();
                 }
  //Dog class extends Animal class

 public class Dog extends Animal{
     public void sound(){
      System.out.println("Woof");
     }
     public static void main(String args[]){
     Animal obj = new Dog();
     obj.sound();
}
}
```

Output:

```
Woof
```

Abstract class

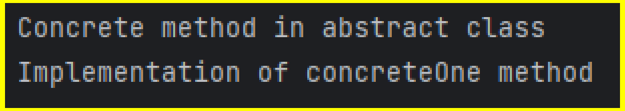
```
//Abstract Class and Method Overriding Example
abstract class BaseAbstract {
    public void concreteMethod() {
        System.out.println("Concrete method in abstract class");
    }
    abstract public void abstractMethodOne();
    abstract public void abstractMethodTwo();
}
class ConcreteImplementation extends BaseAbstract {
    @Override
    public void abstractMethodOne() {
        System.out.println("Implementation of abstractMethodOne");
    }
    @Override
    public void abstractMethodTwo() {
        System.out.println("Implementation of abstractMethodTwo");
    } }
public class AbstractClass {
    public static void main(String[] args) {
        // Cannot instantiate abstract class
        // BaseAbstract obj = new BaseAbstract(); // This would cause an error
        BaseAbstract obj = new ConcreteImplementation();
        obj.concreteMethod();
        obj.abstractMethodOne();
        obj.abstractMethodTwo();
    } }
```



```
Concrete method in abstract class
Implementation of abstractMethodOne
Implementation of abstractMethodTwo
```

Abstract class

```
abstract class BaseAbstract { //abstract class without abstract method
    public void concreteMethod() {
        System.out.println("Concrete method in abstract class");
    }
}
class ConcreteImplementation extends BaseAbstract {
    public void concreteOne() {
        System.out.println("Implementation of concreteOne method");
    }
}
public class AbstractClass {
    public static void main(String[] args) {
        // Cannot instantiate abstract class
        // BaseAbstract obj = new BaseAbstract(); // This would cause an error
        BaseAbstract obj = new ConcreteImplementation();
        obj.concreteMethod();
        //obj.concreteOne(); //error
/*obj is actually an instance of ConcreteImplementation, BaseAbstract reference variable () determines which methods are accessible.*/
        if (obj instanceof ConcreteImplementation) {
            ConcreteImplementation impl = (ConcreteImplementation) obj;
            impl.concreteOne();
        }
    }}
```



```
Concrete method in abstract class
Implementation of concreteOne method
```

Abstract class vs Concrete class

- An abstract class has no use until unless it is extended by some other class.
- An **abstract method** in a class must declare the class abstract as well.
- Concrete class don’t have abstract method . It’s vice versa is not always true: If a class is not having any abstract method then also it can be marked as abstract.
- Abstract class can have non-abstract method (concrete) as well.

Abstract class

- Abstract method

  1)Abstract method has no body.
  2) Always end the declaration with a **semicolon**(;).
  3)It must be **overridden**. An abstract class must be extended and in a same way abstract method must be overridden.
  4) A class has to be declared abstract to have abstract methods.

Rule 1

- There are cases when it is difficult or often unnecessary to implement all the methods in parent class.
- In these cases, the parent class can be declared as abstract, which makes it a special class which is not complete on its own.
- A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.

Rule 2

- Abstract class cannot be instantiated which means you cannot create the object of it.
- To use this class, create another class that extends this class and provides the implementation of abstract methods, then can use the object of that child class to call non-abstract methods of parent class as well as implemented methods(those that were abstract in parent but implemented in child class).

Rule 3

- If a child does not implement all the abstract methods of abstract parent class, then the child class must need to be declared abstract as well.
- **Abstraction is a process to show only “relevant” data and “hide” unnecessary details of an object from the user.**
- Since abstract class allows concrete methods as well, it does not provide 100% abstraction. Abstract class provides partial abstraction.
- **Interfaces** on the other hand are used for 100% abstraction

Interfaces

- Using interface, can specify what a class must do, but not how it does it.
- Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body.
- An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body).
- Also, the variables declared in an interface are public, static & final by default.

Why interface in java?

- To exhibit full abstraction.
    - Methods in interfaces do not have body, they have to be implemented by the class before accessing them.
    - The class that implements interface must implement all the methods of that interface.
- To support multiple inheritance
    - Java programming language does not allow to extend more than one class, However allow to implement more than one interfaces in the class.

Syntax

```
interface MyInterface {
//all methods are abstract by default
   public void methodA();
   public void methodB();
}
```

Interfaces

- Using interface, can specify what a class must do, but not how it does it.
- Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body.
- An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body).
- Also, the variables declared in an interface are public, static & final by default.

Interfaces

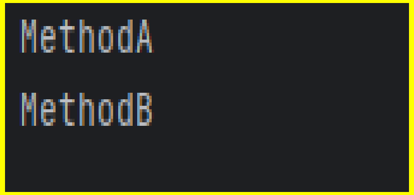
- **Interfaces** are used to achieve abstraction and allow multiple inheritance of types, enabling a class to implement multiple interfaces.
- **Abstract methods** in interfaces define what methods must be implemented by the classes that use the interface.
- **Default and static methods** provide flexibility in interfaces by allowing them to contain method implementations.

```
public interface MyInterface {
    // Abstract method (does not have a body)
    void myMethod();

    // Default method (with a body)
    default void defaultMethod() {
        System.out.println("Default implementation");
    }

    // Static method (with a body)
    static void staticMethod() {
        System.out.println("Static method");
    } }
```

