Object Oriented Programming with Java

Zheng Chen

Inheritance

1. Inheritance
2. Overriding and Hiding
3. null, this, and super
4. Polymorphism
5. final and abstract
6. Object class

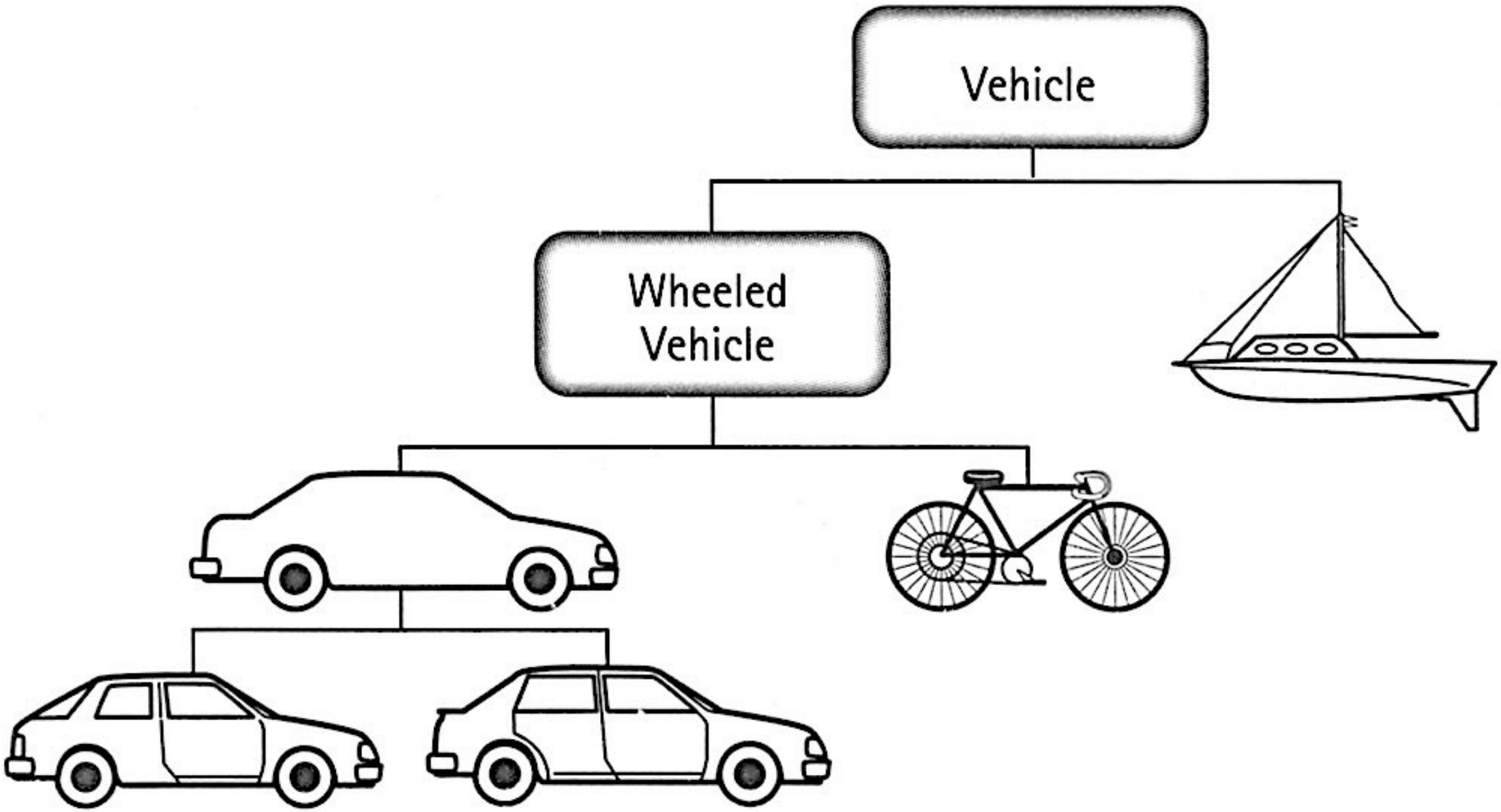
Inheritance

- Inheritance in OOP is a mechanism in which one object acquires all the properties and behaviors of a parent object.

- The idea behind inheritance in OOP is that you can create new classes that are built upon existing classes.

- Inheritance represents the **IS-A** relationship which is also known as a parent-child relationship.

Inheritance Example



Terms used in Inheritance

- **Class**: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

- **Sub Class**: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

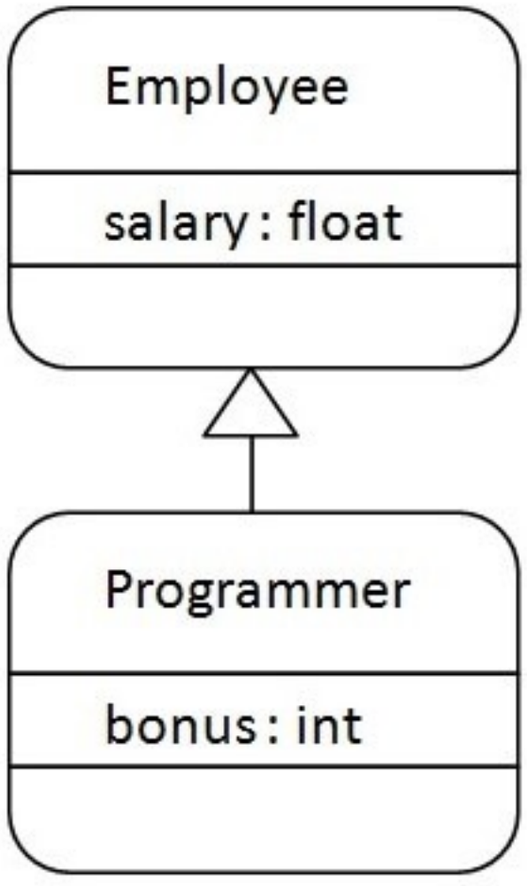
- **Super Class**: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

The syntax of Java Inheritance

```
class SubclassName extends SuperclassName {
    //methods and fields
}
```

