MOD -02 -> OOPs

Inheritance

Inheritance

Inheritance is a mechanism in which one class acquires the property of another class.

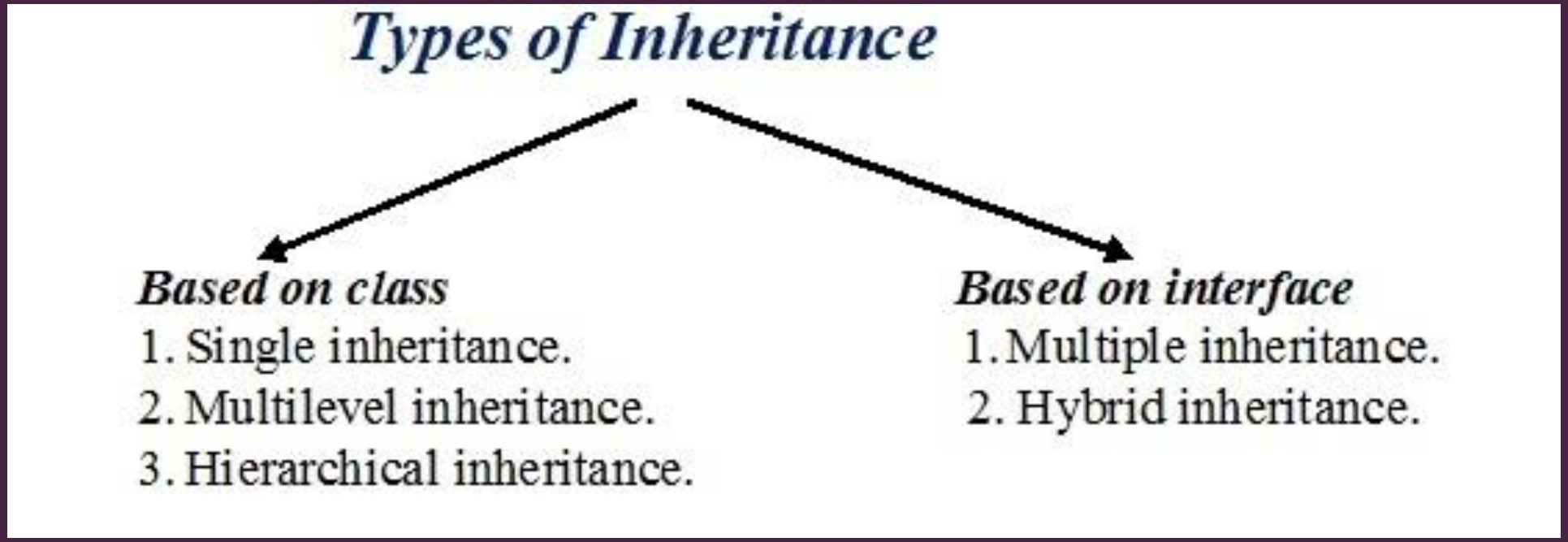
For example, a child inherits the traits of his/her parents.

1. With inheritance, we can reuse the fields and methods of the existing class.
2. Hence, inheritance facilitates Reusability and is an important concept of OOPs.
3. The keyword "extends" is used by the sub class to inherit the features of super class
4. Inheritance is a way to implement IS-A relationship i.e. parent child relationship.

Why inheritance is used?

1. Code re-usability.
2. Run-time polymorphism.

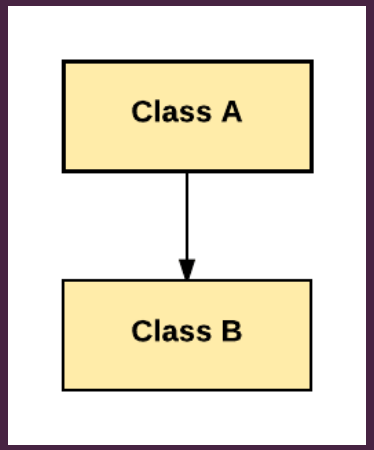
Types of inheritance

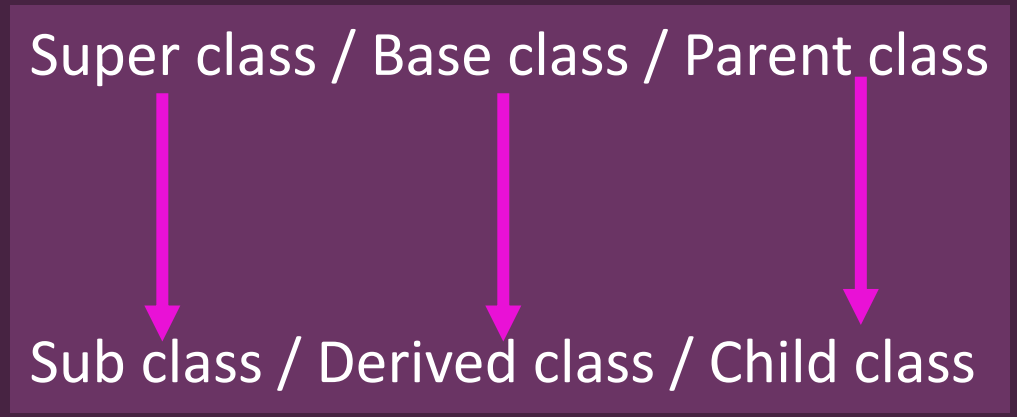


1. Single Inheritance

In Single Inheritance one class extends another class (one class only).