```
public interface ExtendedInterface extends MyInterface {
    void anotherMethod();
}
```

Example of an Interface in Java

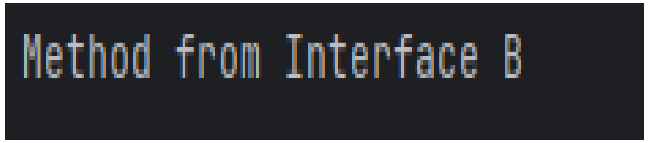
```
package two;
interface A {
   public void methodA();
   public void methodB();
}
class InterfaceExample  implements A {
   @Override
   public void methodA() {
      System.out.println("MethodA ");    }
   @Override
   public  void methodB() {
      System.out.println("MethodB ");     }
   public static void main(String args[])
   {
      InterfaceExample obj=new InterfaceExample ();
      obj.methodA();
      obj.methodB();
   }
}
```



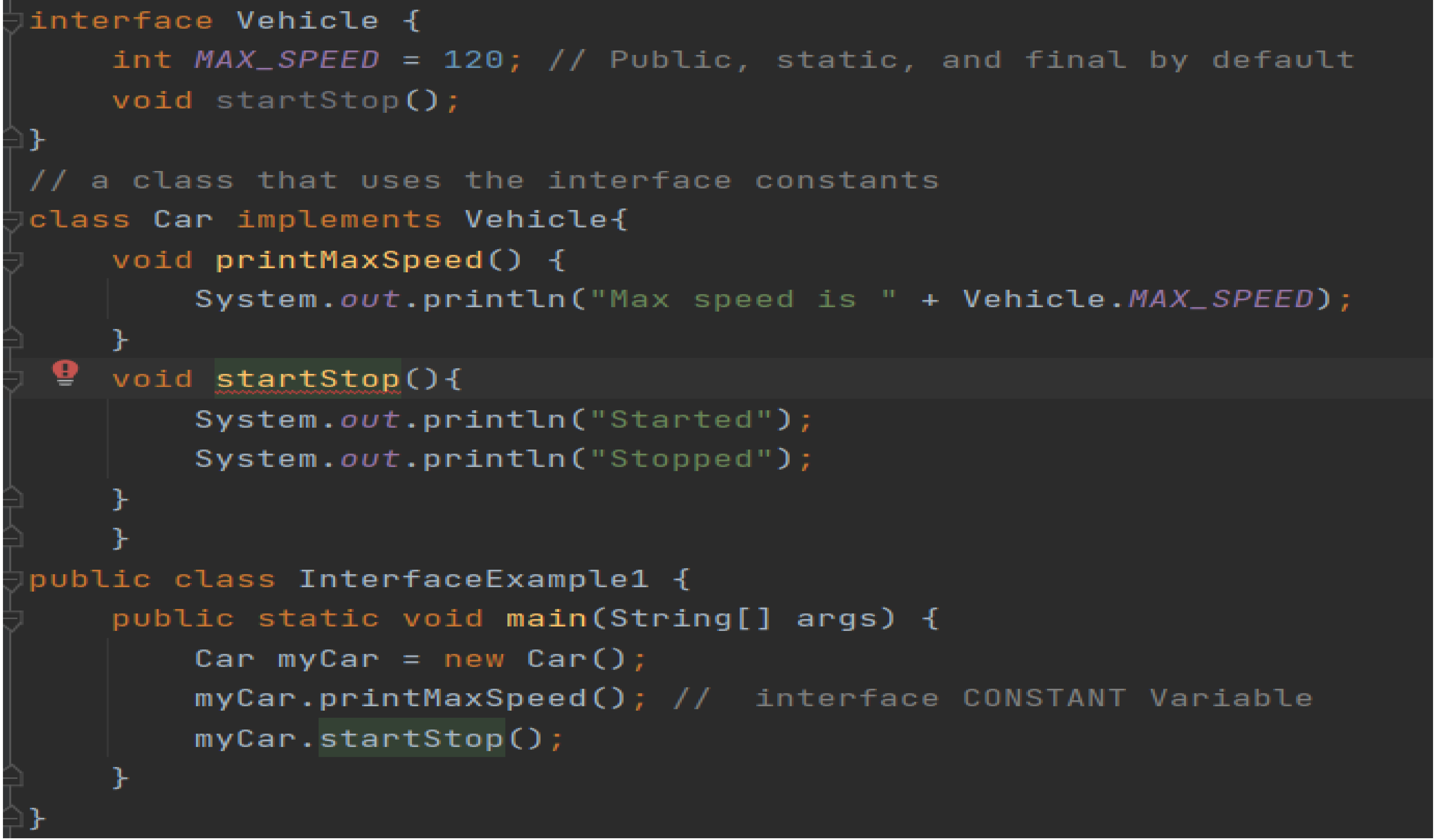
```
MethodA
MethodB
```

This program, the class InterfaceExtend only implements interface InfB, however it has to provide the implementation of all the methods of interface InfA as well, because interface InfB extends InfA.

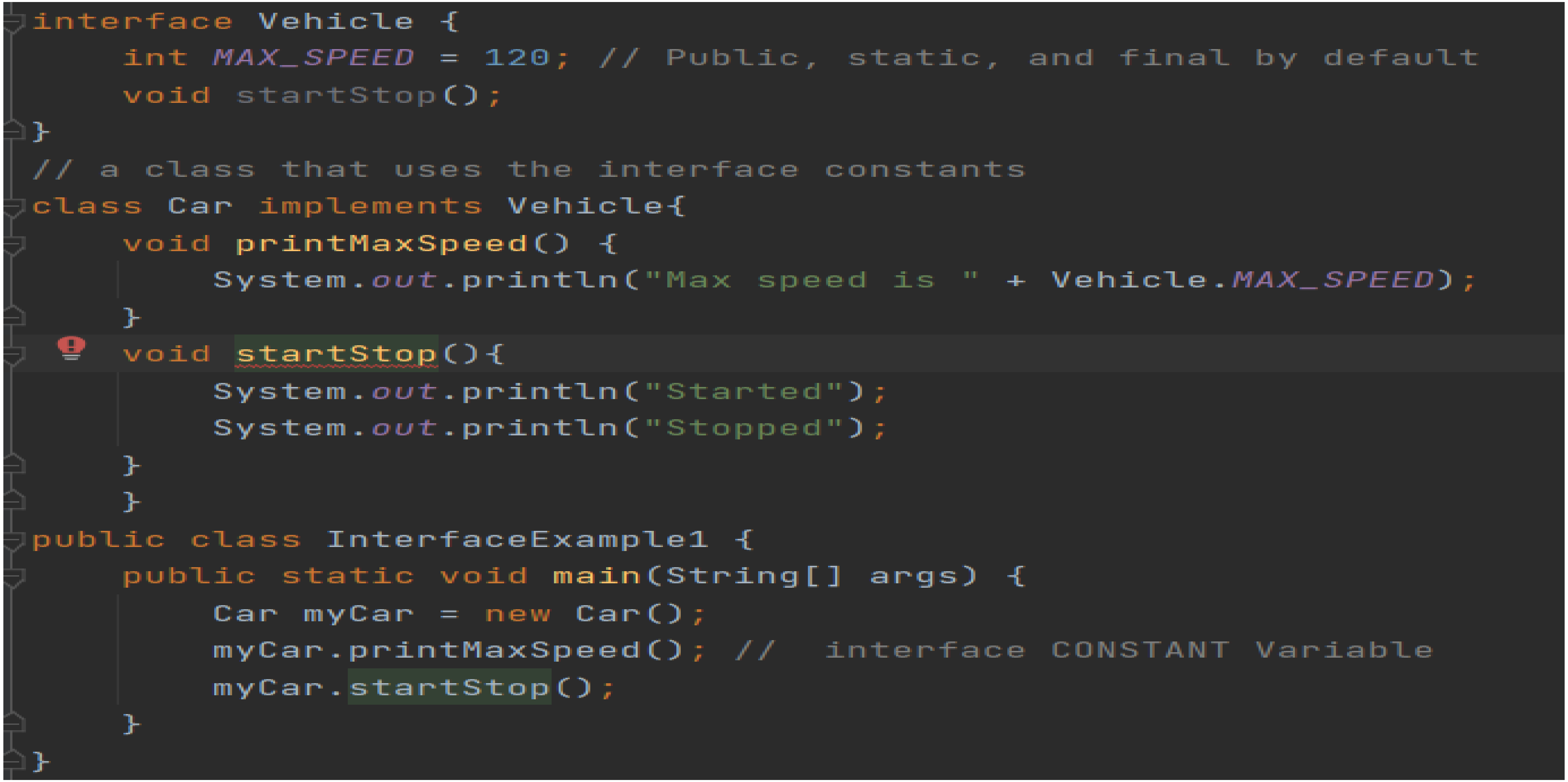
```
interface A {
   public void methodA();
}

interface B extends A {
   public void methodB();
}
class InterfaceExtend  implements A, B {
   @Override
   public void methodA() {
      System.out.println("Method from Interface A ");    }
   @Override
   public  void methodB() {
      System.out.println("Method from Interface B");     }
   public static void main(String args[])
   {
      InterfaceExtend obj=new InterfaceExtend();
      obj.methodB();
   }
}
```



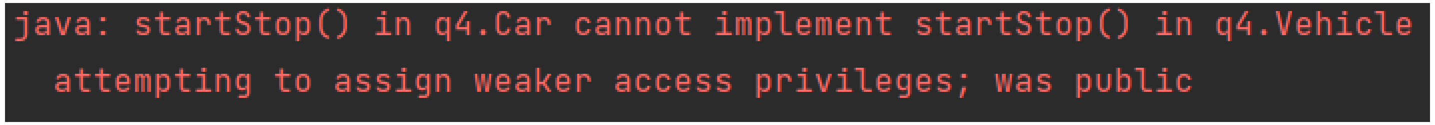
```
Method from Interface B
```