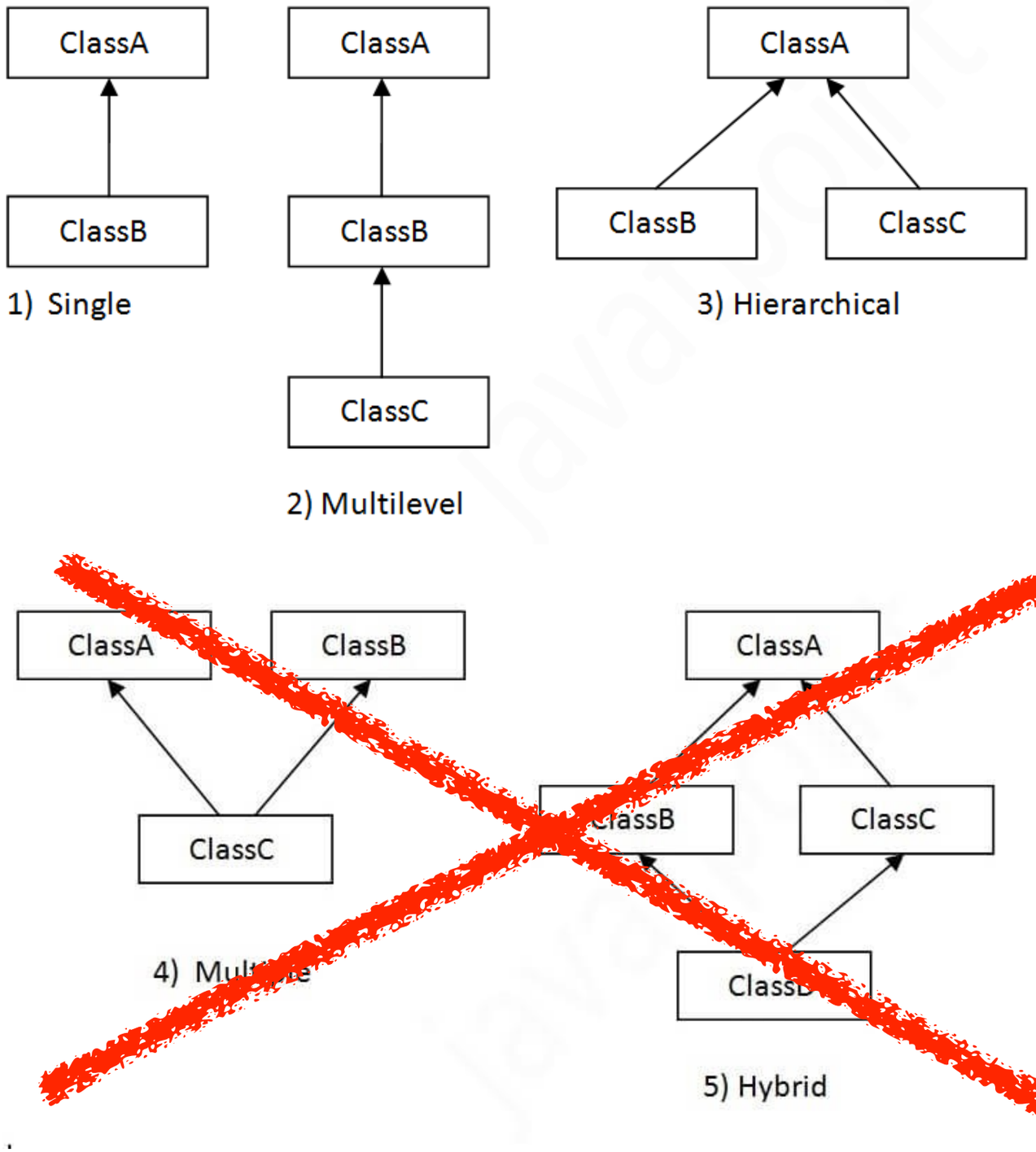
- The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

Java Inheritance Example




```
class Employee{
   float salary=40000;
}
class Programmer extends Employee{
   int bonus=10000;
      public static void main(
                  String args[]){
         Programmer p=new Programmer();
         System.out.println("Programmer
               salary is:"+p.salary);
         System.out.println("Bonus of
               Programmer is:"+p.bonus);
   }
}
```

Types of inheritance in java



- Multiple inheritance is **not supported** in Java through class
- But through the **interface**, multiple inheritance is possible.

Why use inheritance

- For Method Overriding
  - so runtime polymorphism can be achieved.
- For Code Reusability.
  - **Reusability**: As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

Aggregation

- If a class have an entity reference, it is known as Aggregation. Aggregation represents **HAS-A** relationship.

-- Consider a situation, Employee object contains many informations such as id, name, emailld etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc.

```
class Employee{
   int id;
   String name;
   Address address;
   //Address is a class
   ...
}
```

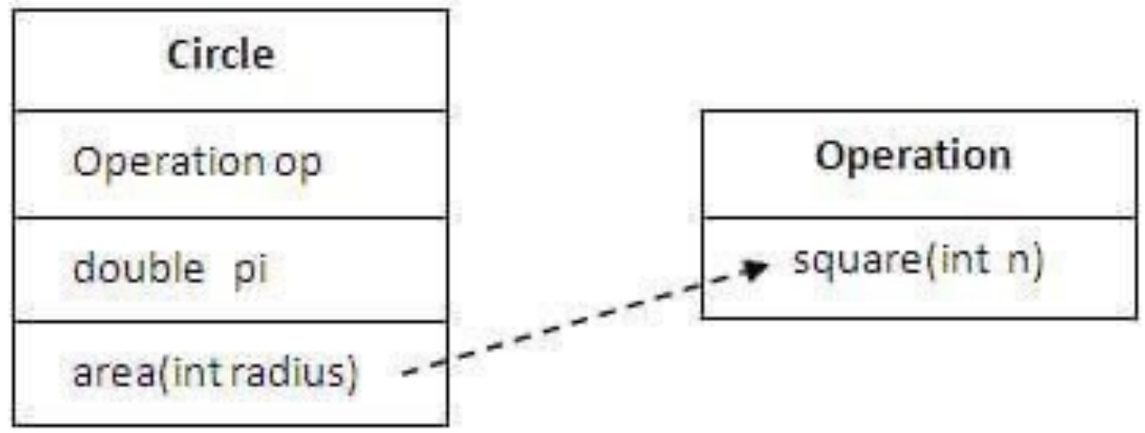
Why use Aggregation?

- **Code reuse** is also best achieved by aggregation when there is no is-a relationship.

- Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; **otherwise**, aggregation is the best choice.

Java aggregation example

```
class Operation{
    int square(int n){
        return n*n;
    }
}
class Circle{
    Operation op;//aggregation
    double pi=3.14;
    double area(int radius){
        op=new Operation();
        int rsquare=op.square(radius);
        //code reusability (delegates the method call).
        return pi*rsquare;
    }
}
```

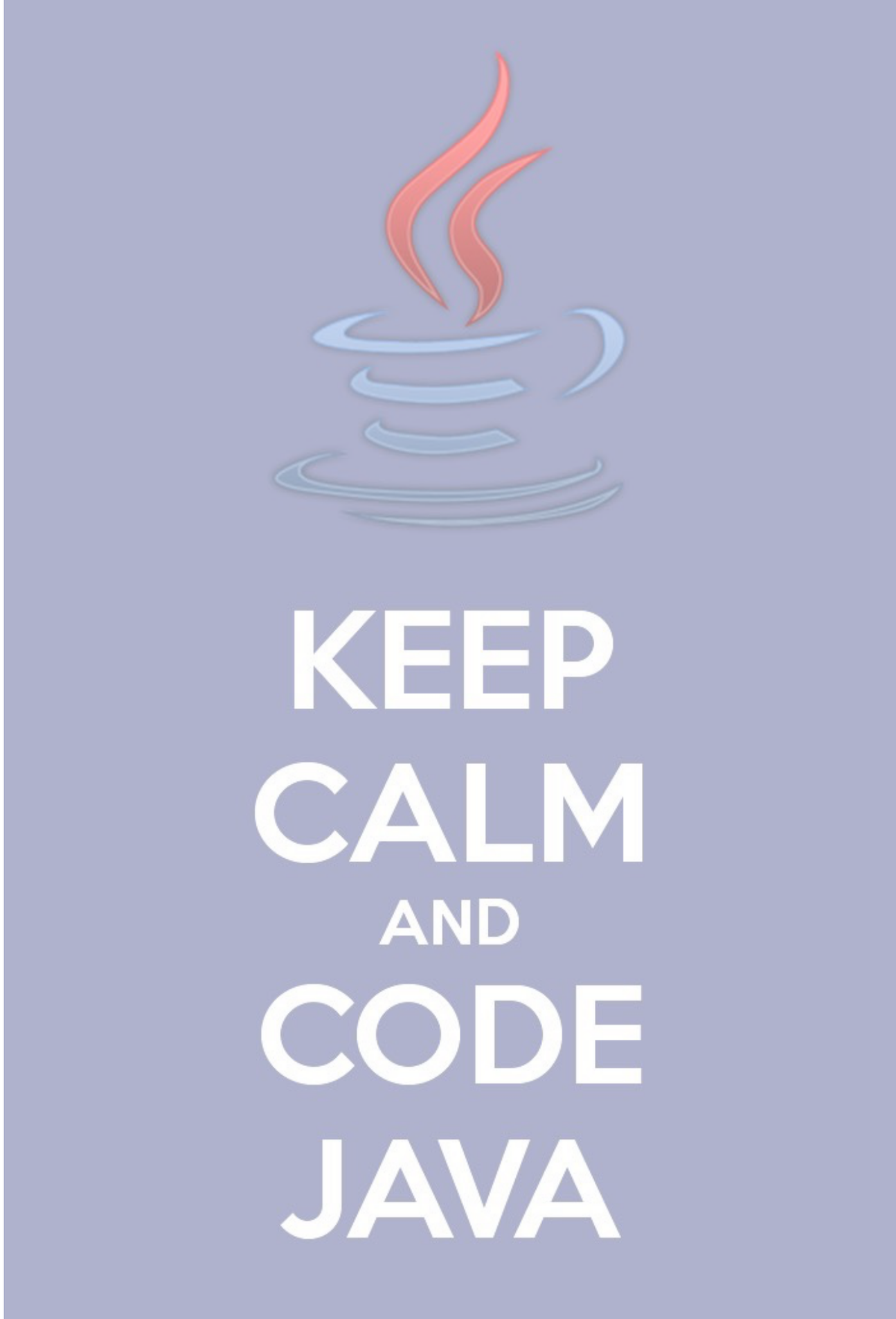


Inheritance vs Aggregation

— IS-A, or HAS-A, that is the question.

Inheritance

1. Inheritance
2. **Overriding and Hiding**
3. null, this, and super
4. Polymorphism
5. final and abstract
6. Object class



Method Overriding

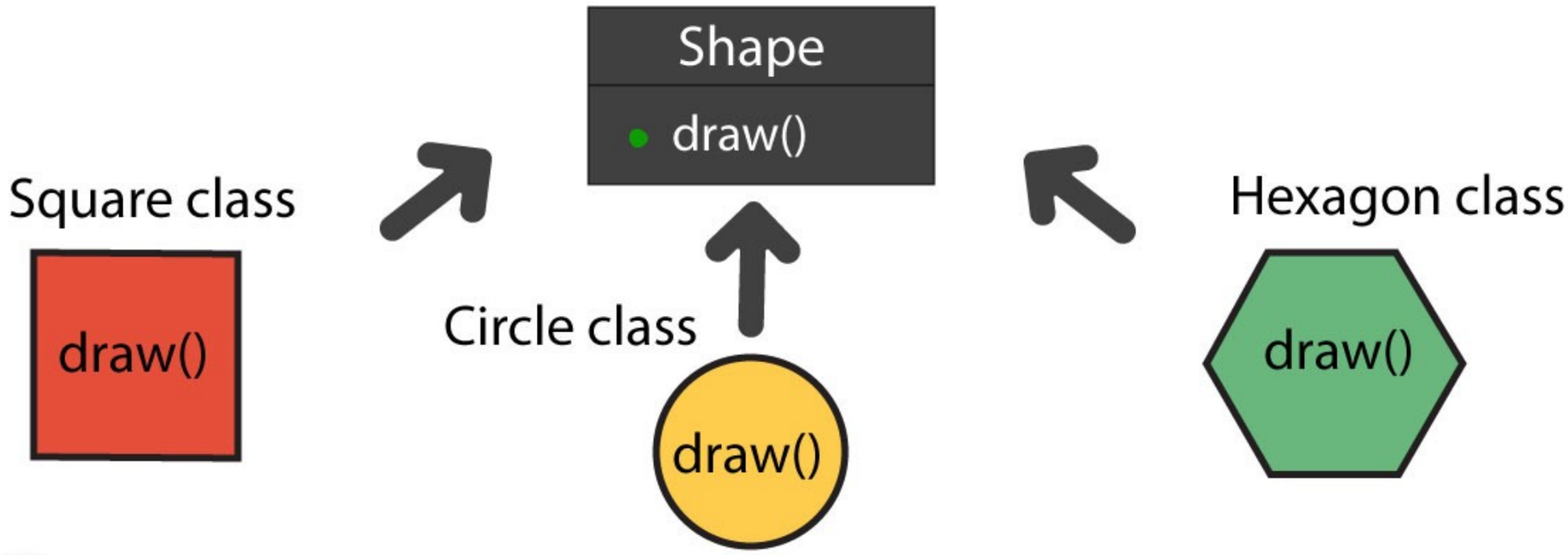
- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding** in Java.