When a derived class inherits the properties and behavior from a single parent class. It is known as single inheritance.





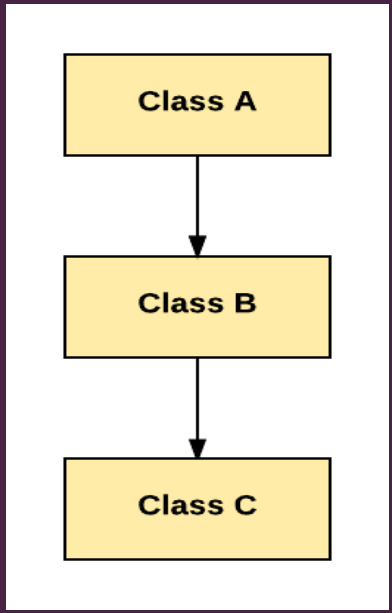
In above diagram, Class B extends only Class A.

Class A is a super class and Class B is a Sub-class.

2. Multilevel Inheritance

In Multilevel Inheritance, one class can inherit from a derived class.

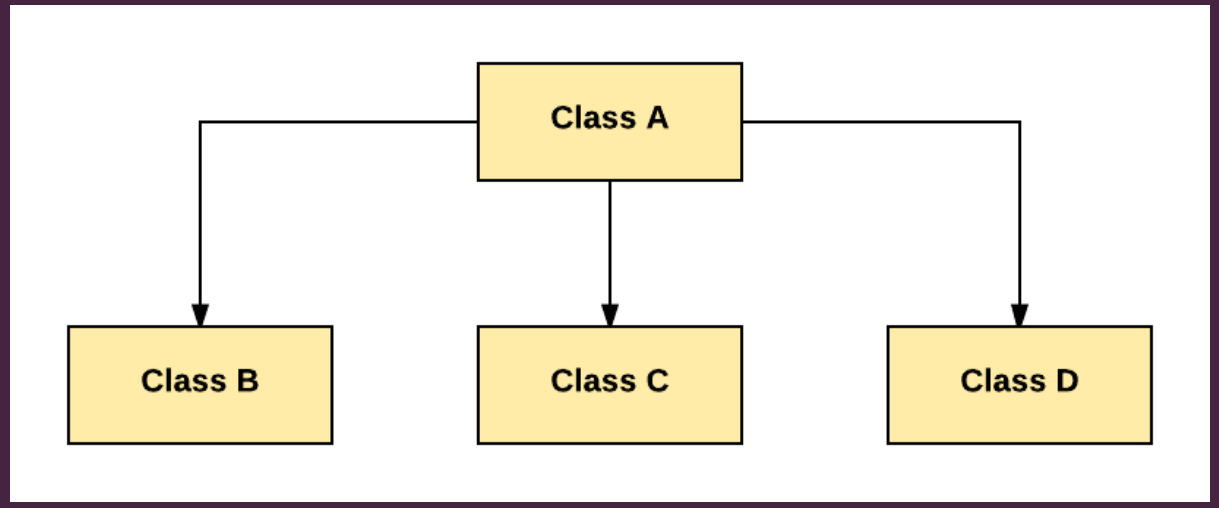
Hence, the derived class becomes the base class for the new class. Ie. When a derived class inherits the properties and behavior from a derived class. It is known as multilevel inheritance.



As per shown in diagram Class C is subclass of B and B is a of subclass Class A.