```java
interface Vehicle {
    int MAX_SPEED = 120; // Public, static, and final by default
    void startStop();
}
// a class that uses the interface constants
class Car implements Vehicle{
    void printMaxSpeed() {
        System.out.println("Max speed is " + Vehicle.MAX_SPEED);
    }
    void startStop(){
        System.out.println("Started");
        System.out.println("Stopped");
    }
    }
public class InterfaceExample1 {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.printMaxSpeed(); //  interface CONSTANT Variable
        myCar.startStop();
    }
}
```

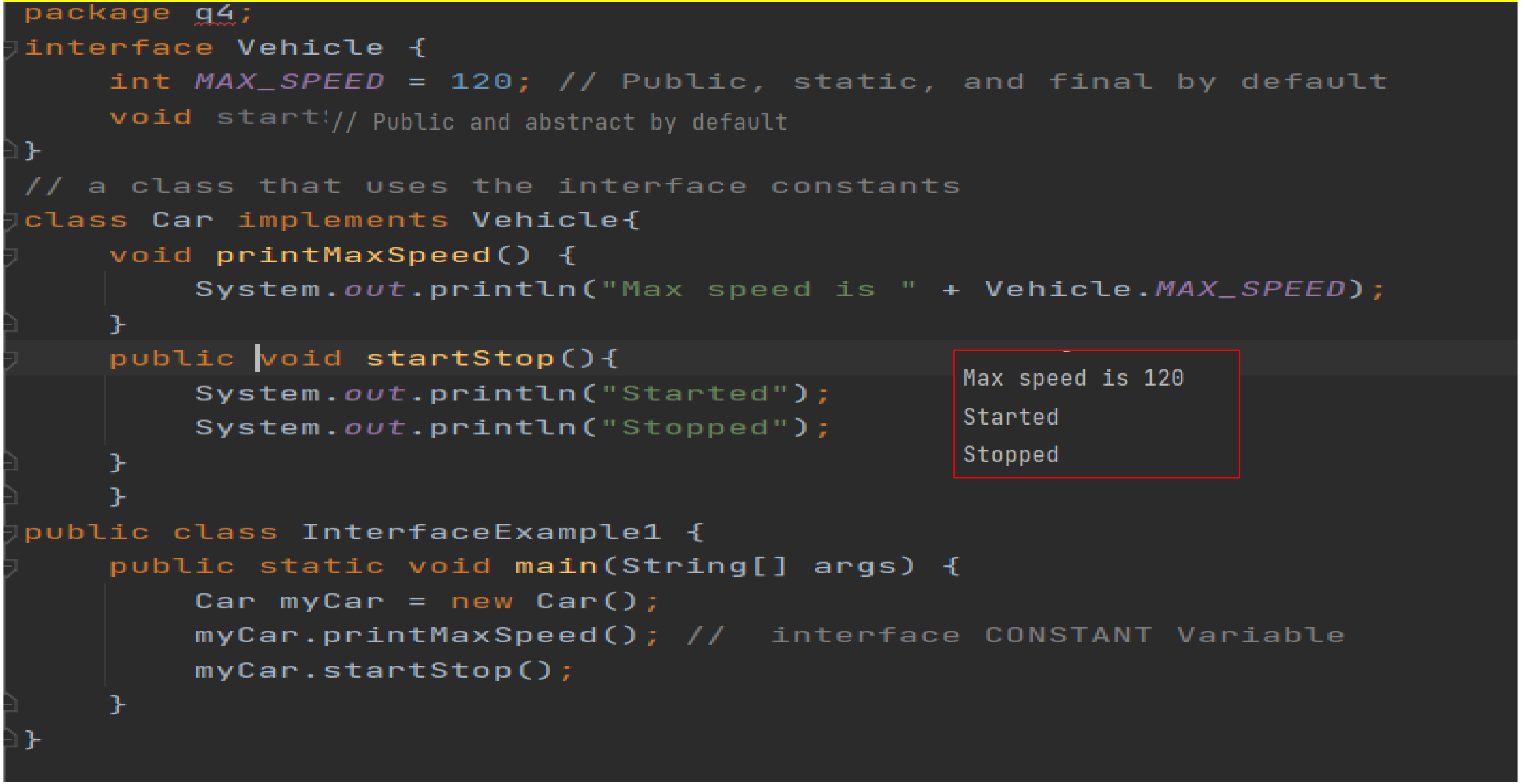


```
interface Vehicle {
    int MAX_SPEED = 120; // Public, static, and final by default
    void startStop();
}
// a class that uses the interface constants
class Car implements Vehicle{
    void printMaxSpeed() {
        System.out.println("Max speed is " + Vehicle.MAX_SPEED);
    }
    void startStop(){
        System.out.println("Started");
        System.out.println("Stopped");
    }
    }
public class InterfaceExample1 {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.printMaxSpeed(); //  interface CONSTANT Variable
        myCar.startStop();
    }
}
```



```
java: startStop() in q4.Car cannot implement startStop() in q4.Vehicle
  attempting to assign weaker access privileges; was public
```

Variables in an interface are implicitly public, static, and final.
They must be initialized when declared.
Interface methods are implicitly public and abstract.
interfaces can also have default methods with a body, and static methods.
The public modifier is required when implementing these methods in a class.



```
package q4;
interface Vehicle {
    int MAX_SPEED = 120; // Public, static, and final by default
    void start// Public and abstract by default
}
// a class that uses the interface constants
class Car implements Vehicle{
    void printMaxSpeed() {
        System.out.println("Max speed is " + Vehicle.MAX_SPEED);
    }
    public void startStop(){
        System.out.println("Started");
        System.out.println("Stopped");
    }
    }
public class InterfaceExample1 {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.printMaxSpeed(); //  interface CONSTANT Variable
        myCar.startStop();
    }
}
```

```
Max speed is 120
Started
Stopped
```

Interface with default methods

```
public interface Floatable {
    default void repair() {
                System.out.println("Repairing Floatable object");
    }
}
public interface Flyable {
    default void repair() {
                System.out.println("Repairing Flyable object");
    }
}
public class ArmoredCar extends Car implements Floatable, Flyable {
    // this won't compile
}
```

Java disallows inheritance of multiple implementations of the same methods, defined in separate interfaces.

Interfaces Extending Other Interfaces

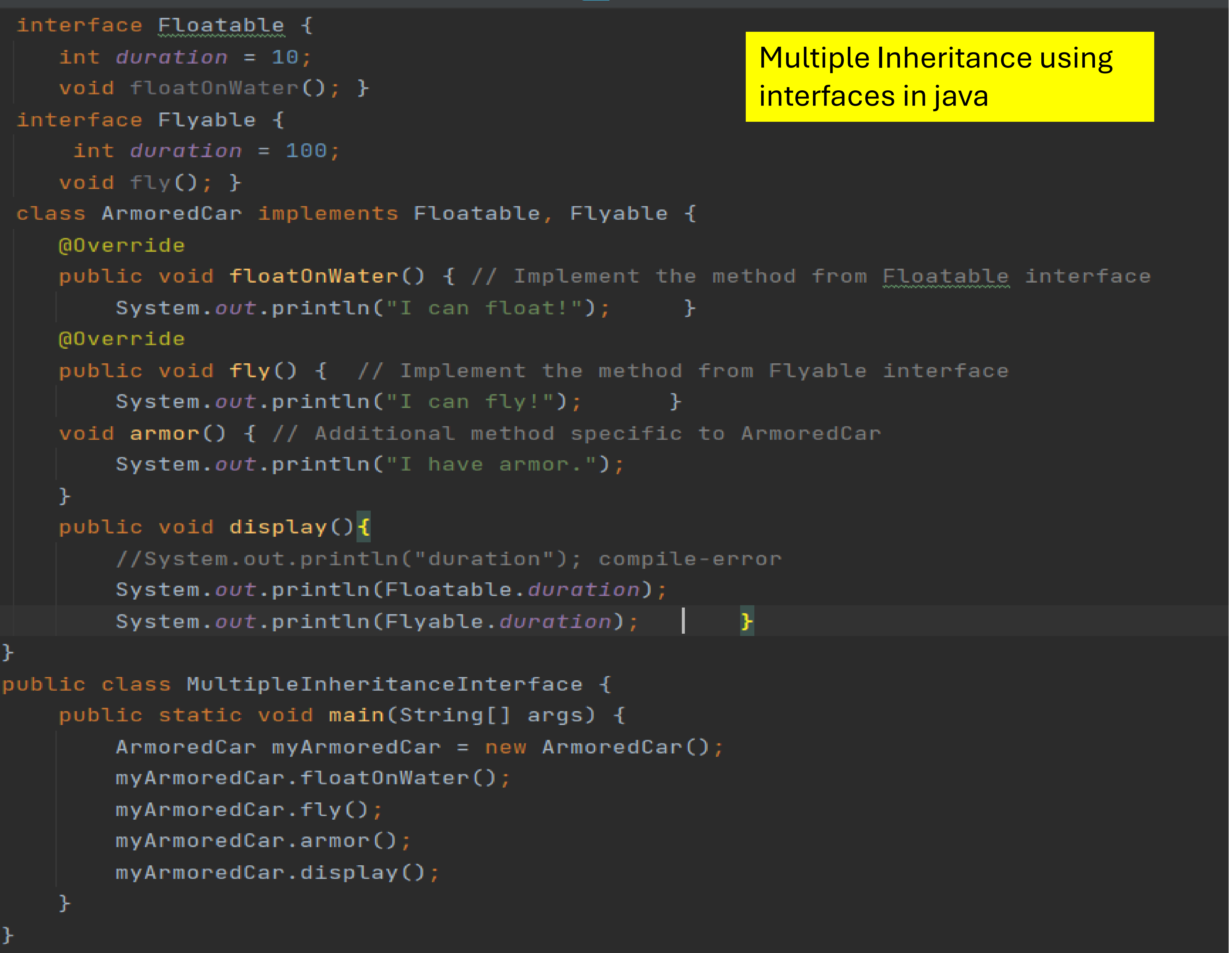
```
public interface Floatable {
   void floatOnWater();
}

interface interface Flyable {
   void fly();
}

public interface SpaceTraveller extends Floatable, Flyable {
   void remoteControl();
}
```

Interfaces

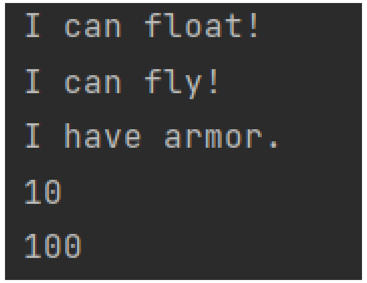
- **Multiple inheritance**

