COMPUTER CODING CLUB
MOTILAL NEHRU NATIONAL INSTITUTE OF TECHNOLOGY, PRAYAGRAJ

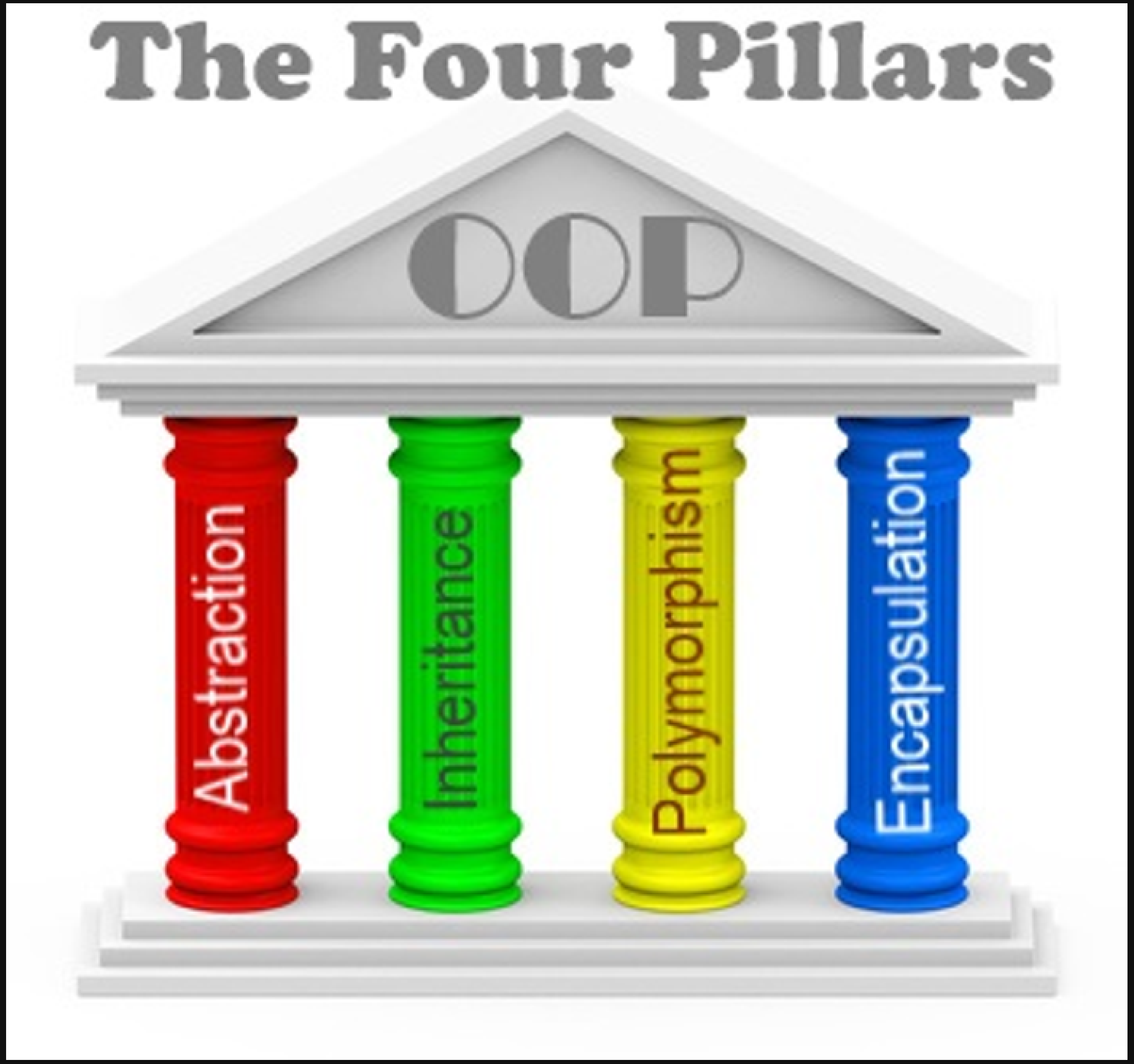
SOFTABLITZ

Pillars of Object Oriented Programming and related concepts

Why is the filename same as classname?

- The filename must have the same name as the public class name in that file, which is the way to tell the JVM that this is an entry point.
- Suppose when we create a program in which more than one class resides and after compiling a java source file, it will generate the same number of the .class file as classes reside in our program. In this condition, we will not able to easily identify which class need to interpret by java interpreter and which class contains the Entry point for the program.
- As a consequence of above point we can have **only one public class in a single java file**. If you want yous software to have multiple public classes then they have to be in different files.
- As a practice what we do is keep only the main class (Class having main function) Public for JVM to access and rest of the classes are kept as default access specifies or private until and unless it needs to be public.

Pillars of Object Oriented Programming



ABSTRACTION

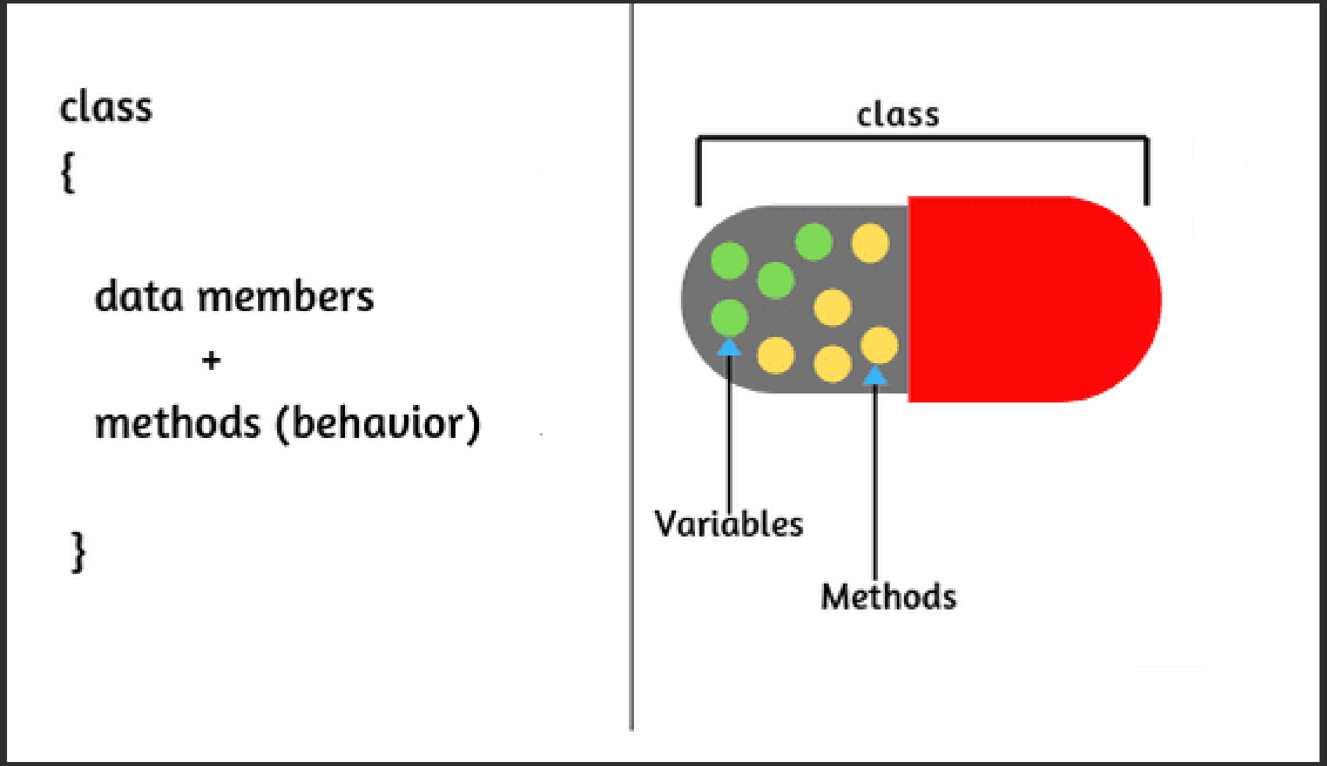
- Abstraction is the process of taking away or removing characteristics from something in order to reduce it to a set of essential characteristics.

- Abstraction means only giving away the relevant information and hiding the rest.

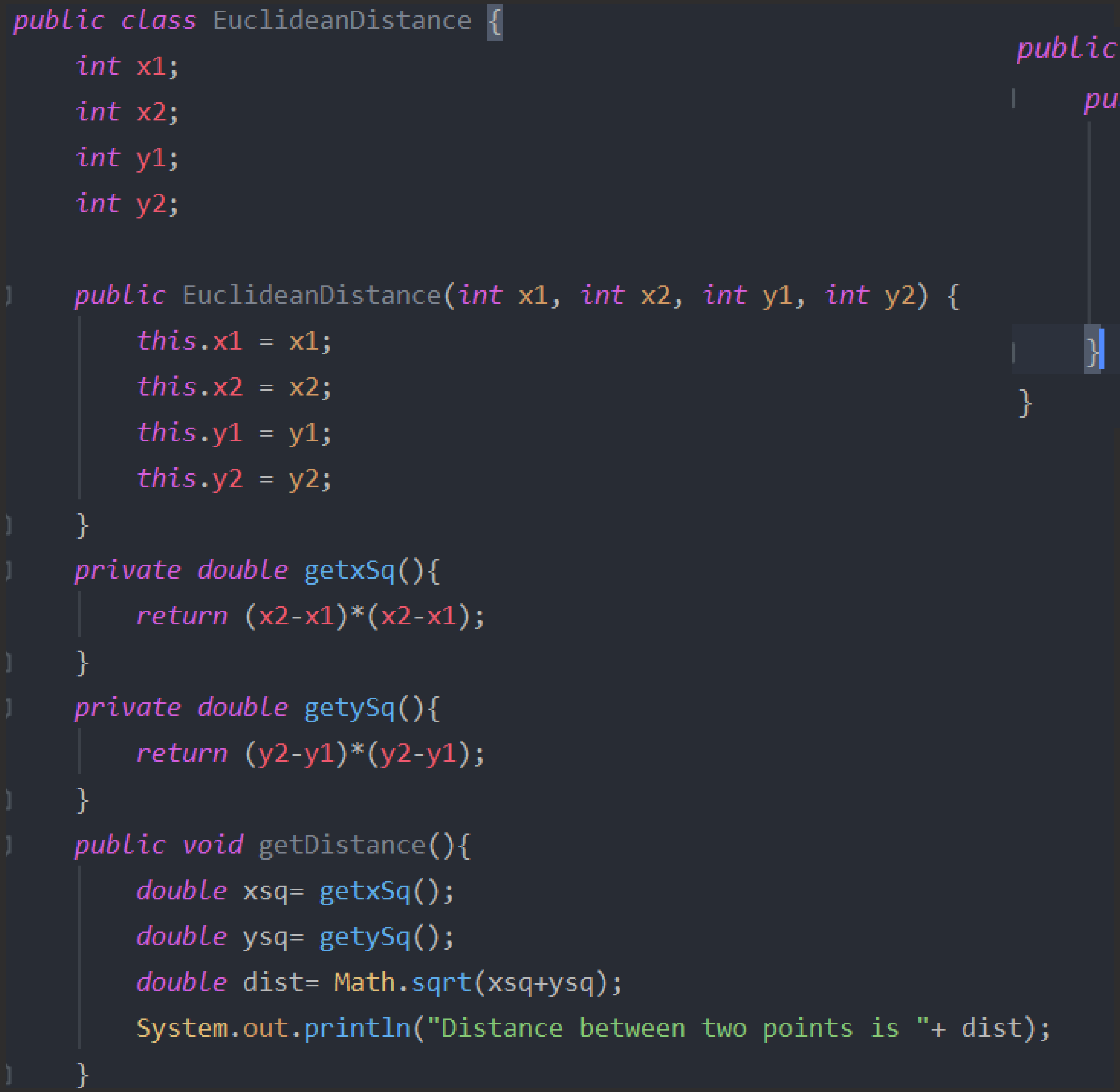
- Through the process of abstraction, a programmer hides all but the relevant data about an object in order to reduce complexity and increase efficiency.