- In other words, a subclass can provide a specific implementation of the method that has been declared by one of its parent class.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.

- Method overriding is used for **runtime polymorphism.**



Rules for Java Method Overriding

- The method must have the **same name** as in the parent class

- The method must have the **same parameter** as in the parent class.

- There must be an IS-A relationship (inheritance).

```
class Vehicle{
   void run(){
      System.out.println("Vehicle is running");
   }
}
class Bike extends Vehicle{
   public static void main(String args[]){
      //creating an instance of child class
      Bike obj = new Bike();
      //calling the method with child class instance
      obj.run();
   }
}
```

Example without method overriding

```
class Vehicle{
   //defining a method
   void run(){
      System.out.println("Vehicle is running");
   }
}
class Bike2 extends Vehicle{
   //defining the same method as in the parent class
   void run(){
      System.out.println("Bike is running safely");
   }
   public static void main(String args[]){
      Bike2 obj = new Bike2();//creating object
      obj.run();//calling method
   }
}
```

Example with method overriding

Override Static Method

- Can we override static method?—No, a **static method cannot be overridden**.

- Why can we not override static method?—It is because the static method is bound with class whereas instance method is bound with an object.

- Can we override java main method?—No, because the main is a static method.

Method Overloading	vs	Method Overriding
Method overloading is used to increase the readability of the program.		Method overriding is used to provide the specific implementation of the method that is already provided by its super class.
Method overloading is performed within class.		Method overriding occurs in two classes that have IS-A (inheritance) relationship.
In case of method overloading, parameter must be different.		In case of method overriding, parameter must be same.
Method overloading is the example of **compile time polymorphism**.		Method overriding is the example of **run time polymorphism.**
In java, method overloading can't be performed by changing return type of the method only. Return type can be same or different in method overloading.		Return type must be same or covariant in method overriding.

Method Hiding

- A class inherits all non-private static methods from its superclass.

- **Redefining an "inherited" static method** in a class is known as method hiding.

- The redefined static method in a subclass hides the static method of its superclass.

```
class MySuper {
   public static void print() {
      System.out.println("Inside MySuper.print()");
   }
}
class MySubclass extends MySuper {
   public static void print() {
      System.out.println("Inside MySubclass.print()");
   }
   public static void main(String[] args) {
      MySuper mhSuper = new MySubclass();
      MySubclass mhSub = new MySubclass();
      MySuper.print();
      MySubclass.print();
      ((MySuper) mhSub).print();
      mhSuper = mhSub;
      mhSuper.print();
      ((MySubclass) mhSuper).print();
   }
}
```

Example of method hiding

Field Hiding

- Variable hiding happens when we declare a property in a local **scope** that has the same name as the one we already have in the outer scope.

- Similarly, when both the child and the parent classes have a variable with the same name, the child's variable hides the one from the parent.

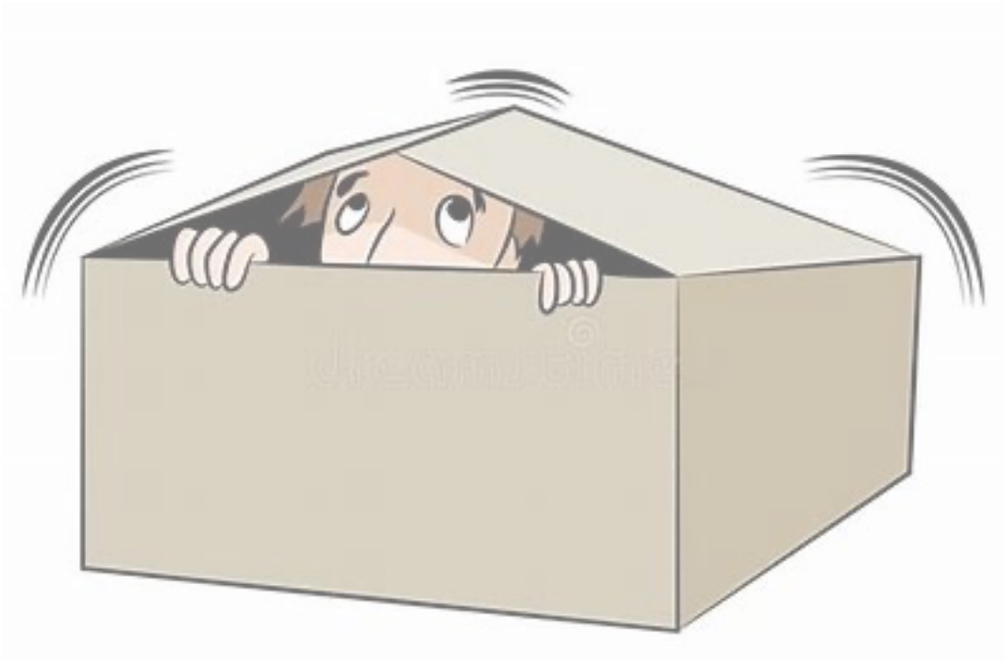
```
public class ParentVariable {
   String instanceVariable = "parent variable";
   public void printInstanceVariable() {
      System.out.println(instanceVariable);
   }
}
public class ChildVariable extends ParentVariable {
   String instanceVariable = "child variable";
   public void printInstanceVariable() {
      System.out.println(instanceVariable);
   }
   public static void main(String[] args) {
    ParentVariable childVariable = new ChildVariable();
    childVariable.printInstanceVariable();
   }
}
```

Example of field hiding

Why use field hiding

- In most cases, we should **avoid** creating variables with the same name both in parent and child classes.