```
public class MyClass implements Interface1, Interface2 {
   @Override
   public void methodFromInterface1() {
      // Implementation
   }

   @Override
   public void methodFromInterface2() {
      // Implementation
   }
}
```

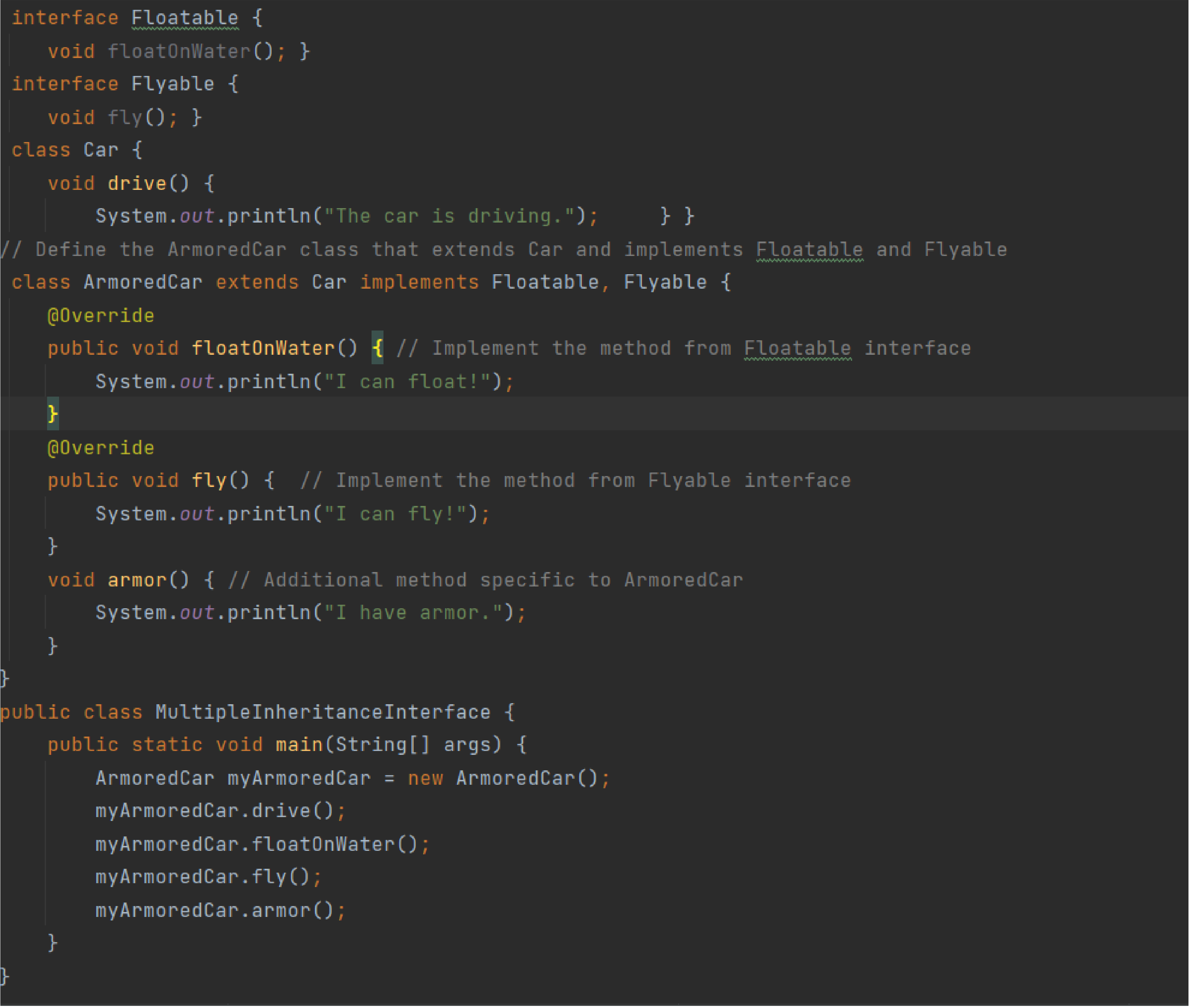
Multiple Inheritance using interfaces in java



```java
interface Floatable {
    int duration = 10;
    void floatOnWater(); }
interface Flyable {
     int duration = 100;
    void fly(); }
class ArmoredCar implements Floatable, Flyable {
    @Override
    public void floatOnWater() { // Implement the method from Floatable interface
        System.out.println("I can float!");      }
    @Override
    public void fly() {  // Implement the method from Flyable interface
        System.out.println("I can fly!");      }
    void armor() { // Additional method specific to ArmoredCar
        System.out.println("I have armor.");
    }
    public void display(){
        //System.out.println("duration"); compile-error
        System.out.println(Floatable.duration);
        System.out.println(Flyable.duration);          }
}
public class MultipleInheritanceInterface {
    public static void main(String[] args) {
        ArmoredCar myArmoredCar = new ArmoredCar();
        myArmoredCar.floatOnWater();
        myArmoredCar.fly();
        myArmoredCar.armor();
        myArmoredCar.display();
    }
}
```

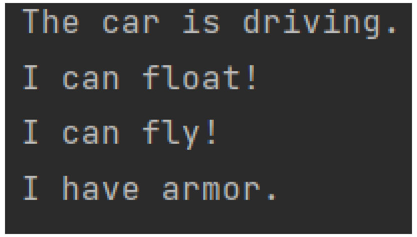


```
I can float!
I can fly!
I have armor.
10
100
```

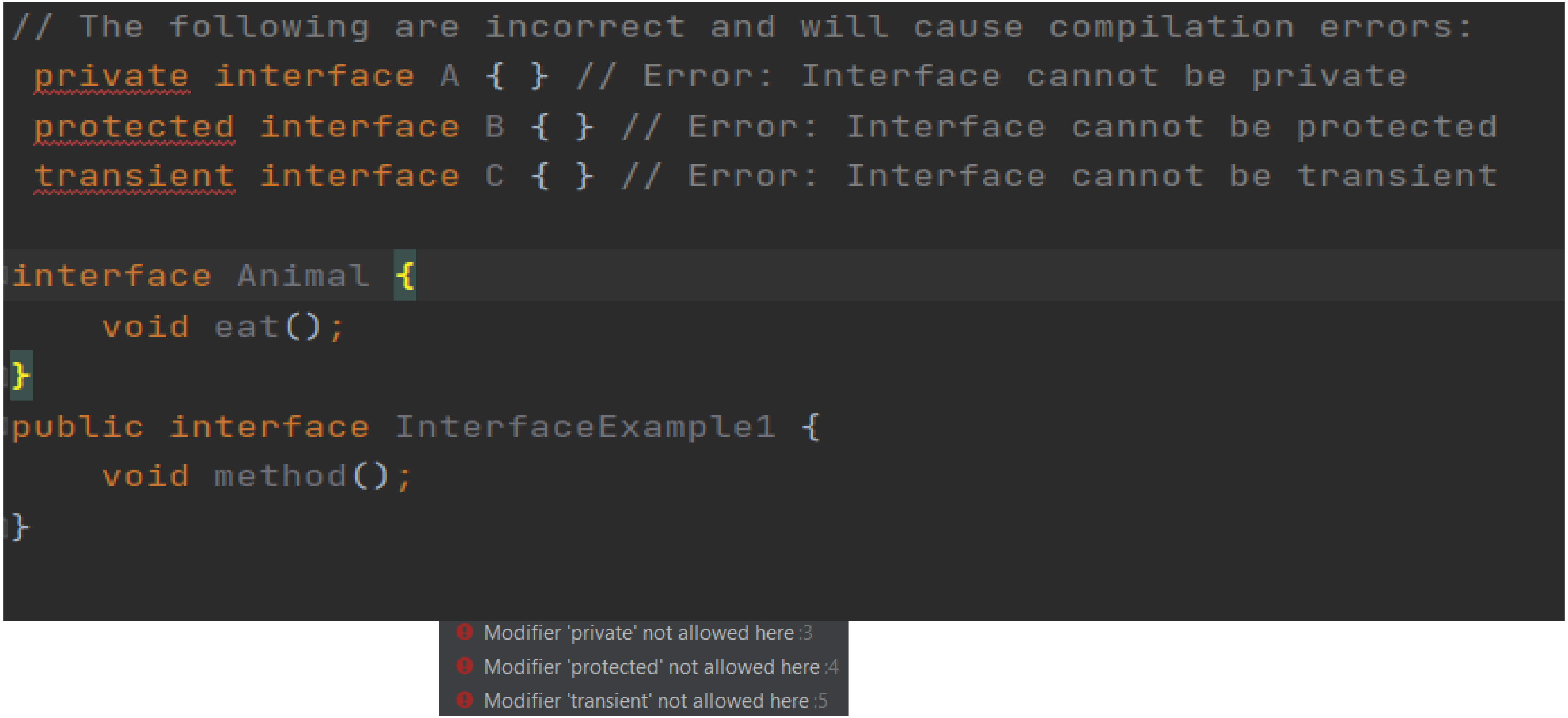