ENCAPSULATION

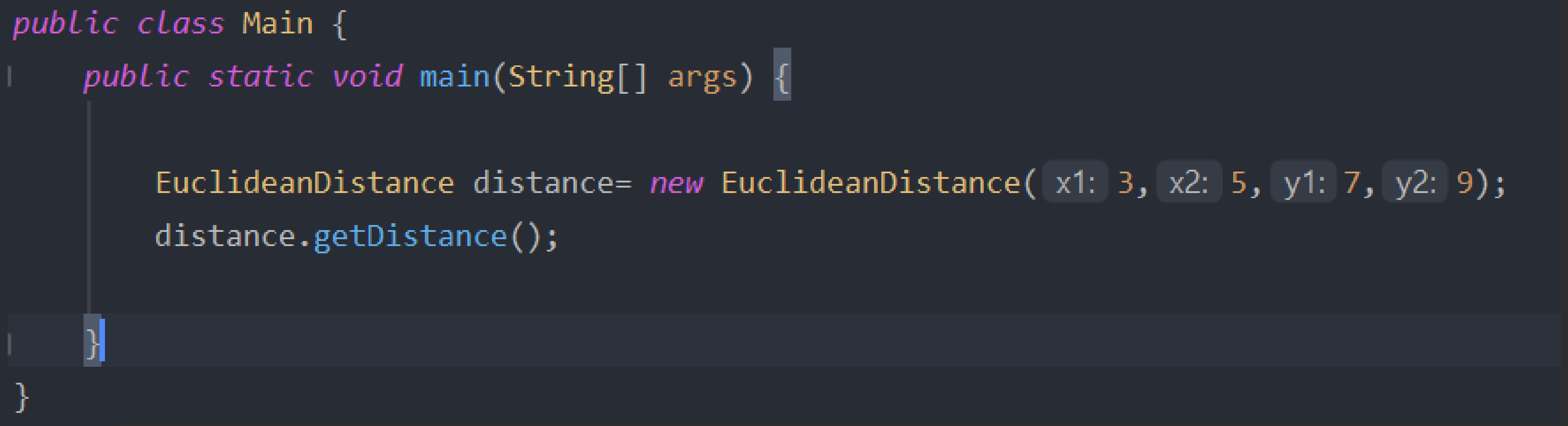
- Encapsulation in Java and many OOP languages is the process of wrapping information and functionality in a class and providing methods of accessing them in order to provide a simple way for users to access the information and make use of the functionality of the class.
- Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse. For example, allowing access to data only through a welldefined set of methods, you can prevent the misuse of that data.
- A class creates a black box which may be used, but the inner workings of which are not open to tampering.



Abstraction using encapsulation



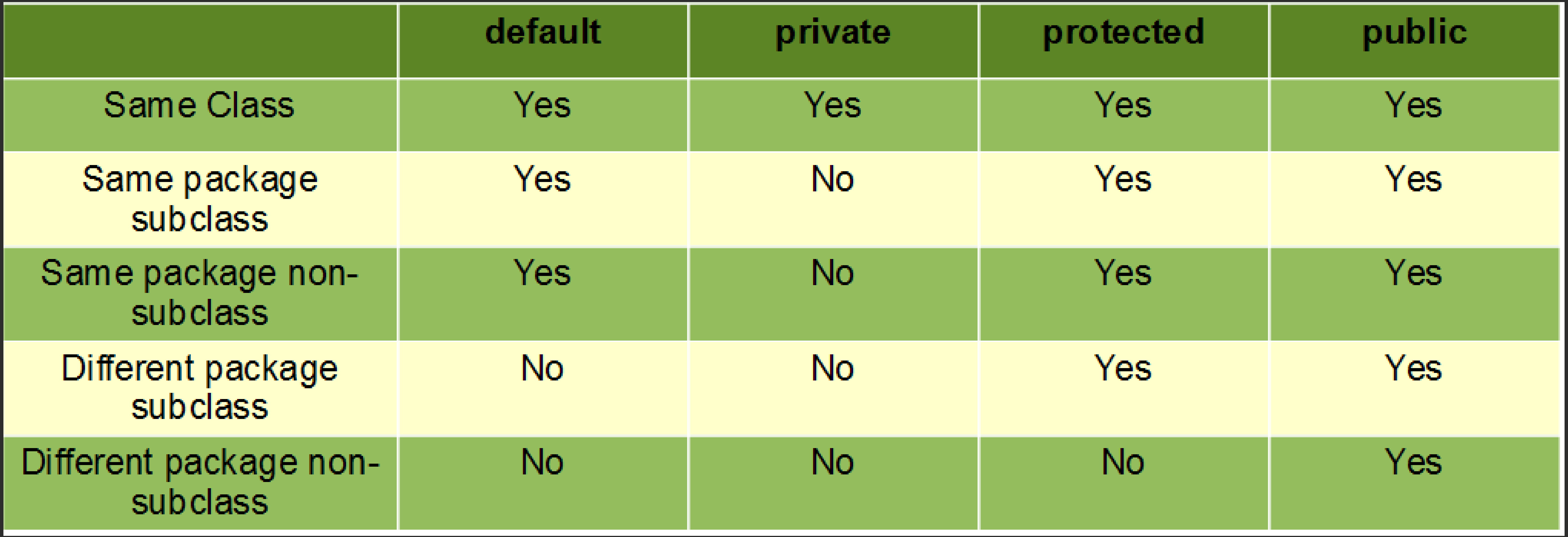
```java
public class EuclideanDistance {
    int x1;
    int x2;
    int y1;
    int y2;

    public EuclideanDistance(int x1, int x2, int y1, int y2) {
        this.x1 = x1;
        this.x2 = x2;
        this.y1 = y1;
        this.y2 = y2;
    }
    private double getxSq(){
        return (x2-x1)*(x2-x1);
    }
    private double getySq(){
        return (y2-y1)*(y2-y1);
    }
    public void getDistance(){
        double xsq= getxSq();
        double ysq= getySq();
        double dist= Math.sqrt(xsq+ysq);
        System.out.println("Distance between two points is "+ dist);
    }
```



```java
public class Main {
    public static void main(String[] args) {

        EuclideanDistance distance= new EuclideanDistance( x1: 3, x2: 5, y1: 7, y2: 9);
        distance.getDistance();

    }
}
```

- The user is only concerned with providing the input coordinates and getting the distance as the output.
- The inner working of how the distance is calculated (using getxSq and getySq methods) is hidden from the user.
- This way the complexity from the user's point of view is reduced using encapsulation.

Access Specifiers



	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Static Keyword

- When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.
- You can declare both methods and variables to be static. The most common example of a static member is main( ). main( ) is declared as static because it must be called before any objects exist.
- When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.
- Outside of the class in which they are defined, static methods and variables can be used independently of any object.
- For Static Methods, - They can only directly call other static methods. - They can only directly access static data. - They cannot refer to this or super in any way.

Final Keyword

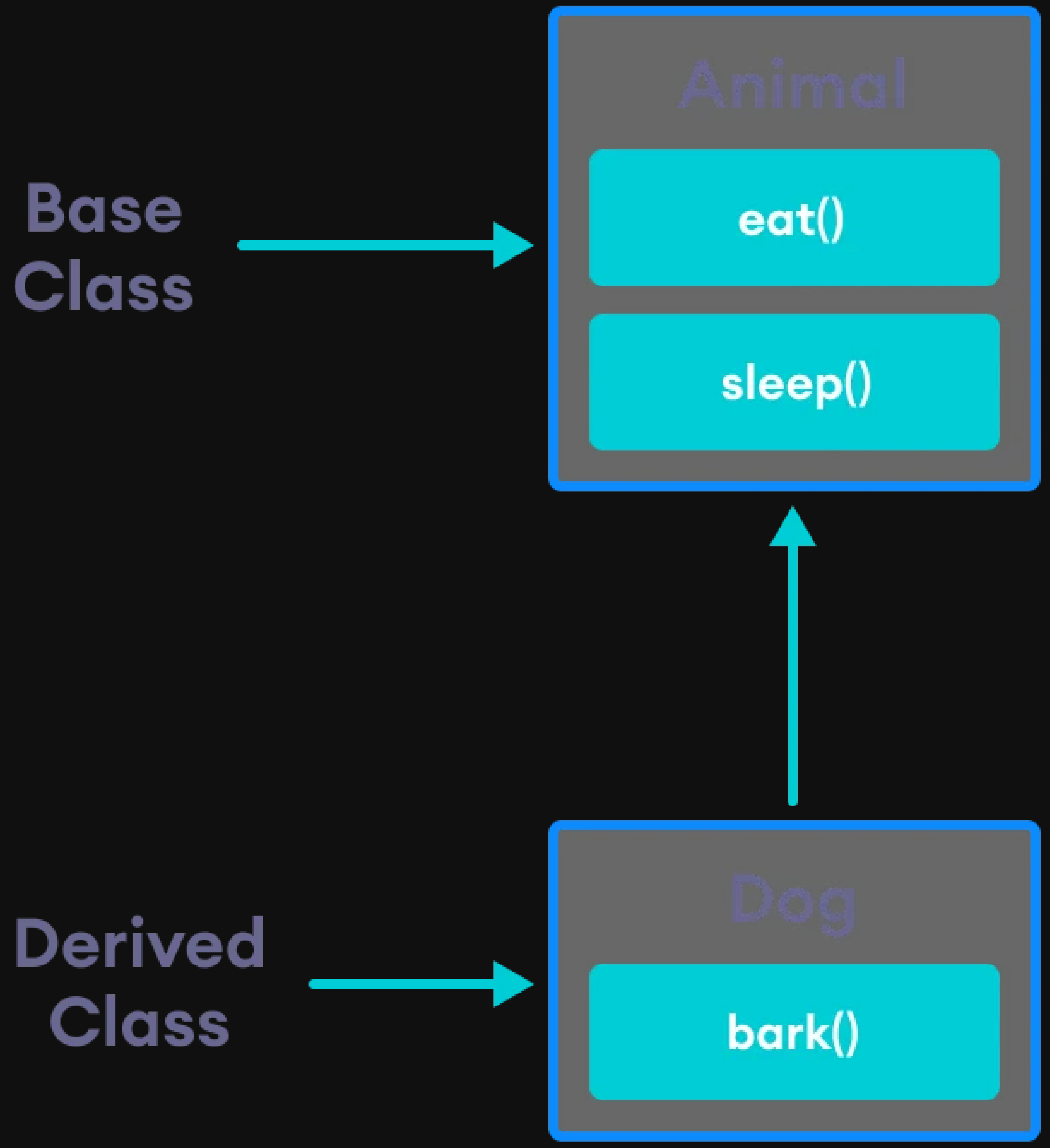
- A field can be declared as final. Doing so prevents its contents from being modified.
- You must initialize a final field when it is declared. You can do this in one of two ways:
    - First, you can give it a value when it is declared.
    - Second, you can assign it a value within a constructor.

Getter and Setter

- Access to class variables is controlled using getters and setters.
- Getters are used to read variable values.
- Setters are used to modify variable values.
- We can selectively provide getters to only expose variables that are meant to be seen by the user of our program.
- Similarly, we can provide sanity logic in setters to make sure our variables are never filled with invalid values.