- Instead, we should use a proper access modifier like private and provide **getter/setter** methods for that purpose.



Inheritance

1. Inheritance
2. Overriding and Hiding
3. **null, this, and super**
4. Polymorphism
5. final and abstract
6. Object class

null

- In Java, null is a reserved word for literal values. It seems like a keyword, but actually, it is a literal similar to true and false.

- Java **null** is Case Sensitive, so we do not write it as NULL or 0 as we would in C.

- Any reference variable automatically has a null value.

- A non-static value cannot be called by a null value, it will throw a **NullPointerException**, but this won't happen with a static method.

this

- this can be used to refer current class instance variable.
- this can be used to invoke current class method
- this() can be used to invoke current class constructor.
- this can be passed as an argument.
- this can be used to return the current class instance from the method.

this is a reference to the current instance of a class

```
public class ThisExample {
    private int x;
    public void setX(int x) {
        //Sometimes it is easier to keep
        //the variable names the same,
        //as they represent the same
        //thing.
        this.x = x;
    }
}
```

this to invoke current class method

```
class ThisExample {
    private int x, y;
    void setValue(int x) {
        this.x = x;
    }
    void setValue(int x, int y) {
        this.y = y;
        this.setValue(x);
    }
}
```

this() to invoke current class constructor

```
class ThisExample {
    private int x, y;
    ThisExample(int x) {
        this.x = x;
    }
    ThisExample(int x, int y) {
        this(x);
        this.y = y;
    }
}
```

this to pass as an argument

```
class ThisExample {
    void print(ThisExample o) {
        System.err.println(
                "Print something...");
    }
    void print() {
        print(this);
    }
}
```

this can be used to return current class instance

```
class ThisExample {
    ThisExample getInstance() {
        return this;
    }
}
```

super

- super can be used to refer parent class instance variable.
- super can be used to invoke parent class method.
- super() can be used to invoke parent class constructor.

super to refer parent class instance variable

```
class Animal {
    String name = "animal";
}

class Dog extends Animal {
    String name = "dog";
    void print() {
        System.out.println(name
                + " is a " + super.name);
    }
    public static void main(String args[]) {
        Dog d = new Dog();
        d.print();
    }
}
```

super to invoke parent class method

```
class Animal {
    private String name = "animal";
    public String getName() {
        return name;
    }
}
class Dog extends Animal {
    String name = "dog";
    public String getName() {
        return name + " is a " + super.getName();
    }
    void print() {
        System.out.println(getName());
    }
    public static void main(String args[]) {
        Dog d = new Dog();
        d.print();
    }
}
```

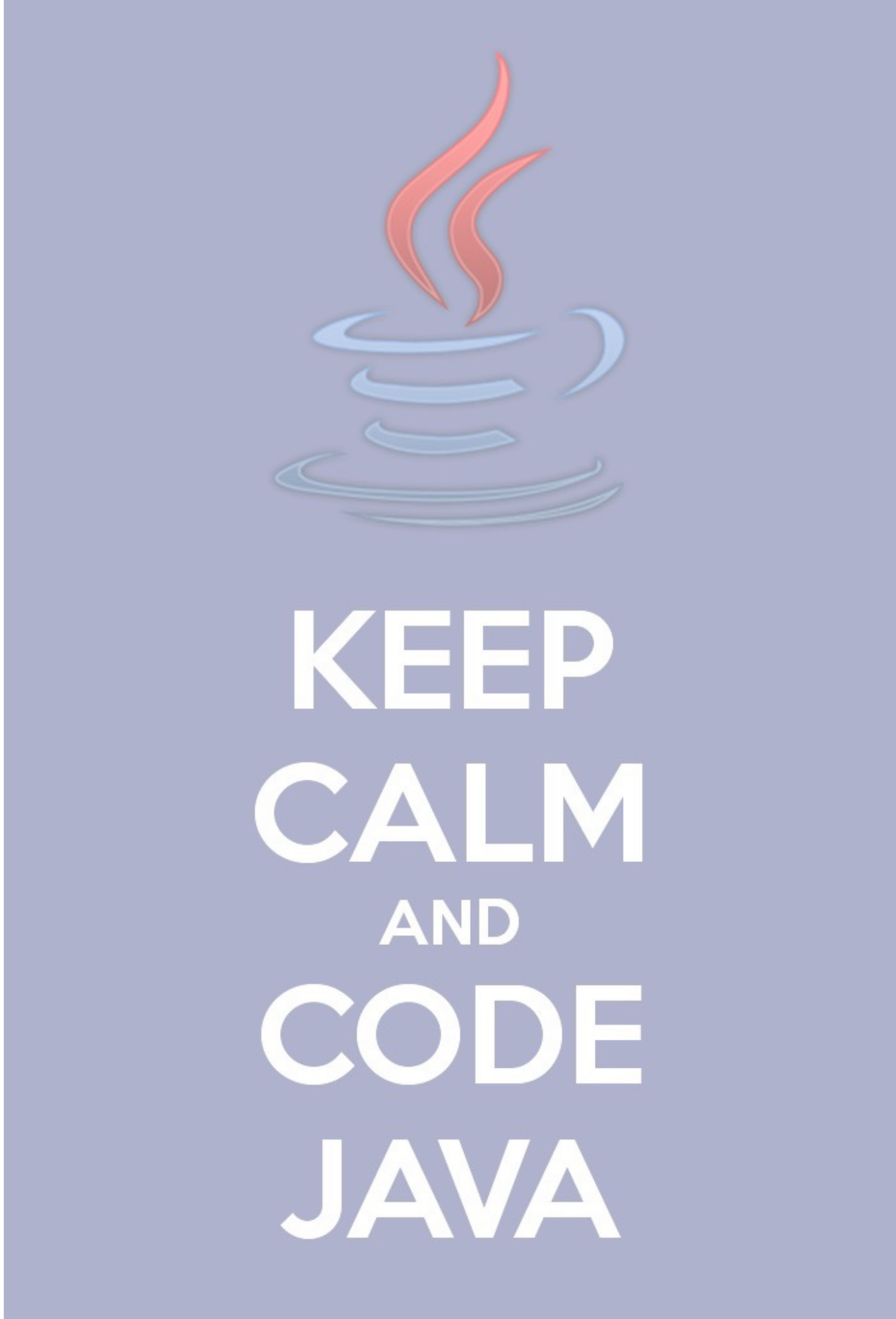
super to invoke parent class constructor

```
class Animal {
    Animal() {
        System.out.println("animal is created");
    }
}

class Dog extends Animal {
    Dog() {
        super(); // call by default
        System.out.println("dog is created");
    }
    public static void main(String args[]) {
        Dog d = new Dog();
    }
}
```

Inheritance

1. Inheritance
2. Overriding and Hiding
3. null, this, and super
4. **Polymorphism**
5. final and abstract
6. Object class



Polymorphism

- Polymorphism in OOP is a concept by which we can **perform a single action in different ways**.
  - Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

- We can perform polymorphism by method overloading and method overriding.