```java
interface Floatable {
    void floatOnWater(); }
interface Flyable {
    void fly(); }
class Car {
    void drive() {
        System.out.println("The car is driving.");     } }
// Define the ArmoredCar class that extends Car and implements Floatable and Flyable
class ArmoredCar extends Car implements Floatable, Flyable {
    @Override
    public void floatOnWater() { // Implement the method from Floatable interface
        System.out.println("I can float!");
    }
    @Override
    public void fly() {  // Implement the method from Flyable interface
        System.out.println("I can fly!");
    }
    void armor() { // Additional method specific to ArmoredCar
        System.out.println("I have armor.");
    }
}
public class MultipleInheritanceInterface {
    public static void main(String[] args) {
        ArmoredCar myArmoredCar = new ArmoredCar();
        myArmoredCar.drive();
        myArmoredCar.floatOnWater();
        myArmoredCar.fly();
        myArmoredCar.armor();
    }
}
```

Multiple Inheritance using interfaces in java



```
The car is driving.
I can float!
I can fly!
I have armor.
```



```java
// The following are incorrect and will cause compilation errors:
 private interface A { } // Error: Interface cannot be private
 protected interface B { } // Error: Interface cannot be protected
 transient interface C { } // Error: Interface cannot be transient

interface Animal {
    void eat();
}
public interface InterfaceExample1 {
    void method();
}
```

```
Modifier 'private' not allowed here :3
Modifier 'protected' not allowed here :4
Modifier 'transient' not allowed here :5
```

Summary

- No instantiate for interface in java. That means object of an interface cannot be created.
- Interface provides full abstraction as none of its methods have body. On the other hand abstract class provides partial abstraction as it can have a
- Abstract and concrete(methods with body) methods both.
- `implements` keyword is used by classes to implement an interface.
- While providing implementation in class of any method of an interface, it needs to be mentioned as public.
- Class that implements any interface must implement all the methods of that interface, else the class should be declared abstract.
- Interface cannot be declared as private, protected or transient.
- **All the interface methods are by default abstract and public.**

Summary

- Variables declared in interface are **public, static and final** by default.

```
interface Try {
 int  a=10;
 public int a=10;
 public static final int a=10;
final int a=10;
static int a=0; }
```

- Interface variables must be initialized at the time of declaration otherwise compiler will throw an error.

```
interface Try {
    int x;//Compile-time error }
```

Summary

- Inside any implementation class, change of interface variables not allowed because by default, they are public, static and final.

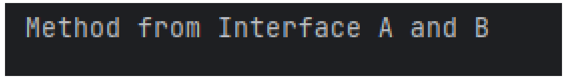
```
class Sample implements Try {
 public static void main(String args[]) {
       x=20; //compile time error }
}
```

- An interface can extend any interface but cannot implement it.
- Class implements interface and interface extends interface.
- A **class** can implement any **number of interfaces**.

Summary

- If there are **two or more same methods** in two interfaces and a class implements both interfaces, implementation of the method once is enough.

```
interface A {
  void method();
}
interface B {
  void method();
}
class InterfaceMethod implements A, B {
  // Implementing the method() from both interfaces
  @Override
  public void method() {
    System.out.println("Method from Interface A and B");
  }
  public static void main(String[] args) {
    InterfaceMethod obj = new InterfaceMethod();
    obj.method();
  }
}
```

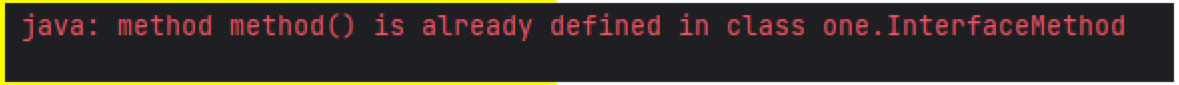


```
Method from Interface A and B
```

Summary

- A class cannot implement two interfaces that have methods with same name but different return type.

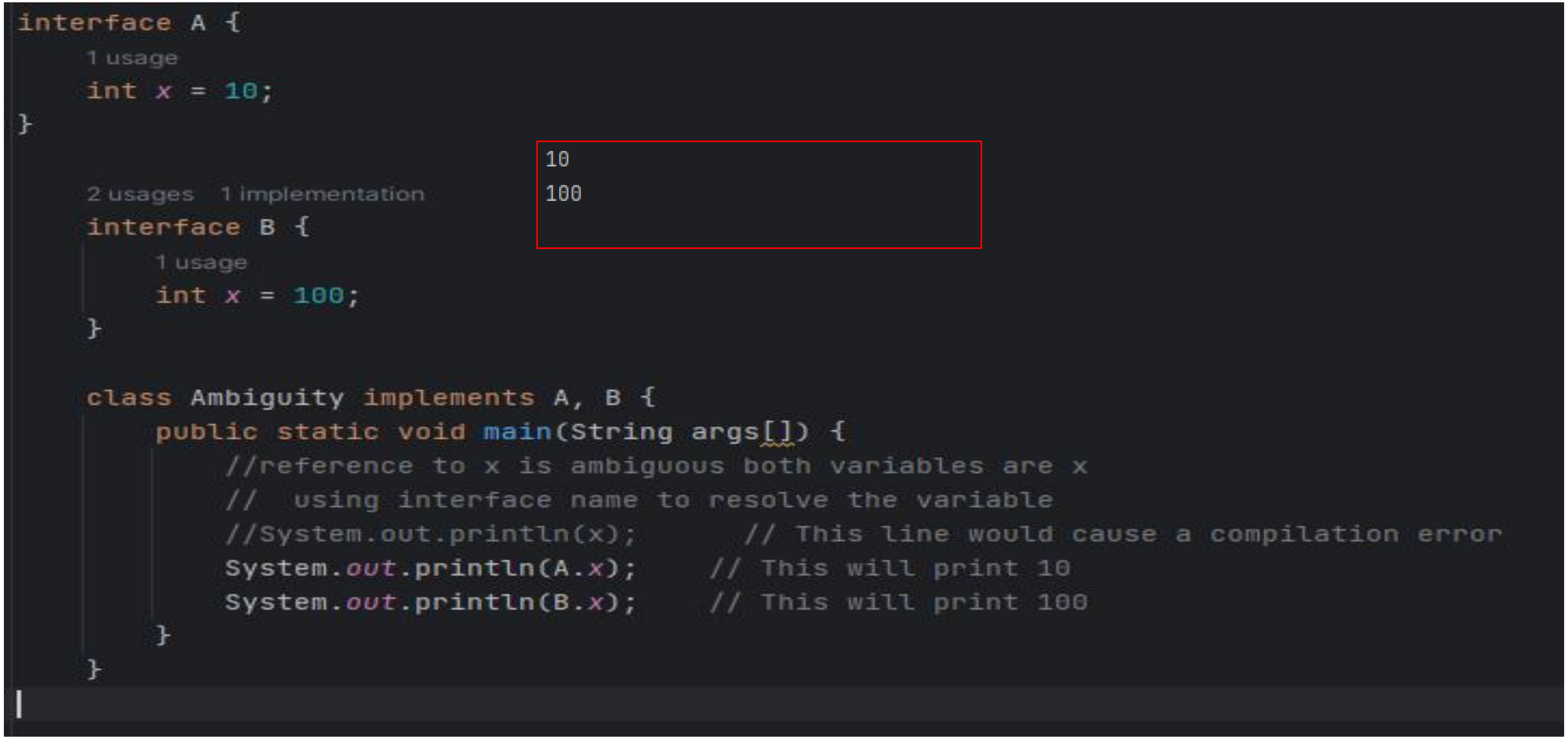
```
package one;
interface A {
  void method(); }
interface B {
  int method(); }
class InterfaceMethod implements A, B {
  // Implementing the method() from both interfaces
  @Override
  public void method() {  //error
    System.out.println("Method from Interface A and B");    }
  public  int method() { //error
    System.out.println("Method from Interface A and B");   return 1;    }
  public static void main(String[] args) {
    InterfaceMethod obj = new InterfaceMethod();
    obj.method();
    System.out.println(obj.method());
  } }
```



```
java: method method() is already defined in class one.InterfaceMethod
```

Summary

- Variable names conflicts can be resolved by interface name.