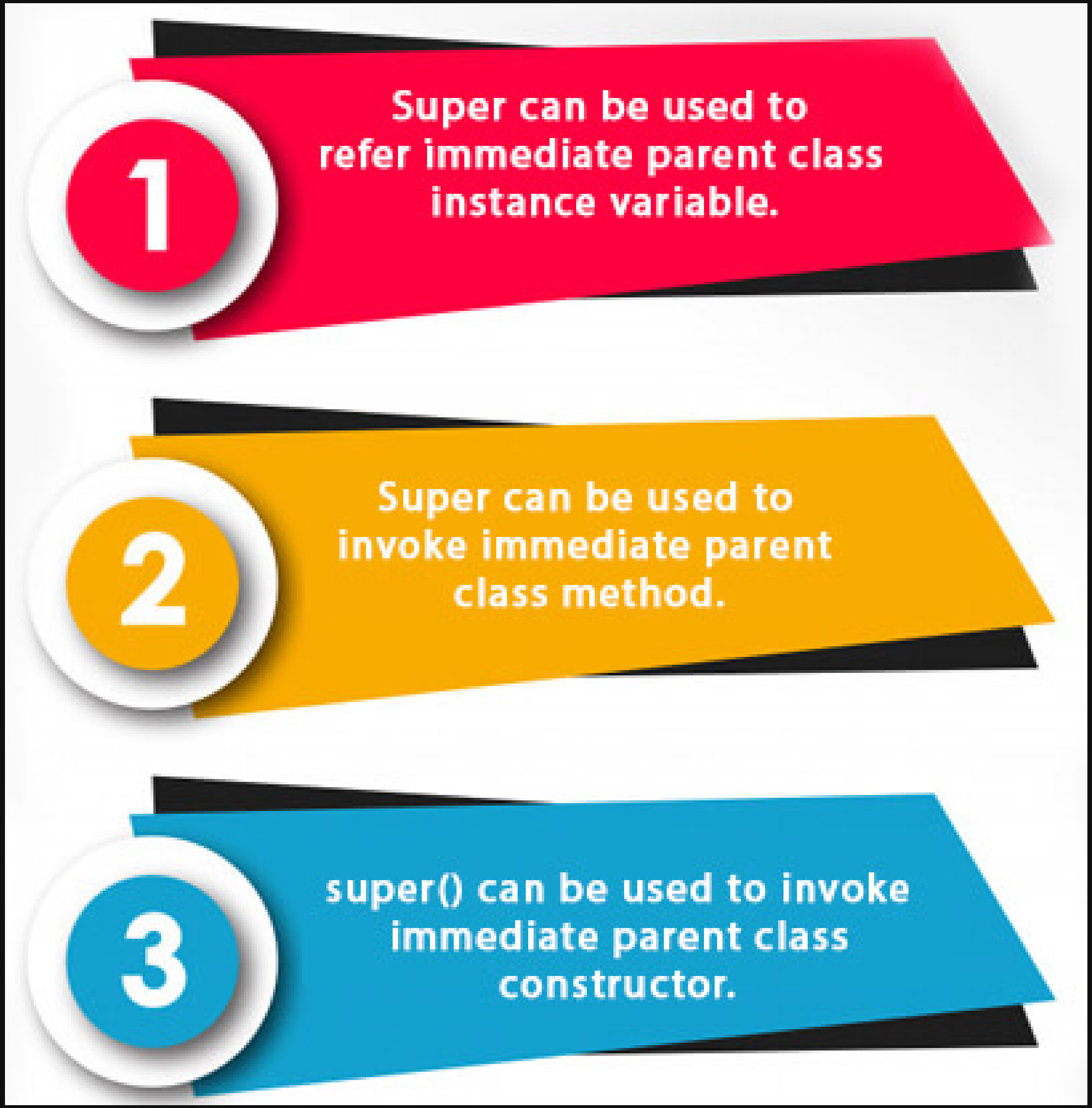
INHERITANCE

- Inheritance is an important pillar of OOP.
- It is the mechanism in java by which one class is allowed to inherit the features (fields and methods) of another class.
- To inherit a class, you simply incorporate the definition of one class into another by using the extends keyword.
- Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as private.
- Once you have created a superclass that defines the general aspects of an object, that superclass can be inherited to form specialized classes.



Super Keyword

- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword super.



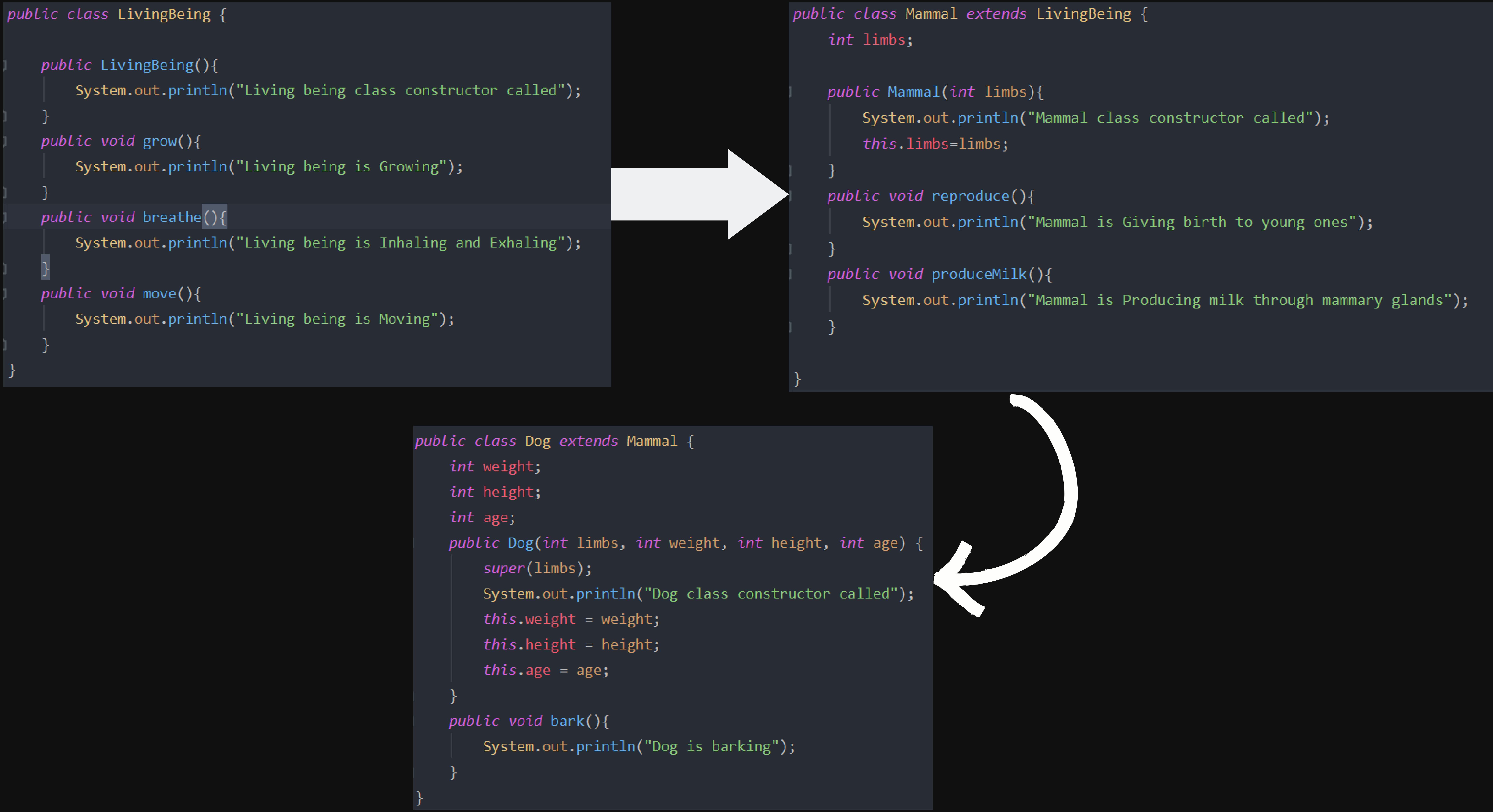
1 Super can be used to refer immediate parent class instance variable.

2 Super can be used to invoke immediate parent class method.

3 super() can be used to invoke immediate parent class constructor.

Super Keyword

- In a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass.
- Java enforces that the call to super (explicit or not) must be the first statement in the constructor.
- This is to prevent the subclass part of the object from being initialized before the superclass part of the object being initialized.
- If super( ) is not explicitly typed out, then the default or parameterless constructor of each superclass will be executed.

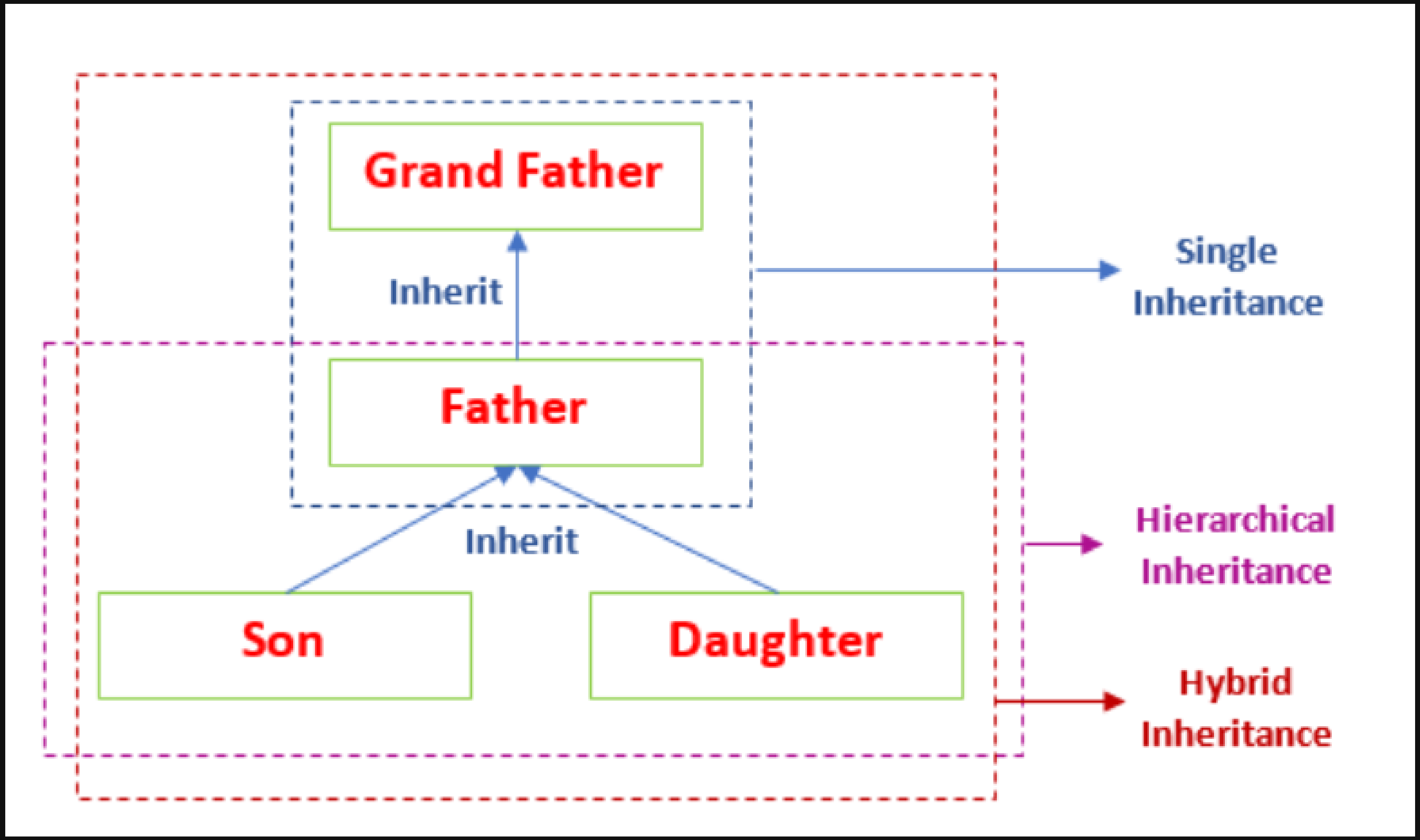


```java
public class LivingBeing {

    public LivingBeing(){
        System.out.println("Living being class constructor called");
    }
    public void grow(){
        System.out.println("Living being is Growing");
    }
    public void breathe(){
        System.out.println("Living being is Inhaling and Exhaling");
    }
    public void move(){
        System.out.println("Living being is Moving");
    }
}
```

```java
public class Mammal extends LivingBeing {
    int limbs;

    public Mammal(int limbs){
        System.out.println("Mammal class constructor called");
        this.limbs=limbs;
    }
    public void reproduce(){
        System.out.println("Mammal is Giving birth to young ones");
    }
    public void produceMilk(){
        System.out.println("Mammal is Producing milk through mammary glands");
    }

}
```

```java
public class Dog extends Mammal {
    int weight;
    int height;
    int age;
    public Dog(int limbs, int weight, int height, int age) {
        super(limbs);
        System.out.println("Dog class constructor called");
        this.weight = weight;
        this.height = height;
        this.age = age;
    }
    public void bark(){
        System.out.println("Dog is barking");
    }
}
```