- There are two types of polymorphism in Java: **compile-time polymorphism** and **runtime polymorphism**.

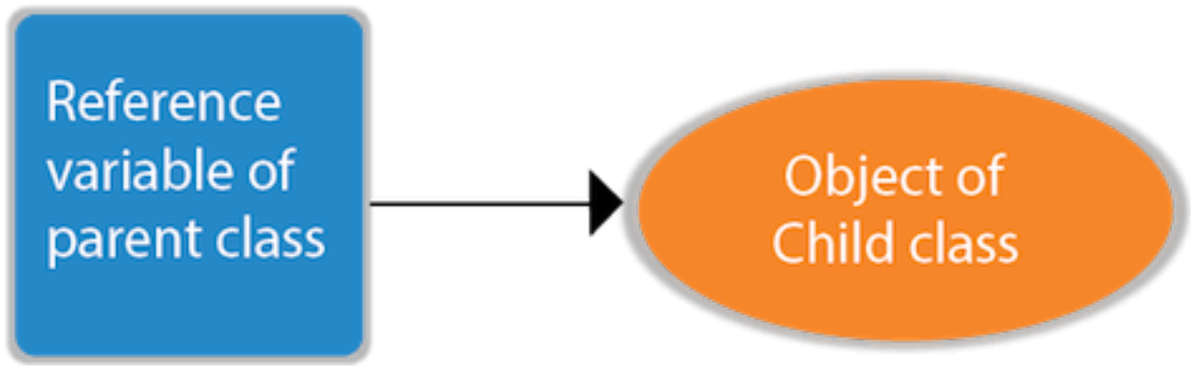
Runtime Polymorphism

- Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

- In this process, an overridden method is called through the **reference variable of a superclass**. The determination of the method to be called is based on the **object being referred** to by the reference variable.

Upcasting

- If the reference variable of Parent class refers to the object of Child class, it is known as upcasting.



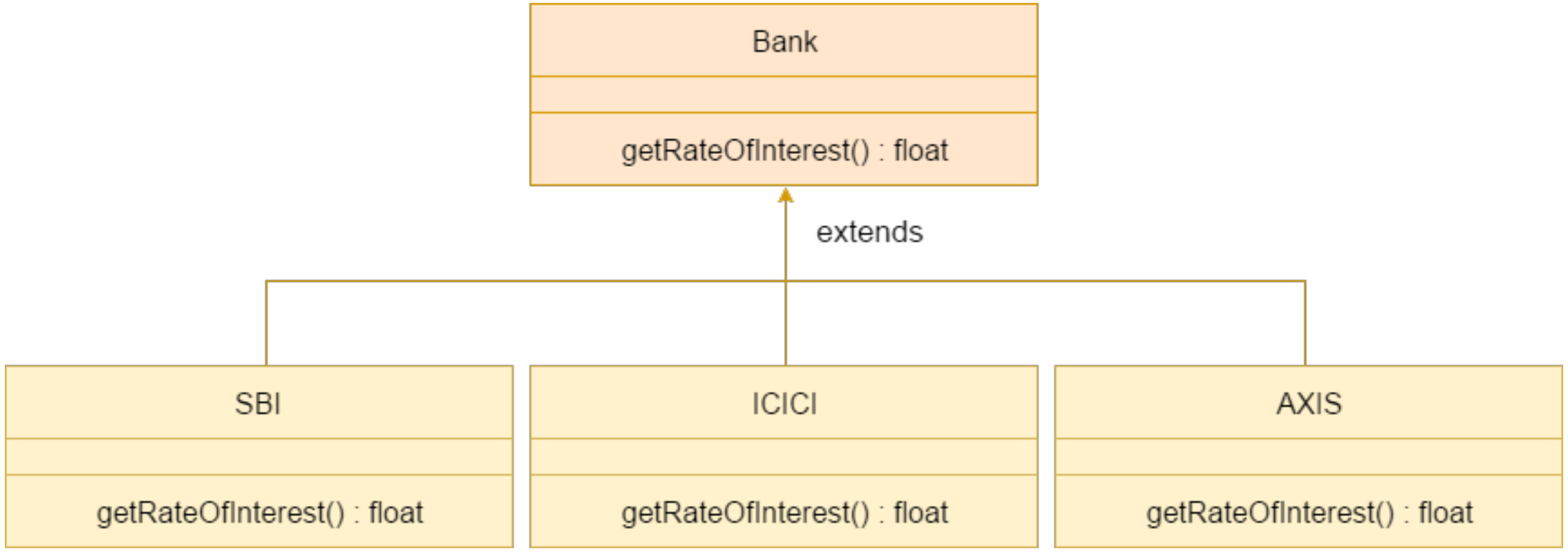

```
class A{}
class B extends A{}
A a=new B();  //upcasting
```

Runtime Polymorphism Example 1

```
class Bike{
    void run(){
        System.out.println("running");
    }
}
class Splendor extends Bike{
    void run(){
        System.out.println(
                "running safely with 60km");
    }
    public static void main(String args[]){
        Bike b = new Splendor();//upcasting
        b.run();
    }
}
```

Runtime Polymorphism Example 2

- Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.