```
interface A {
    1 usage
    int x = 10;
}

    2 usages   1 implementation
    interface B {
        1 usage
        int x = 100;
    }

    class Ambiguity implements A, B {
        public static void main(String args[]) {
            //reference to x is ambiguous both variables are x
            //  using interface name to resolve the variable
            //System.out.println(x);      // This line would cause a compilation error
            System.out.println(A.x);    // This will print 10
            System.out.println(B.x);    // This will print 100
        }
    }
```

```
10
100
```

Polymorphism

- **Polymorphism in Java** is the task that performs a single action in different ways. i.e., a single method can perform different actions depending on the type of object
- “poly” means many and “morphism” means form. It just means many forms.
- The method that gets called at **run-time** depends on the *type* of the object at run-time.
- Polymorphism in Java is mainly categorized into two types:
  - **Compile-time Polymorphism**
    - Compile-time polymorphism, also known as static polymorphism, is achieved by method overloading.
    - Compile-time polymorphism is the method to be executed is determined at the time of compilation.
  - Runtime Polymorphism
    - Runtime polymorphism, also known as dynamic polymorphism, is achieved through method overriding.
    - Runtime overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.

Polymorphism

- **Method Overloading**: **Compile-time Polymorphism**
  - Overloading occurs when two or more methods in the same class have the same name but different parameters.
  - when multiple methods in the same class have the same name but different parameters (type, number, or both).
  - The method to be executed is determined at compile-time based on the method signature.
- **Method Overriding**: Runtime Polymorphism
  - Overriding occurs when the method signature is the same in the superclass and the child class.
  - when a subclass provides a specific implementation of a method that is already defined in its superclass is called method overriding.
  - The method to be executed is determined at runtime based on the actual object type.

Overriding vs Overloading

- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different data
- Overriding lets you define a similar operation in different ways for different object types
- **overriding** a method is when a subclass has method with the same signature (name and parameter list) as its superclass
  - Mover’s act() and Bouncer’s act()
- **Overloading** a method is when two methods have the same name, but different parameter lists
  - Arrays.sort(array, begin, end) and Arrays.sort(array)
- cannot overload static methods

Static vs Dynamic polymorphism

Description	Static Polymorphism	Dynamic Polymorphism
Definition	Multiple methods in the same class have the same name but different parameters	Subclass provides a specific implementation of a method that is already defined in its superclass
Resolution Time	Resolved at Compile-time	Resolved at Runtime
How it's achieved	method overloading	Method overriding
Binding	Early binding	Late binding
Performance	Generally faster as it's resolved at compile-time	Slightly slower due to runtime resolution, but provides more flexibility
Usage	Used when different implementations is based on different parameter types or counts	Used when subclasses need to provide their own implementations of a method
Who?	The compiler determines which method to call based on the method signature	The JVM determines which method to call based on the actual object type at runtime

overriding

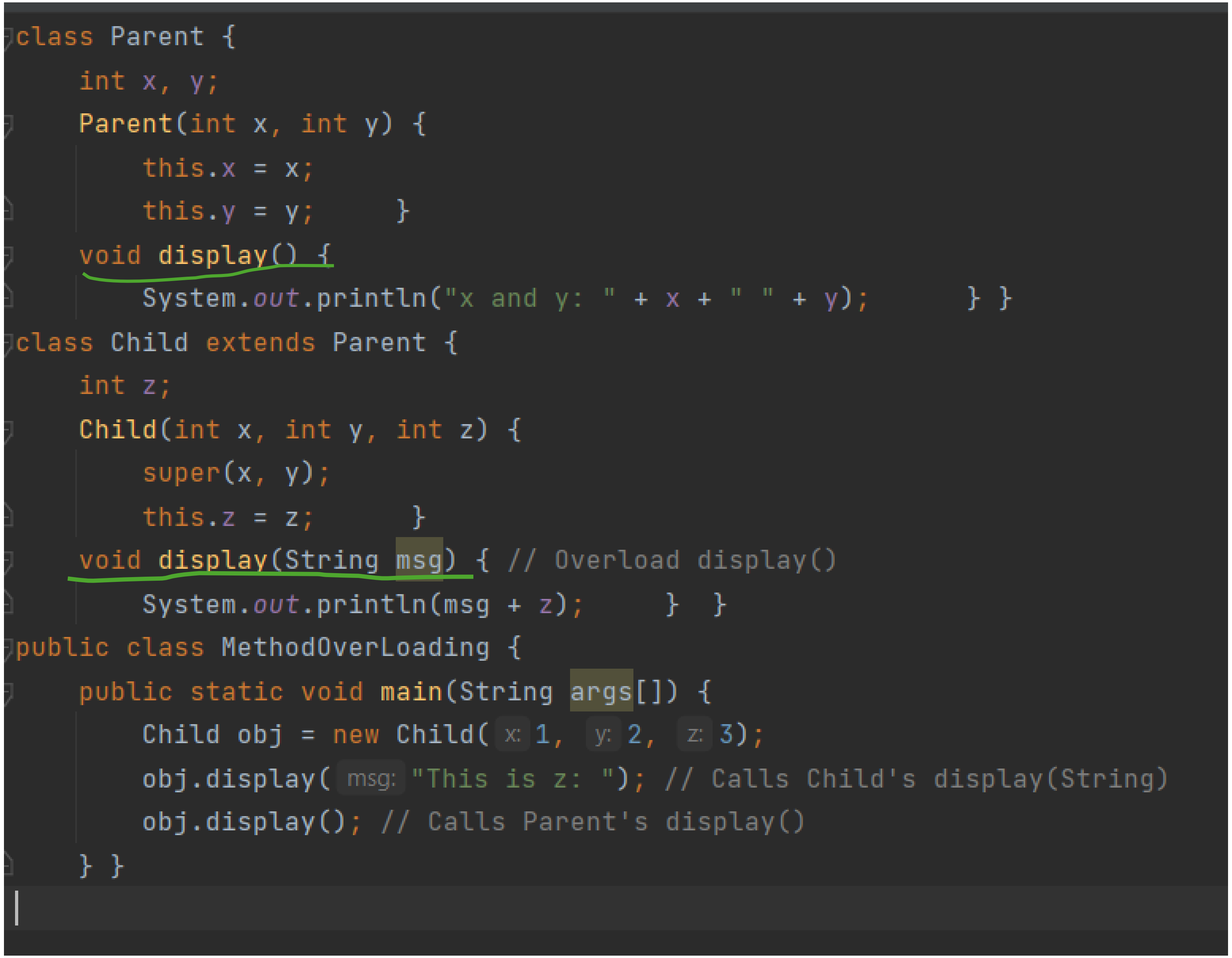
- Method overriding:
  - A method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass
- A parent method can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- Any method that is not `final` may be overridden by a descendant class
- Same signature as method in ancestor
- May not reduce visibility
- May use the original method if simply want to add more behavior to existing
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

```
class A {
   int i, j;
   A(int a, int b) {
      i = a;
      j = b;
   }
   void show() {
      System.out.println("i and j: " + i + " " + j);
   }
   void callme() {
      System.out.println("Inside A's callme method");
   } }
```

```
public class Main {
   public static void main(String[] args) {
      A a = new A(1, 2);
      a.show();  // Calls A's show method
      a.callme(); // Calls A's callme method
      B b = new B(3, 4, 5);
     // Calls B's show method,  also calls A's show method
      b.show();
 // Calls B's overloaded show()  with a string argument
      b.show("Value of k: ");
      b.callme(); // Calls B's callme method  }    }
```

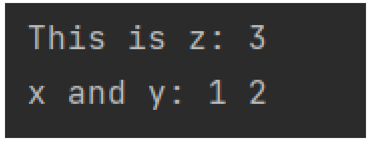
```
class B extends A {
   int k;

   B(int a, int b, int c) {
      super(a, b);
      k = c;
   }
   @Override
   void show() {
      super.show();  // Calls A's show() method
      System.out.println("k: " + k);
   }