Types of Inheritance in Java

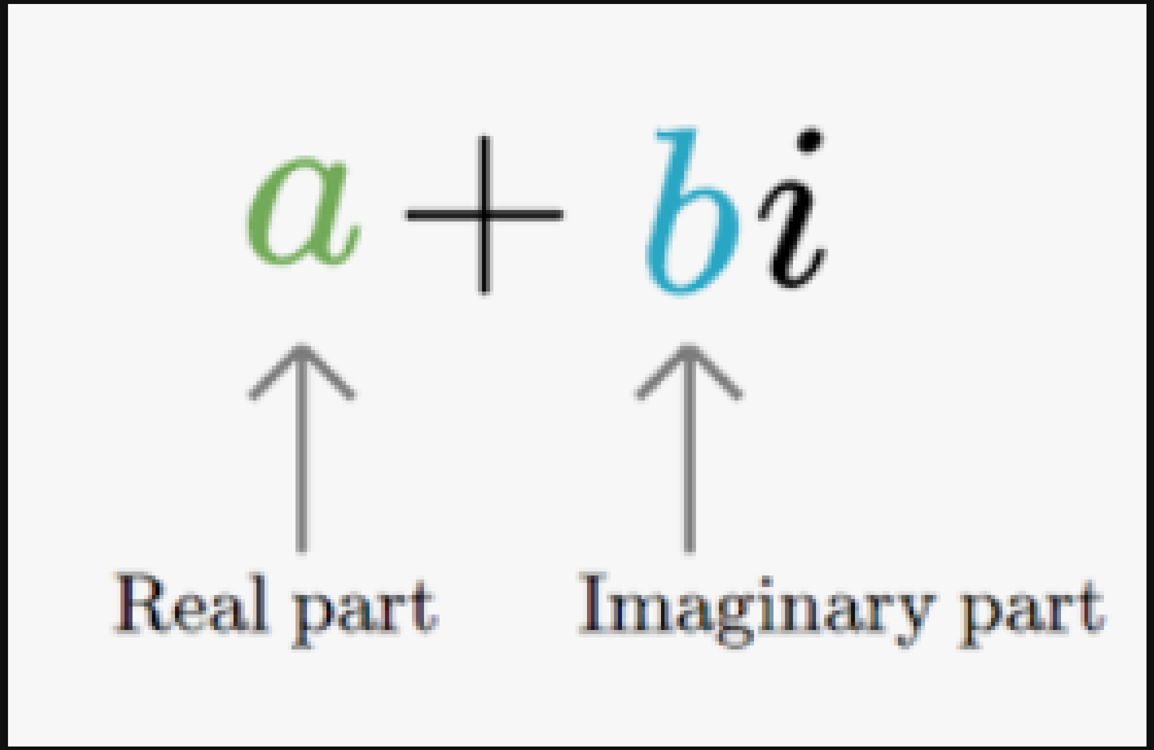


Object Class

- Object class is the parent of every class in Java. All other classes are subclasses of Object. That is, Object is a superclass of all other classes.
- A reference variable of type Object can refer to an object of any other class.
- The Object class provides some common behaviors to all the objects such as object can be compared, object can be described in String format, etc.
- For this description, it provides methods such as equals(), toString(), etc.

Object Class: equals()

- equals( ) method compares two objects.
- It returns true if the objects are equal, and false otherwise.
- In the Object class implementation, for any non-null reference values x and y , this method returns true if and only if x and y refer to the same object.
- You can override the equals() method in Object class and give your own definition for equality. Let's see an example.

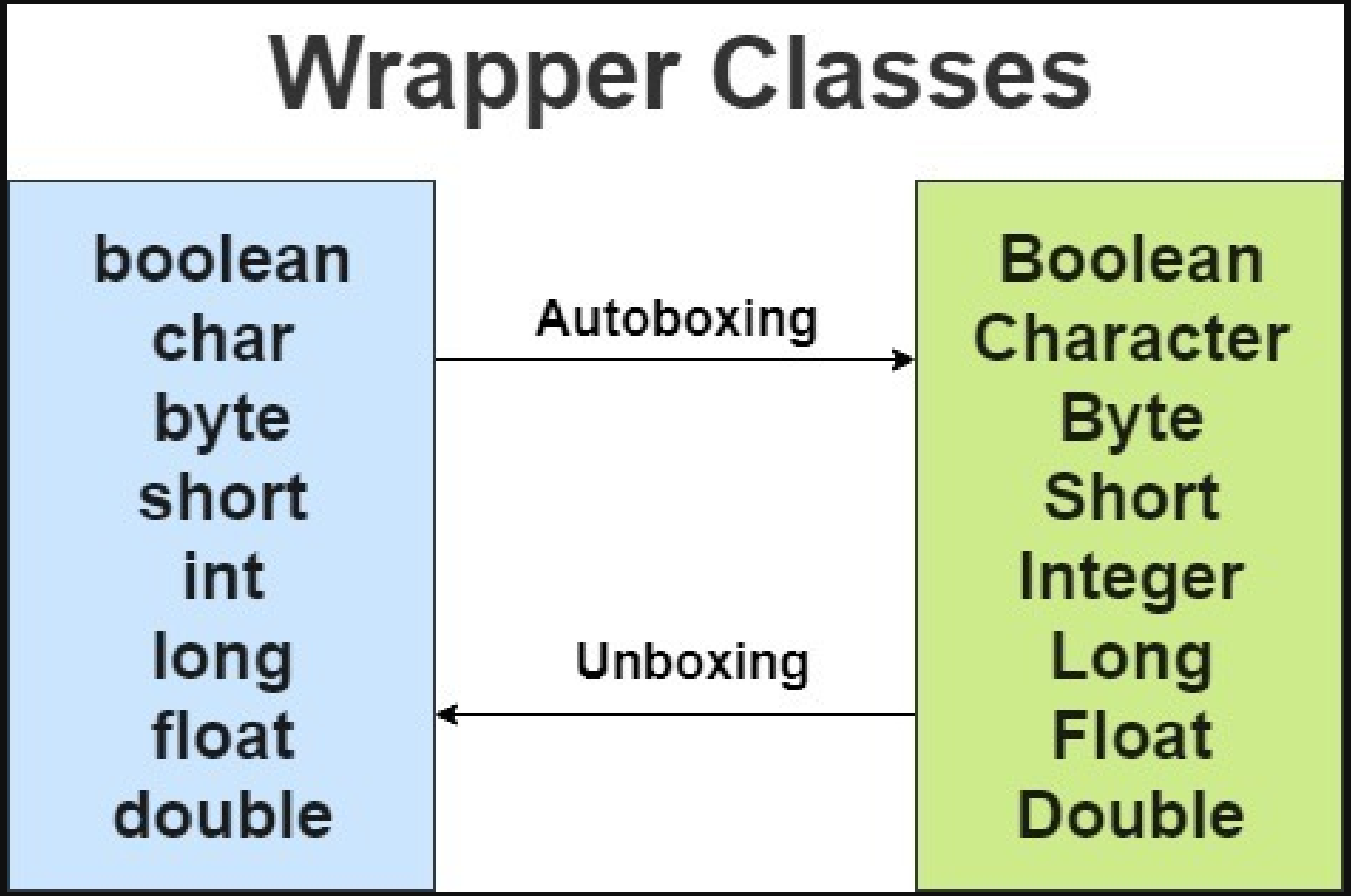


Object Class: toString()

- toString( ) method returns a string that contains a description of the object on which it is called.
- Also, this method is automatically called when an object is output using println().
- Many classes override this method. Doing so allows them to tailor a description specifically for the types of objects that they create.
- Let's take the example of a Student class.

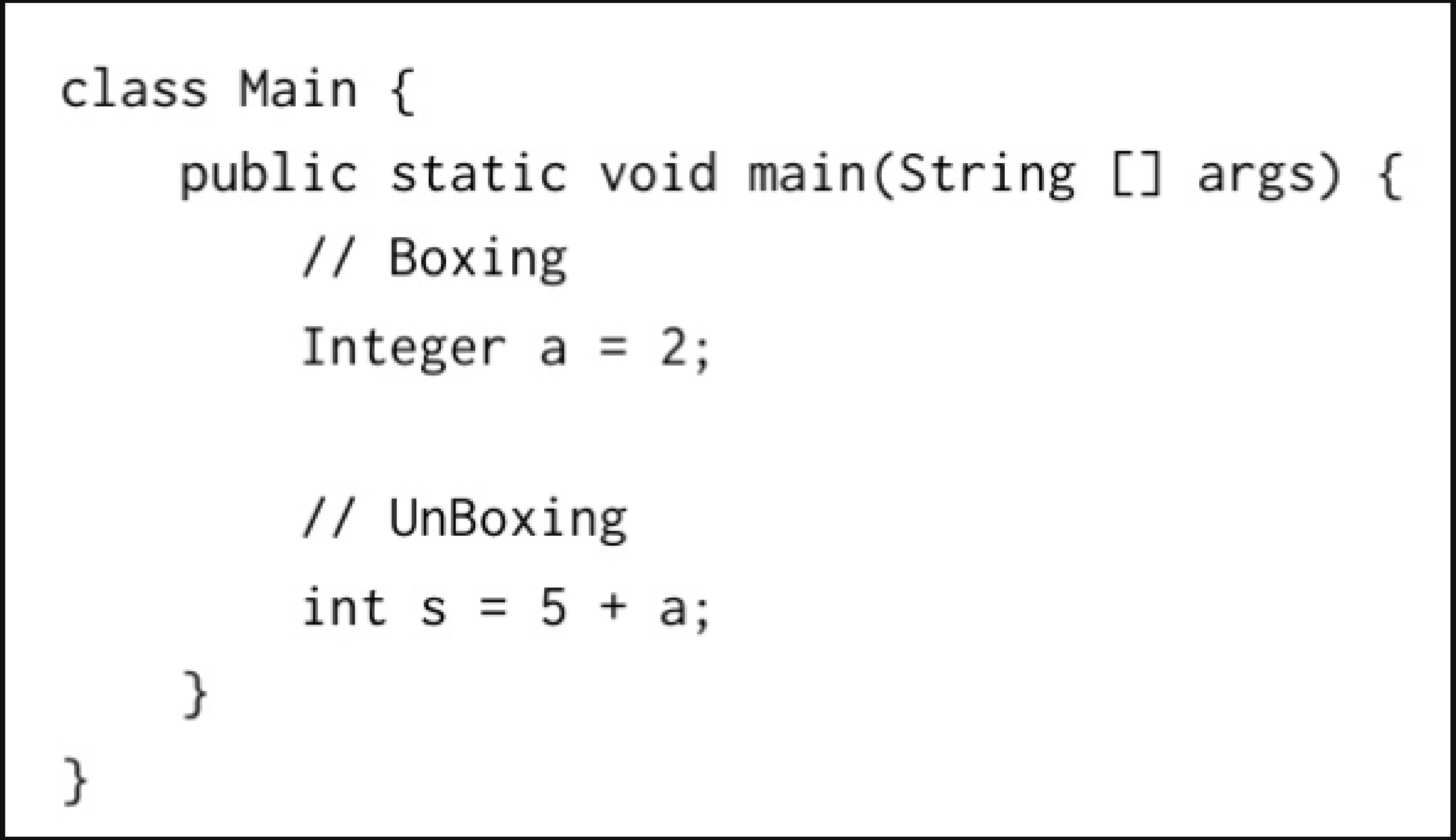
Wrapper Class

- Java provides type wrappers, which are classes that encapsulate a primitive type within an object.
- These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.



Autoboxing and Unboxing

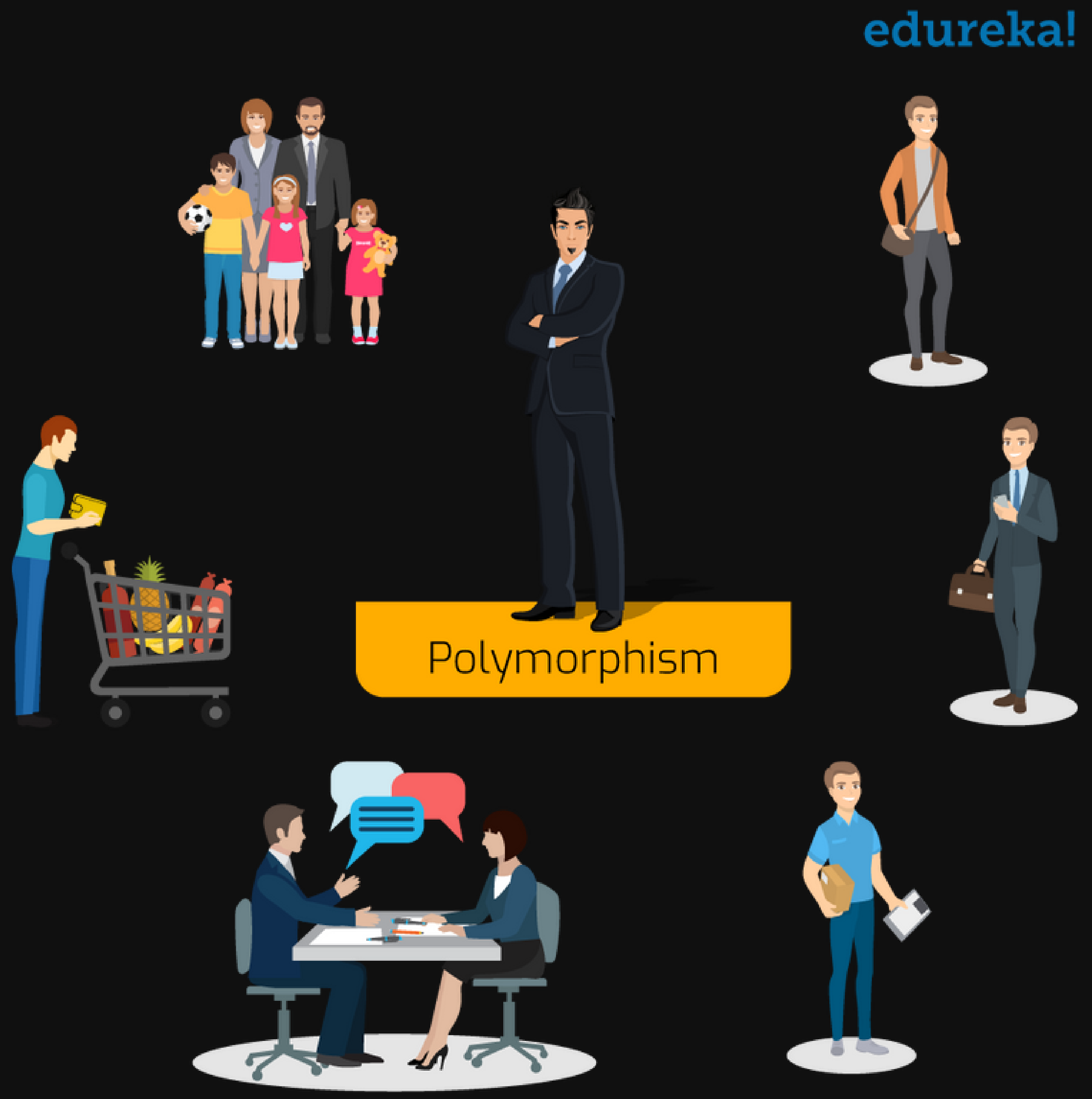
The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing.



```
class Main {
    public static void main(String [] args) {
        // Boxing
        Integer a = 2;

        // UnBoxing
        int s = 5 + a;
    }
}
```