Runtime Polymorphism Example 2

```
class Bank {
    float getRateOfInterest() {
        return 0;
    }
}
```

```
class ICICI extends Bank {
    float getRateOfInterest() {
        return 7.3f;
    }
}
class AXIS extends Bank {
    float getRateOfInterest() {
        return 9.7f;
    }
}
class SBI extends Bank {
    float getRateOfInterest() {
        return 8.4f;
    }
}
```

Runtime Polymorphism Example 2

```
class TestPolymorphism {
    public static void main(String args[]) {
        Bank b;

        b = new SBI();
        System.out.println("SBI Rate of Interest: "
                + b.getRateOfInterest());

        b = new ICICI();
        System.out.println("ICICI Rate of Interest: "
                + b.getRateOfInterest());

        b = new AXIS();
        System.out.println("AXIS Rate of Interest: "
                + b.getRateOfInterest());
    }
}
```

Runtime Polymorphism Example 3

- A method is overridden, not the data members, so runtime polymorphism can only be achieved by method, not data members.

```
class Bike {
    int speedlimit = 90;
    public int getSpeedlimit() {
        return speedlimit;
    }
}
public class Honda extends Bike {
    int speedlimit = 150;
    public int getSpeedlimit() {
        return speedlimit;
    }
    public static void main(String args[]) {
        Bike obj = new Honda();
        System.out.println(
                obj.speedlimit);//90
        System.out.println(
                obj.getSpeedlimit());//150
    }
}
```

Runtime Polymorphism with Multilevel Inheritance

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

class Dog extends Animal {
    void eat() {
        System.out.println(
            "eating fruits");
    }
}
```

```
class BabyDog extends Dog {
    void eat() {
        System.out.println(
                "drinking milk");
    }
    public static void main(
                String args[]) {
        Animal a1, a2, a3;
        a1 = new Animal();
        a2 = new Dog();
        a3 = new BabyDog();
        a1.eat();
        a2.eat();
        a3.eat();
    }
}
```

Compile time polymorphism

- Polymorphism that is resolved during compile time is known as static polymorphism.
    - Method overloading is an example of compile time polymorphism.
    - Hide a static method is also an example of compile time polymorphism.

Compile time Polymorphism vs Run time Polymorphism	
In Compile time Polymorphism, call is resolved by the **compiler**.	In Run time Polymorphism, call is **not** resolved by the compiler.
It is also known as Static binding, Early binding and overloading as well.	It is also known as Dynamic binding, Late binding and overriding as well.
Overloading is compile time polymorphism where more than one methods share the same name with different parameters or signature and different return type.	**Overriding** is run time polymorphism having same method with same parameters or signature, but associated in a class & its subclass.
It provides **fast execution** because known early at compile time.	It provides **slow execution** as compare to early binding because it is known at runtime.
Compile time polymorphism is **less flexible** as all things execute at compile time.	Run time polymorphism is **more flexible** as all things execute at run time.

Inheritance

1. Inheritance
2. Overriding and Hiding
3. null, this, and super
4. Polymorphism
5. **final and abstract**
6. Object class

final

- The final keyword does not allow modifying or replacing its original value or definition.

- The final keyword can be used in the following three contexts:
  - A variable declaration
  - A class declaration
  - A method declaration

final variable

- If a variable is declared final, it can be assigned a value only once.

```
final int YES  = 1;

public void  test(final int x)  {
    ...
}

public static final String MSG;
static {
   MSG = "final static variable";
}
```

final class

- If a class is declared final, it cannot be extended (or subclassed).

```
final class A {
    // methods and fields
}

// The following class is illegal.
class B extends A {
    // COMPILE-ERROR! Can't subclass A
}
```

final method

- If a method is declared final, it cannot be overridden in the subclasses of the class that contains the method.

```
class A {
    final void m() {
        System.out.println("A final method.");
    }
}

class B extends A {
    void m() {
        // COMPILE-ERROR! Can't override.
        System.out.println("Illegal!");
    }
}
```

Abstraction in Java

- Abstraction is a process of hiding the implementation details and showing only functionality to the user.

- Abstraction shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

- Abstraction lets you focus on what the object does instead of how it does it.

abstract class

- A class which is declared with the abstract keyword is known as an abstract class in Java.

- Example of abstract class

```
public abstract class Shape  {
     ……
}
```

Rules for abstract class

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- **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.

abstract method

- A method which is declared as abstract and does not have implementation is known as an abstract method.

- Example of abstract method

```
abstract void printShape();
//no method body
```

Rules for abstract method

- Abstract methods don’t have body.

- If a class has an abstract method it should be declared abstract, the **vice versa is not true**, which means an abstract class doesn’t need to have an abstract method compulsory.

- If a regular class extends an abstract class, then the class must have to implement all the abstract methods of abstract parent class or it has to be declared abstract as well.

Example of abstract class with abstract method

```
abstract class Bike {
    abstract void run();
}
class Honda extends Bike {
    void run() {
        System.out.println("running safely");
    }
    public static void main(String args[]) {
        Bike obj = new Honda();
        obj.run();
    }
}
```

Another example

```
abstract class Shape {
    abstract void draw();
}

class Rectangle extends Shape {
    void draw() {
        System.out.println(
            "drawing rectangle");
    }
}
```

```
class Circle extends Shape {
    void draw() {
        System.out.println(
            "drawing circle");
    }
    public static void main(
                String args[]) {
        Shape s = new Circle();
        s.draw();
    }
}
```

3rd example

```
abstract class Bank {
    abstract int getRateOfInterest();
}

class SBI extends Bank {
    int getRateOfInterest() {
        return 7;
    }
}

class PNB extends Bank {
    int getRateOfInterest() {
        return 8;
    }
}
```