   // Method overloading
   void show(String msg) {
      System.out.println(msg + k);
   }

   @Override
   void callme() {
      System.out.println("Inside B's callme method");
   }
}
```

Method overriding occurs only when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded.

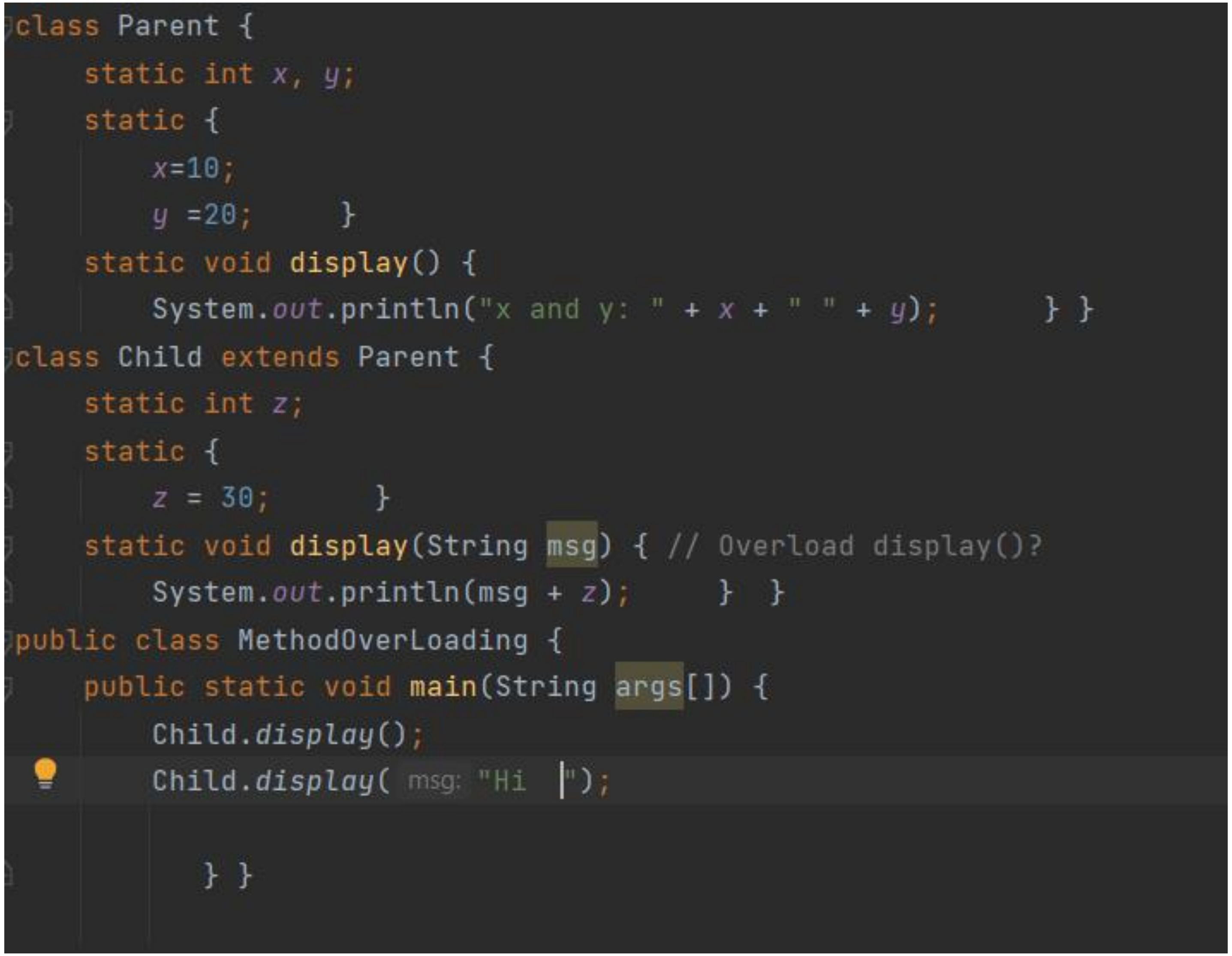


```
class Parent {
    int x, y;
    Parent(int x, int y) {
        this.x = x;
        this.y = y;     }
    void display() {
        System.out.println("x and y: " + x + " " + y);      } }
class Child extends Parent {
    int z;
    Child(int x, int y, int z) {
        super(x, y);
        this.z = z;      }
    void display(String msg) { // Overload display()
        System.out.println(msg + z);     }  }
public class MethodOverLoading {
    public static void main(String args[]) {
        Child obj = new Child( x: 1,  y: 2,  z: 3);
        obj.display( msg: "This is z: "); // Calls Child's display(String)
        obj.display(); // Calls Parent's display()
    } }
```

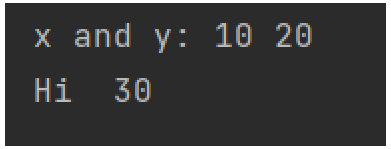
Method overriding occurs only when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded.



```
This is z: 3
x and y: 1 2
```



```
class Parent {
    static int x, y;
    static {
        x=10;
        y =20;     }
    static void display() {
        System.out.println("x and y: " + x + " " + y);      } }
class Child extends Parent {
    static int z;
    static {
        z = 30;      }
    static void display(String msg) { // Overload display()?
        System.out.println(msg + z);      }  }
public class MethodOverLoading {
    public static void main(String args[]) {
        Child.display();
        Child.display( msg: "Hi  ");