POLYMORPHISM

- Polymorphism in it's literal sense means "many forms".
- In simple words it means performing an action in different ways. Similarly an object behaves differently in different situations.
- Inheritance and polymorphism go hand in hand.
- To know whether an object is polymorphic, you can perform a simple test. If the object successfully passes multiple is-a or instanceof tests, it's polymorphic.



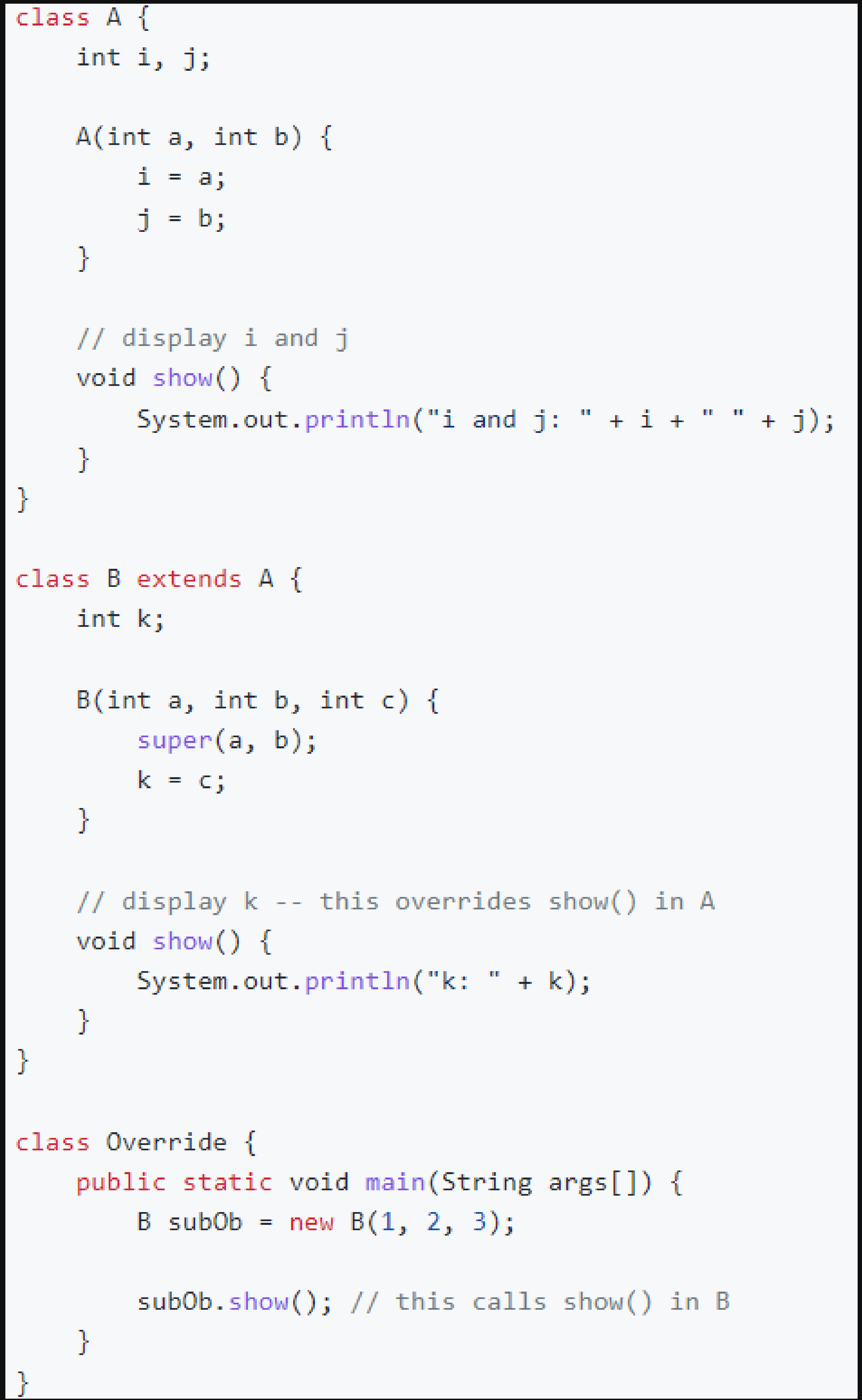
Question

- Every instance/object in Java is polymorphic. True or False?

Run time polymorphism

- **Method overriding:** In a class hierarchy, when 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.

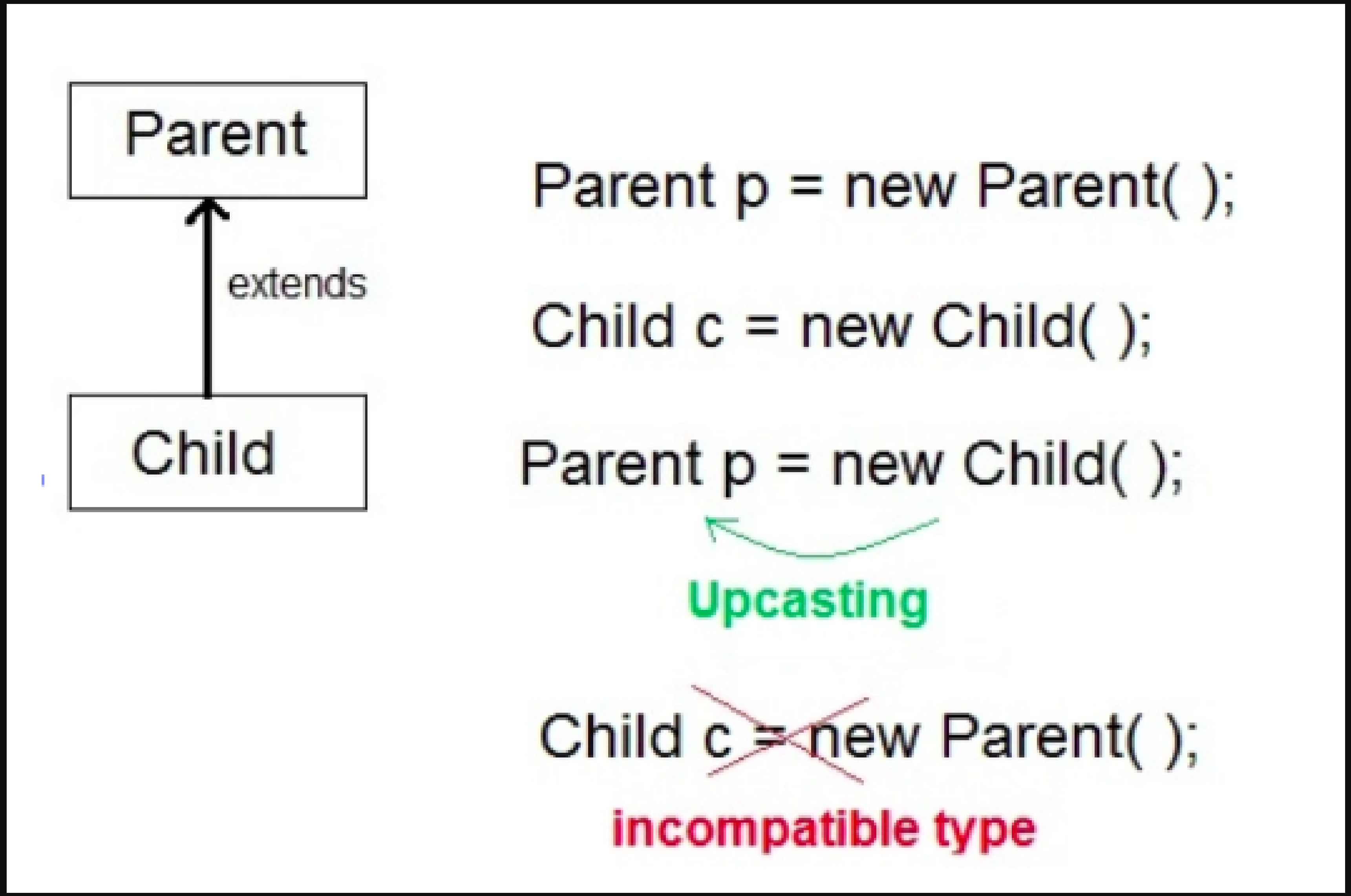
- In the given example the *show()* method is overridden by B, where it provides it's own implementation.



```java
class A {
    int i, j;

    A(int a, int b) {
        i = a;
        j = b;
    }

    // display i and j
    void show() {
        System.out.println("i and j: " + i + " " + j);
    }
}

class B extends A {
    int k;

    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    // display k -- this overrides show() in A
    void show() {
        System.out.println("k: " + k);
    }
}

class Override {
    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);

        subOb.show(); // this calls show() in B
    }
}
```