```
class TestBank {
    public static void main(
                    String args[]) {
        Bank b;
        b = new SBI();
        System.out.println(
                "Rate of Interest is: "
                + b.getRateOfInterest()
                + " %");
        b = new PNB();
        System.out.println(
                "Rate of Interest is: "
                + b.getRateOfInterest()
                + " %");
    }
}
```

4th example

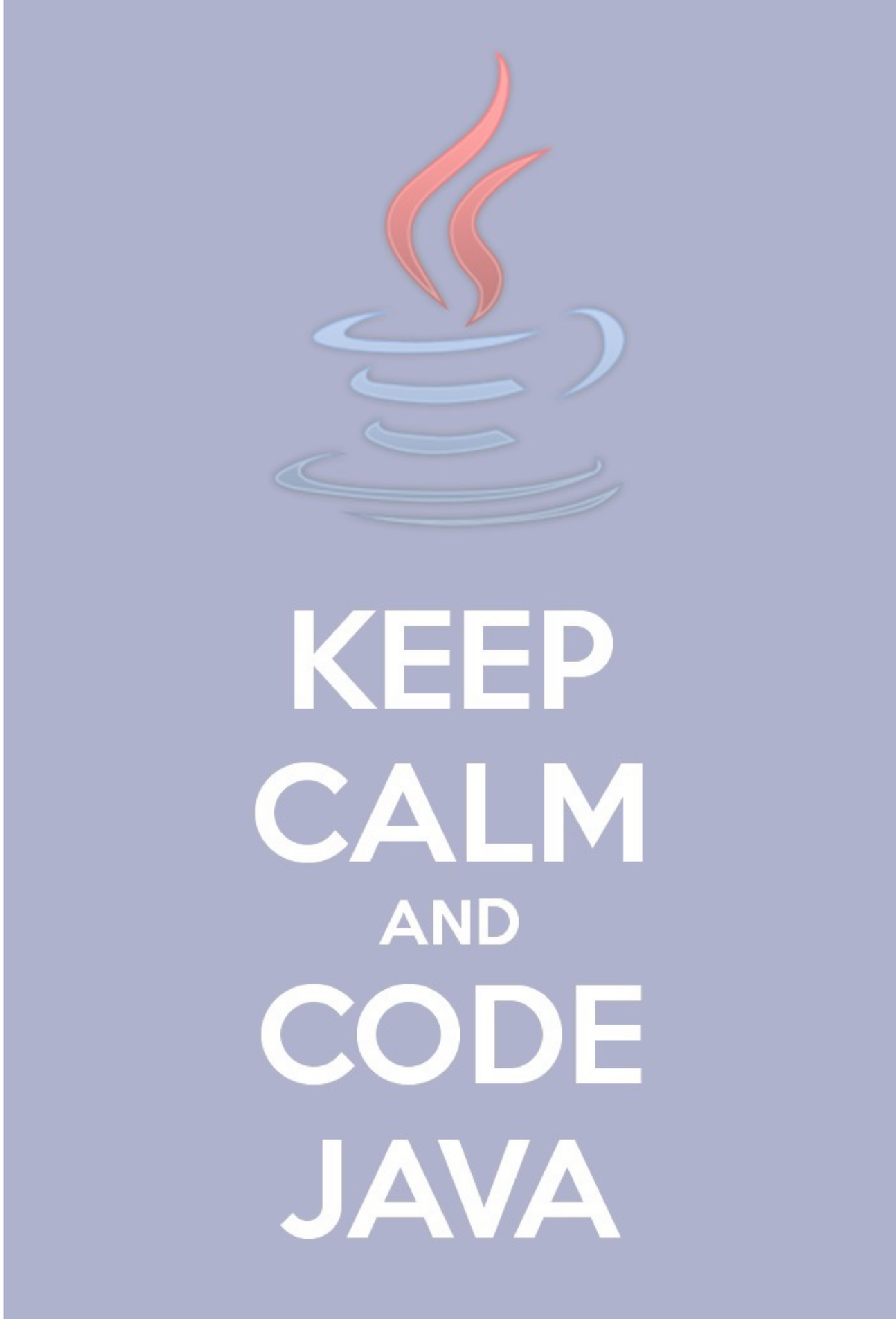
```
abstract class Bike {
    Bike() {
        System.out.println(
            "bike is created");
    }
    abstract void run();
    void changeGear() {
        System.out.println(
            "gear changed");
    }
}
```

```
class Honda extends Bike {
    void run() {
        System.out.println(
            "running safely..");
    }
}

public class TestAbstraction {
    public static void main(
            String args[]) {
        Bike obj = new Honda();
        obj.run();
        obj.changeGear();
    }
}
```

Inheritance

1. Inheritance
2. Overriding and Hiding
3. null, this, and super
4. Polymorphism
5. final and abstract
6. **Object class**



Object class in Java

- The Object class is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

- The Object class is beneficial if you want to refer any object whose type you don't know.

- The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.

Methods of Object class

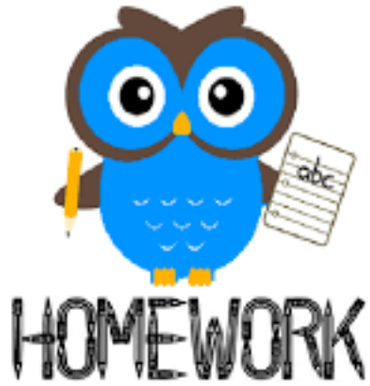
Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.

Methods of Object class

Method	Description
public String toString()	returns the string representation of this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

Methods of Object class

Method	Description
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).



Person

-name:String
-address:String

+Person(name:String,address:String)
+getName():String
+getAddress():String
+setAddress(address:String):void
+toString():String

"Person[name=?,address=?]"

extends

Student

-program:String
-year:int
-fee:double

+Student(name:String,address:String,
  program:String,year:int,fee:double)
+getProgram():String
+setProgram(program:String):void
+getYear():int
+setYear(year:int):void
+getFee():double
+setFee(fee:double):void
+toString():String

"Student[Person[name=?,address=?],
program=?,year=?,fee=?]"

Staff

-school:String
-pay:double

+Staff(name:String,address:String,
  school:String,pay:double)
+getSchool():String
+setSchool(school:String):void
+getPay():double
+setPay(pay:double):void
+toString():String

"Staff[Person[name=?,address=?],
school=?,pay=?]"


abc
HOMEWORK


<<abstract>> Shape
#color:String
#filled:boolean
+Shape()
+Shape(color:String,filled:boolean)
+getColor():String
+setColor(color:String):void
+isFilled():boolean
+setFilled(filled:boolean):void
+getArea():double
+getPerimeter:double
+toString():String
Circle
#radius:double
+Circle()
+Circle(radius:double)
+Circle(radius:double,
color:String,filled:boolean)
+getRadius():double
+setRadius(radius:double):void
+getArea():double
+getPerimeter():double
+toString():String
Rectangle
#width:double
#length:double
+Rectangle()
+Rectangle(width:double,length:double)
+Rectangle(width:double,length:double,
color:String,filled:boolean)
+getWidth():double
+setWidth(width:double):void
+getLength():double
+setLength(legnth:double):void
+getArea():double
+getPerimeter():double
+toString():String
Square
+Square()
+Square(side:double)
+Square(side:double,color:String,
filled:boolean)
+getSide():double
+setSide(side:double):void
+setWidth(side:double):void
+setLength(side:double):void
+toString():String