           } }
```



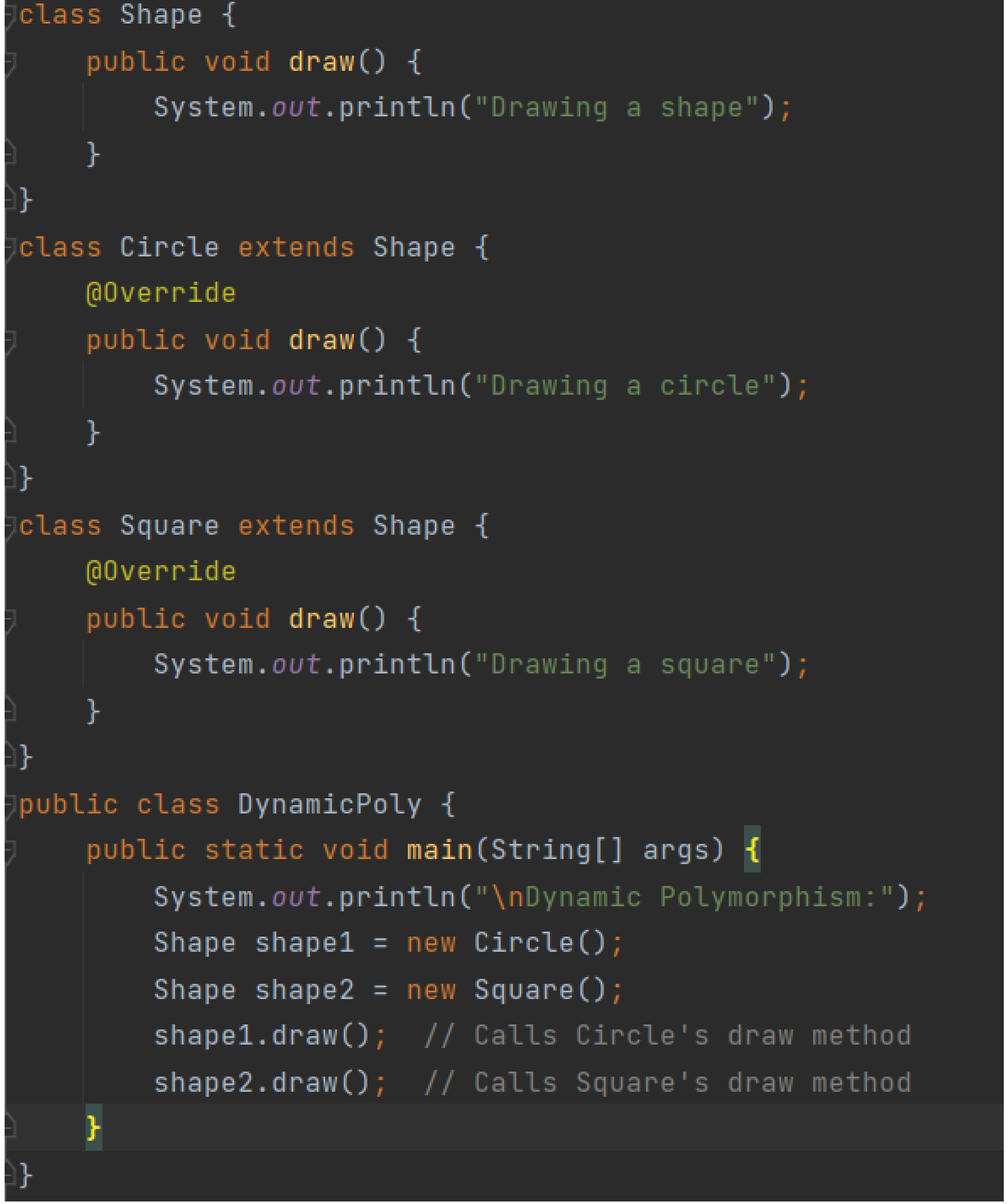
```
x and y: 10 20
Hi  30
```

Dynamic method dispatch

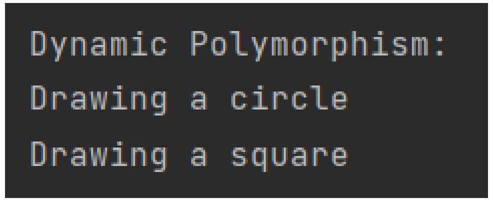
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Dynamic method dispatch implements run-time polymorphism.
- Static methods cannot be overridden, only hidden in subclasses
- Dynamic dispatch allows for polymorphic behavior, where the same method call can result in different actions based on the actual object type
-
  ```
  Animal animal2 = new Dog();
  animal2.makeSound(); // Calls Dog’s makeSound()
  ```
- animal2.makeSound() calls Dog's version of the method, even though the reference type is Animal

Why Overridden Methods?

- Overridden methods allow Java to support run-time polymorphism.
- Polymorphism allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
- Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.

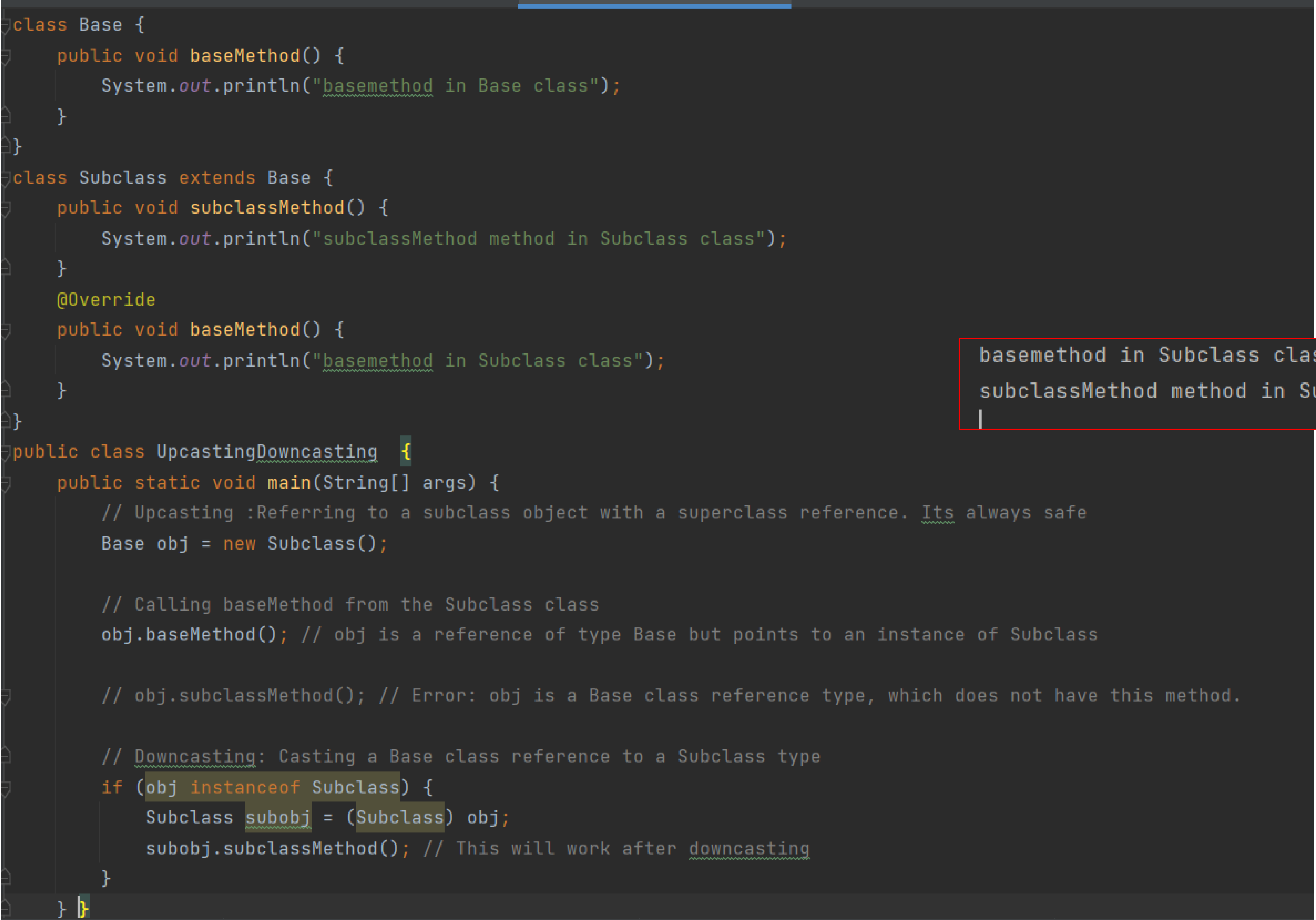


```java
class Shape {
    public void draw() {
        System.out.println("Drawing a shape");
    }
}
class Circle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a circle");
    }
}
class Square extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a square");
    }
}
public class DynamicPoly {
    public static void main(String[] args) {
        System.out.println("\nDynamic Polymorphism:");
        Shape shape1 = new Circle();
        Shape shape2 = new Square();
        shape1.draw();  // Calls Circle's draw method
        shape2.draw();  // Calls Square's draw method
    }
}
```



```
Dynamic Polymorphism:
Drawing a circle
Drawing a square
```

Dynamic Polymorphism And Upcasting vs Downcasting



```java
class Base {
    public void baseMethod() {
        System.out.println("basemethod in Base class");
    }
}
class Subclass extends Base {
    public void subclassMethod() {
        System.out.println("subclassMethod method in Subclass class");
    }
    @Override
    public void baseMethod() {
        System.out.println("basemethod in Subclass class");
    }
}
public class UpcastingDowncasting  {
    public static void main(String[] args) {
        // Upcasting :Referring to a subclass object with a superclass reference. Its always safe
        Base obj = new Subclass();

        // Calling baseMethod from the Subclass class
        obj.baseMethod(); // obj is a reference of type Base but points to an instance of Subclass

        // obj.subclassMethod(); // Error: obj is a Base class reference type, which does not have this method.

        // Downcasting: Casting a Base class reference to a Subclass type
        if (obj instanceof Subclass) {
            Subclass subobj = (Subclass) obj;
            subobj.subclassMethod(); // This will work after downcasting
        }
    } }
```

```
basemethod in Subclass class
subclassMethod method in Subclass class
```

Method Overriding

```
Class Figure {
   double dim1, dim2;
   Figure(double a, double b) {
      dim1 = a;
      dim2 = b;
   }
 double area() {
      System.out.println("Area for Figure is
undefined.");
      return 0;
   }
```

```
class Rectangle extends Figure {
   Rectangle(double a, double b) {
      super(a, b);
   }

   @Override
   double area() {
      System.out.println("Inside Area for Rectangle.");
      return dim1 * dim2;
   }
}
```

```
Class Triangle extends Figure {
   Triangle(double a, double b) {
      super(a, b);     }
   @Override
   double area() {
      System.out.println("Inside Area for Triangle.");
      return dim1 * dim2 / 2;
   } }
```

```
// Runtime polymorphism demo
      Figure f = new Figure(10, 10);
      Rectangle rect = new Rectangle(9, 5);
      Triangle t = new Triangle(10, 8);
      Figure figref;
      figref = rect;
      System.out.println("Area is " + figref.area());
      figref = t;
      System.out.println("Area is " + figref.area());
      figref = f;
      System.out.println("Area is " + figref.area());
   }
}
```

The dual mechanisms of inheritance and run-time polymorphism, it is possible to define one consistent interface that is used by several different, yet related, types of objects