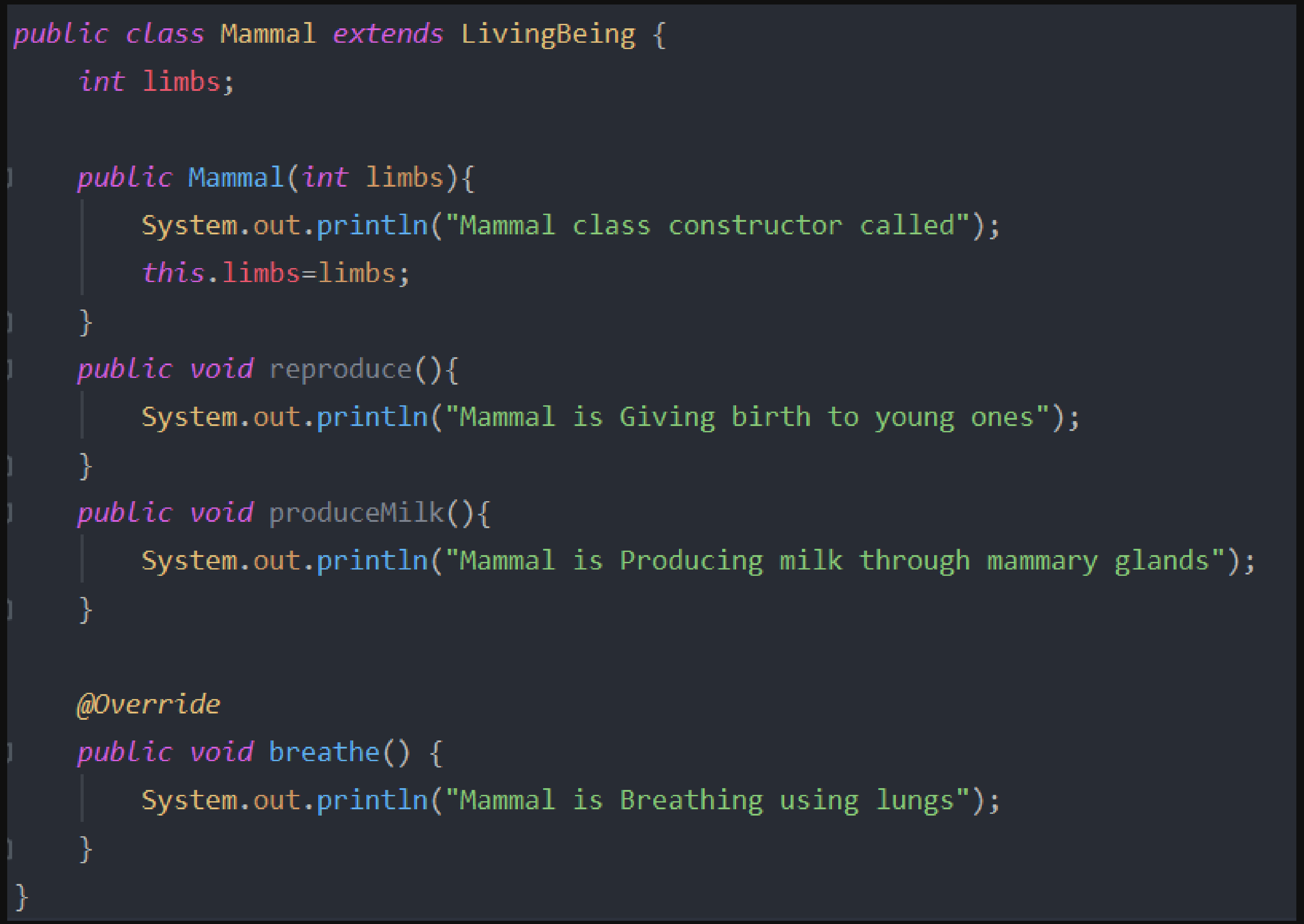
Dynamic method dispatch

- Mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- This is how Java implements run-time polymorphism.
- A superclass reference variable can refer to a subclass object.
- It is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.



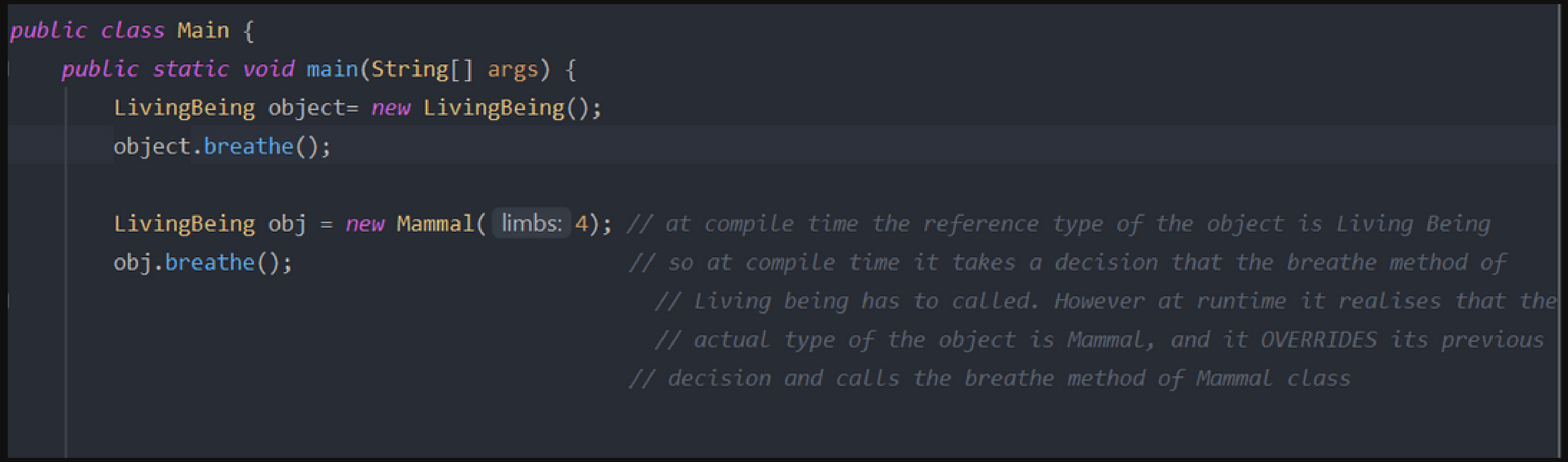
```
Parent p = new Parent( );
Child c = new Child( );
Parent p = new Child( );
Child c = new Parent( );
```

- Consider the previous example of Living beings and mammals.
- In mammals we have overriden the breathe method declared in Living beings (parent class)

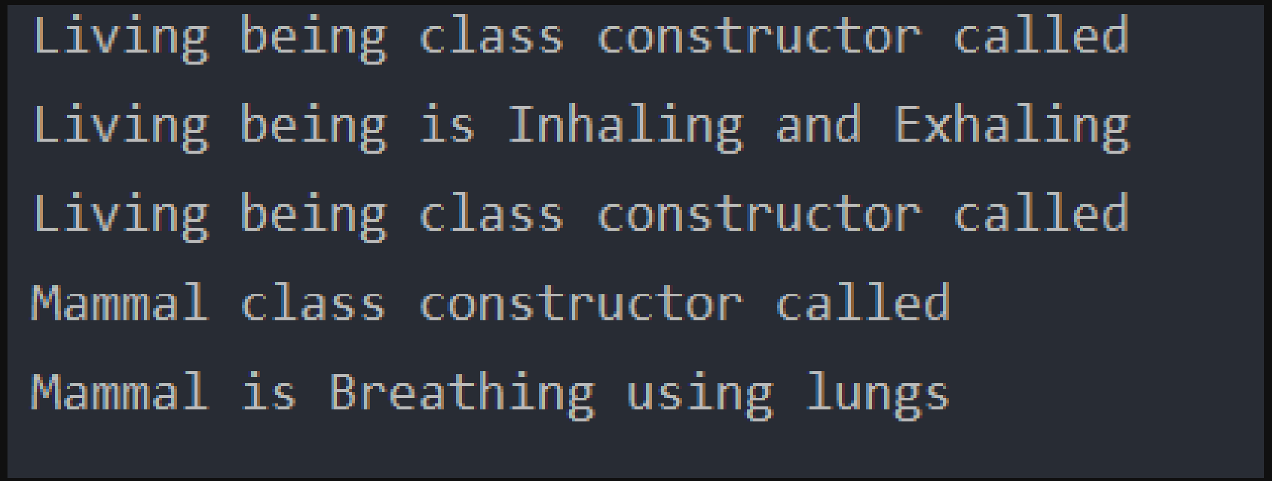


```java
public class Mammal extends LivingBeing {
    int limbs;

    public Mammal(int limbs){
        System.out.println("Mammal class constructor called");
        this.limbs=limbs;
    }
    public void reproduce(){
        System.out.println("Mammal is Giving birth to young ones");
    }
    public void produceMilk(){
        System.out.println("Mammal is Producing milk through mammary glands");
    }

    @Override
    public void breathe() {
        System.out.println("Mammal is Breathing using lungs");
    }
}
```

- In the main method, first the breathe method is called twice and it gives different results.
- This is how run-time polymorphism is achieved.



```java
public class Main {
    public static void main(String[] args) {
        LivingBeing object= new LivingBeing();
        object.breathe();

        LivingBeing obj = new Mammal( limbs: 4); // at compile time the reference type of the object is Living Being
        obj.breathe();                           // so at compile time it takes a decision that the breathe method of
                                                   // Living being has to called. However at runtime it realises that the
                                                   // actual type of the object is Mammal, and it OVERRIDES its previous
                                                 // decision and calls the breathe method of Mammal class
```

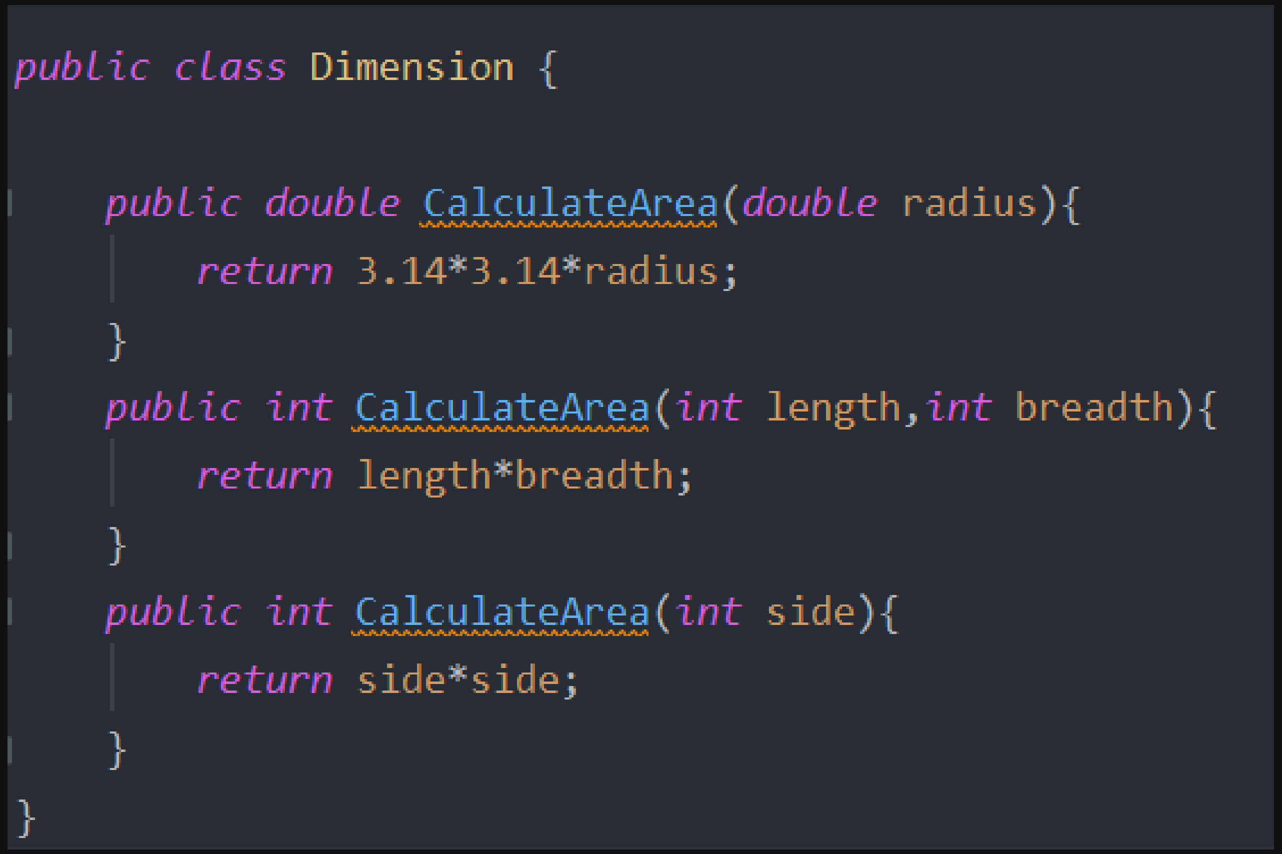


```
Living being class constructor called
Living being is Inhaling and Exhaling
Living being class constructor called
Mammal class constructor called
Mammal is Breathing using lungs
```

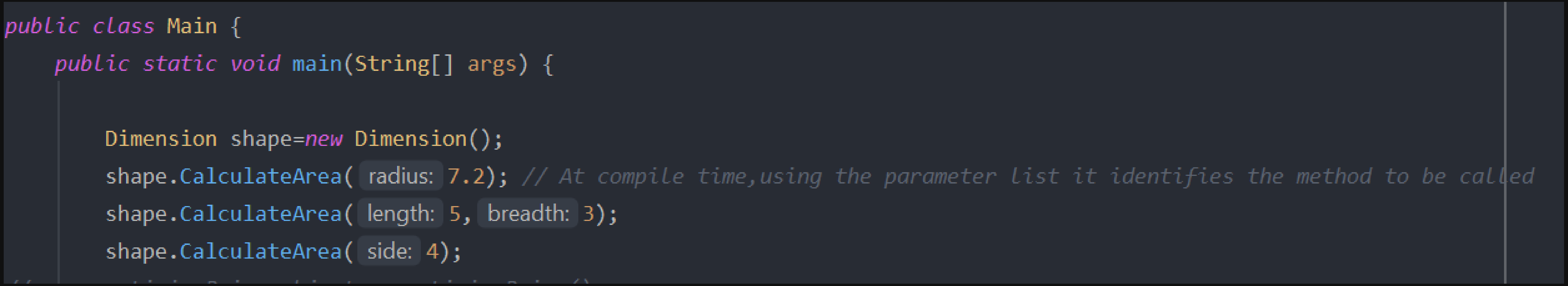
Compile time polymorphism

- When there are multiple functions with the same name but different method signature then these functions are said to be overloaded.

- Method signature can differ by the number of input parameters or type of input parameters, or a mixture of both.



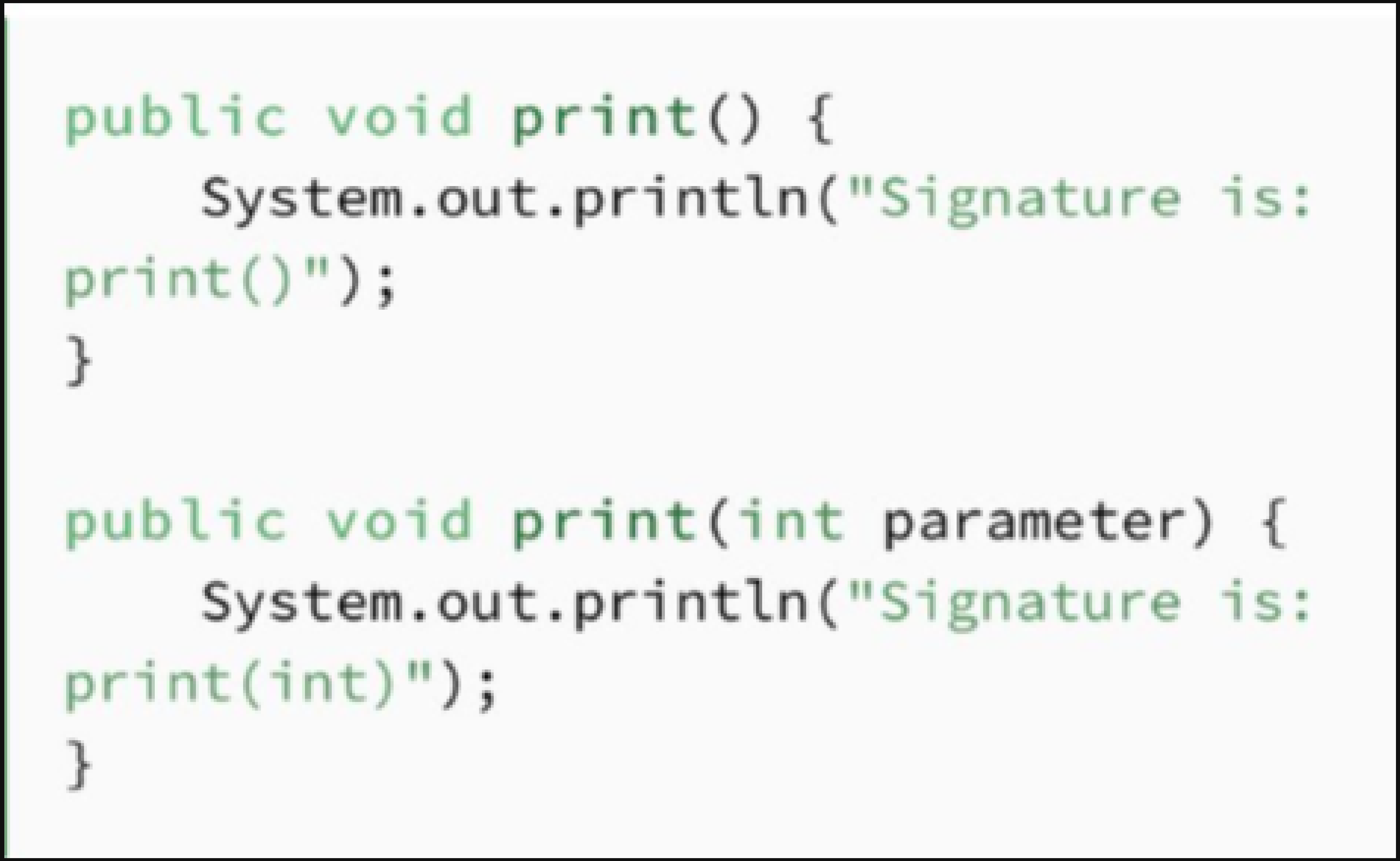
```java
public class Dimension {

    public double CalculateArea(double radius){
        return 3.14*3.14*radius;
    }
    public int CalculateArea(int length,int breadth){
        return length*breadth;
    }
    public int CalculateArea(int side){
        return side*side;
    }
}
```



```java
public class Main {
    public static void main(String[] args) {

        Dimension shape=new Dimension();
        shape.CalculateArea( radius: 7.2); // At compile time,using the parameter list it identifies the method to be called
        shape.CalculateArea( length: 5, breadth: 3);
        shape.CalculateArea( side: 4);
```

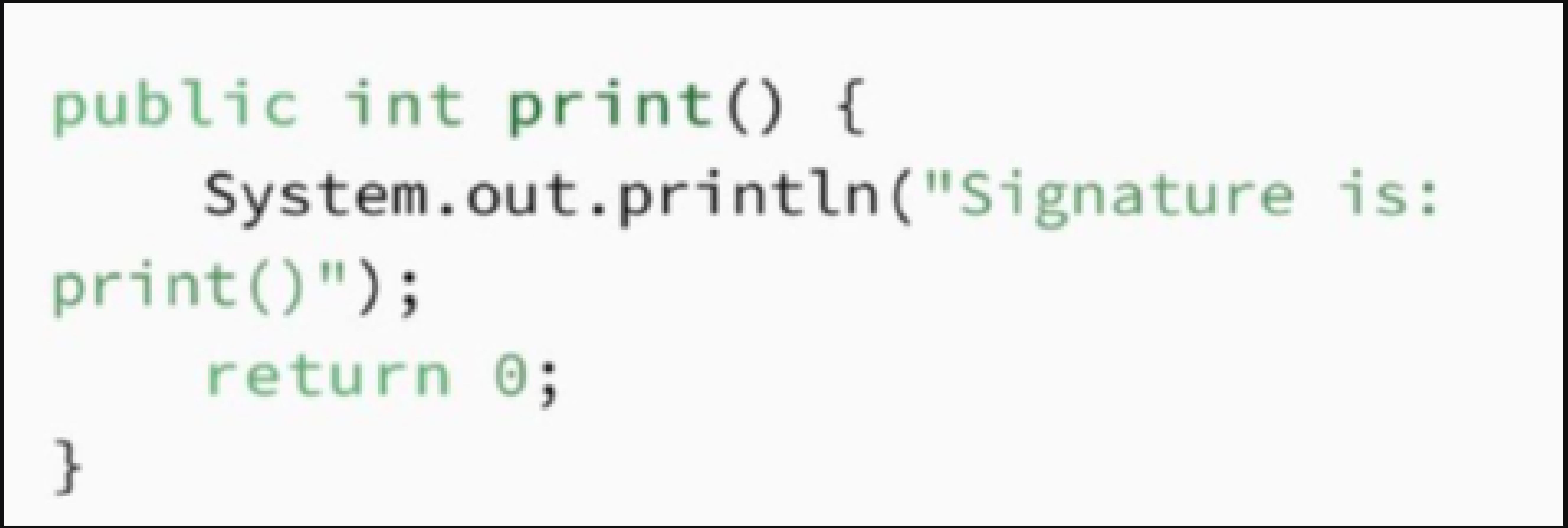
Will the following codes compile?

1:



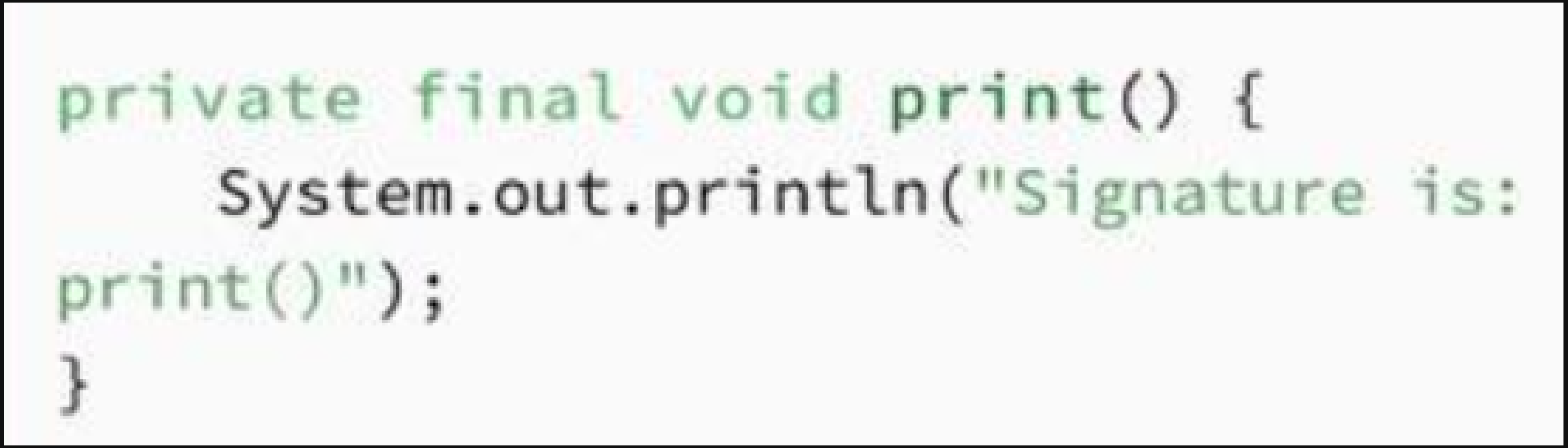
```
public void print() {
    System.out.println("Signature is:
print()");
}

public void print(int parameter) {
    System.out.println("Signature is:
print(int)");
}
```

2:



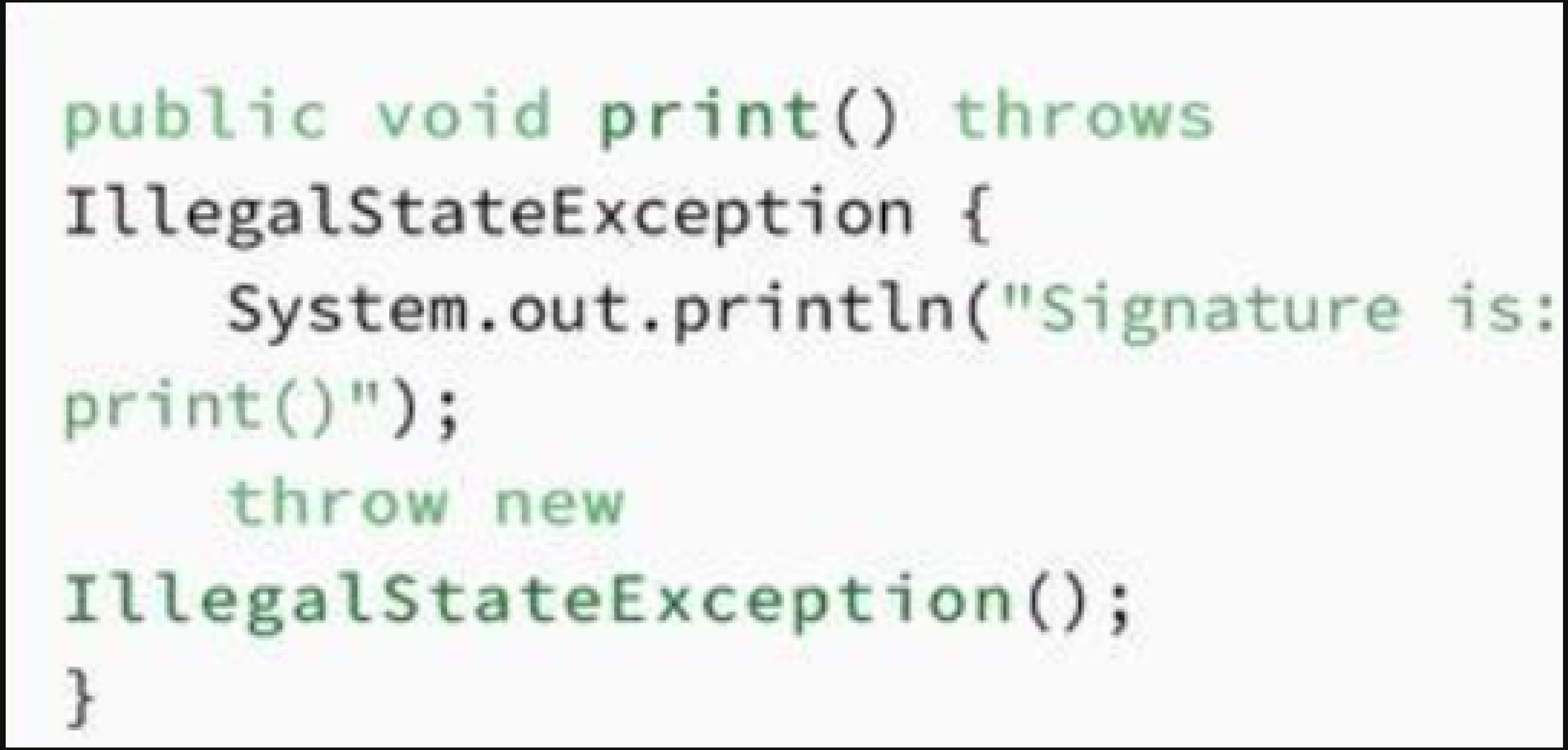
```
public int print() {
    System.out.println("Signature is:
print()");
    return 0;
}
```

3:



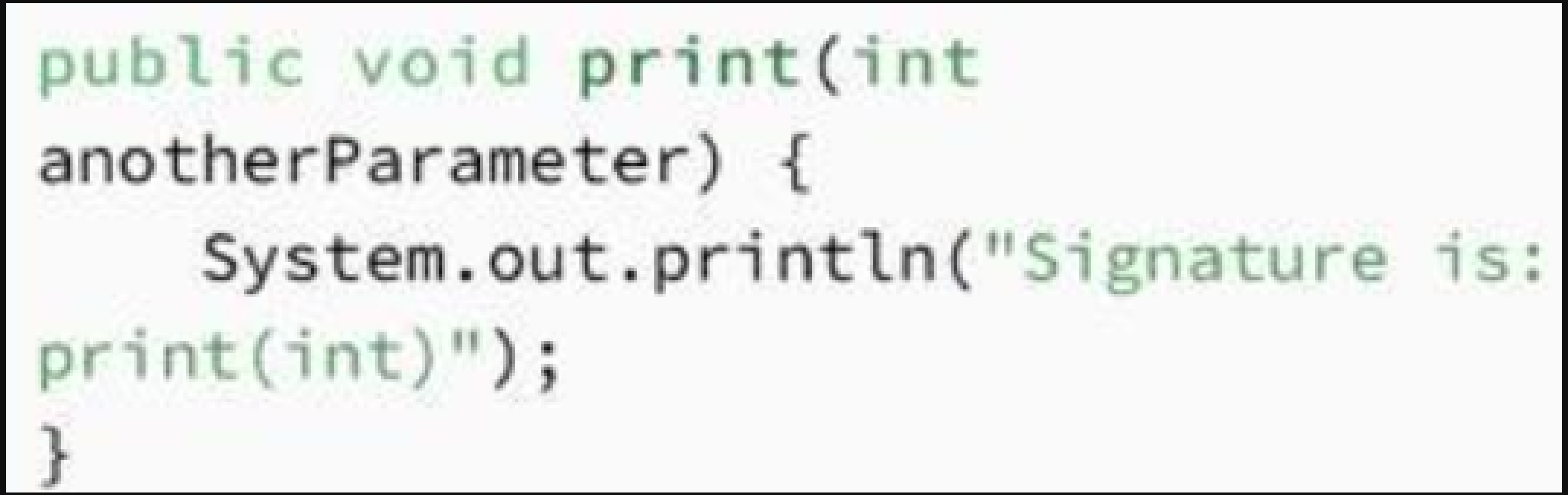
```
private final void print() {
    System.out.println("Signature is:
print()");
}
```

4:

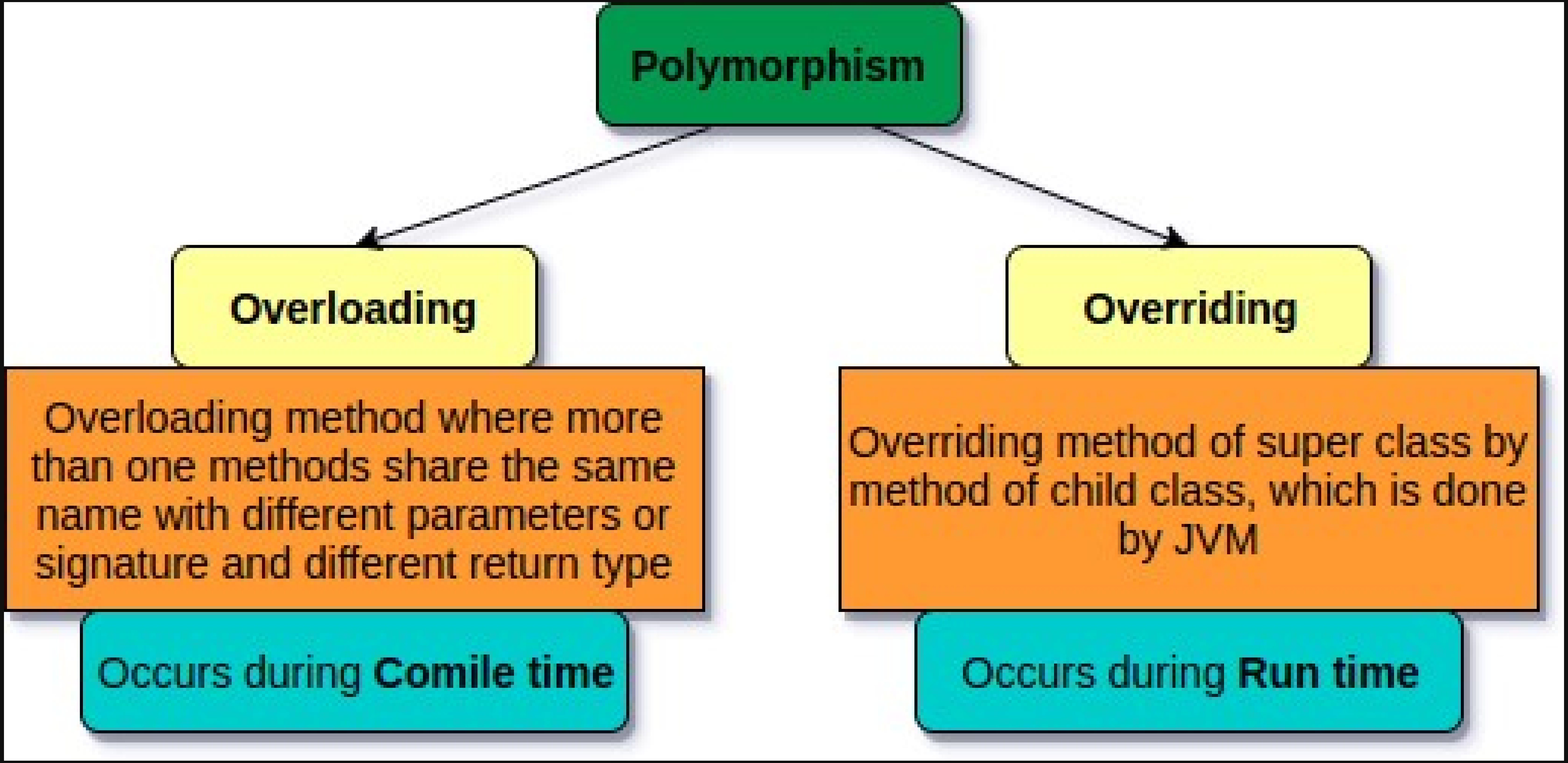


```
public void print() throws
IllegalStateException {
    System.out.println("Signature is:
print()");
    throw new
IllegalStateException();
}
```

5:

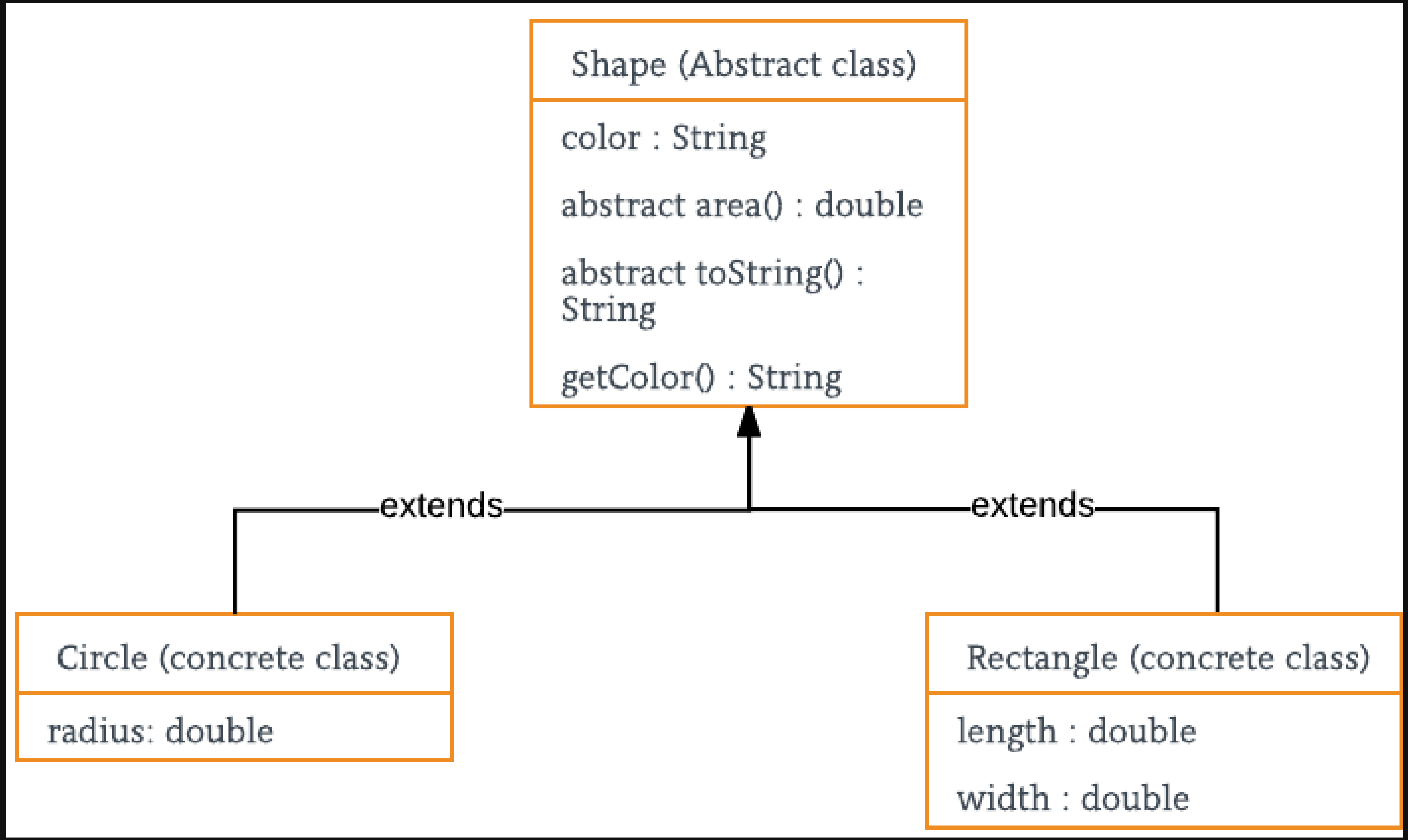


```
public void print(int
anotherParameter) {
    System.out.println("Signature is:
print(int)");
}
```


Polymorphism
Overloading
Overriding
Overloading method where more than one methods share the same name with different parameters or signature and different return type
Overriding method of super class by method of child class, which is done by JVM
Occurs during **Comile time**
Occurs during **Run time**

Abstract Class

- A class which is declared with the abstract keyword is known as an abstract class in Java.
- It can have abstract and non-abstract methods (method with the body).
- It needs to be extended and its method implemented.
- It cannot be instantiated.
- It can have constructors and static methods also.



Interface

- An interface in Java is a blueprint of a class. It has static constants and abstract methods. It cannot have a method body.
- It is used to achieve abstraction and multiple inheritance in Java.
- It cannot be instantiated.
- A method declared in an interface is by default abstract.

- Since Java 8, we can have default and static methods in an interface.
- Since Java 9, we can have private methods in an interface.

```
public interface GPS {
    public void getCoordinates();
}

public interface Radio {
    public void startRadio();
    public void stopRadio();
}

public class Smartphone implements GPS,Radio {
    public void getCoordinates() {
        // return some coordinates
    }
    public void startRadio() {
      // start Radio
    }
    public void stopRadio() {
        // stop Radio
    }
}
```