3. Hierarchical Inheritance

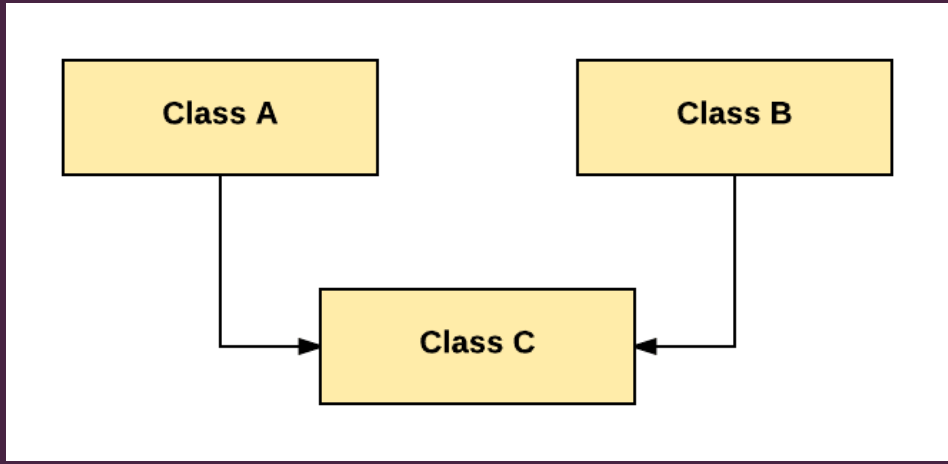
In Hierarchical Inheritance, one class is inherited by many sub classes.



As per above example, Class B, C, and D inherit the same class A.

4. Multiple Inheritance

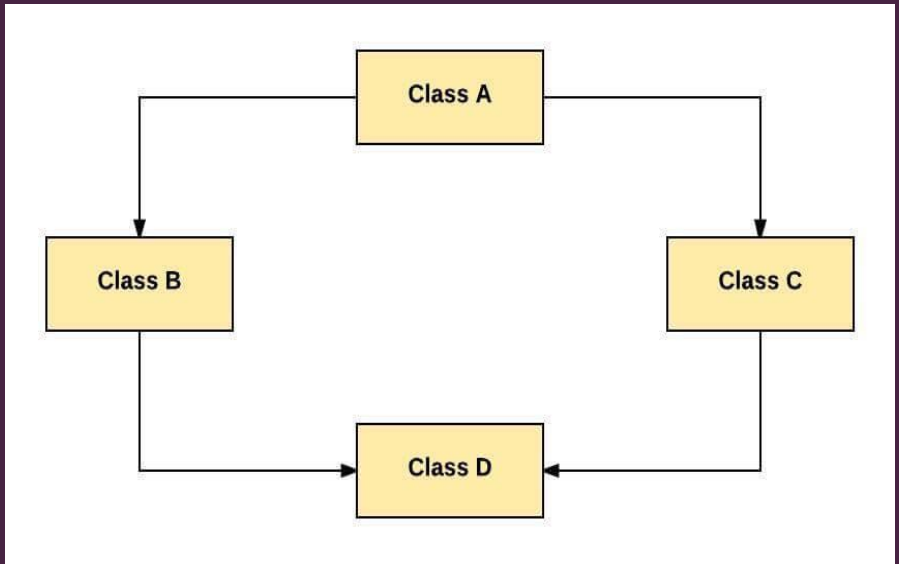
In Multiple Inheritance, one class extending more than one class. Java does not support multiple inheritances.



As per above diagram, Class C extends Class A and Class B both.

5. Hybrid Inheritance

Hybrid inheritance is a combination of Single and Multiple inheritance.



As per above example, all the public and protected members of Class A are inherited into Class D, first via Class B and secondly via Class C.

1. Single inheritance

```java
package inheritancepack;


//Parent class (Base class)
class Teacher {
    public String name;
    public int age;


    Teacher(String name, int age) {
        this.name = name;
        this.age = age;
    }


    void teach() {
        System.out.println(name + " is teaching.");
    }

}
```

```java
//Child class (Derived class)
class MathTeacher extends Teacher {
    String subject;


    MathTeacher(String name, int age, String subject) {


    super(name, age); // Calling parent constructor
    /*
    * If you try this.name = name; in MathTeacher: It gives an error only if
    * there’s no name field in MathTeacher, because Java checks the current class
    * for that field. Since name is in the superclass, you’d be accessing an
    * inherited variable without declaring it locally, which may confuse Java about
    * where name belongs if not handled carefully. So, those fields should be
    * initialized using the Teacher's constructor. super(name, age); calls the
    * parent constructor, which already does:this.name = name; and this.age = age;
    */


    this.subject = subject;
    }
```

```
    void explainMathConcept() {
        System.out.println(name + " is explaining a " + subject + " concept.");
    }
}


public class SingleInheritance {


    public static void main(String[] args) {
        MathTeacher mTeacher = new MathTeacher("Arun", 40, "Algebra");
        mTeacher.teach(); // From parent class
        mTeacher.explainMathConcept(); // From child class
    }
}
```

```java
class Teacher {
    public String name;
    public int age;

    /* commented constructor in super/base/parent class
    Teacher(String name, int age) {
        this.name = name;
        this.age = age;
    }
    */
```

```java
MathTeacher(String name, int age, String subject) {


//super(name, age); // no need this
//instead you can use

this.name=name;
this.age=age;

Or
super.name=name;
super.age=age;
```

Using super(name, age) is the clean and recommended way, because it ensures:

- Constructor logic stays in the parent class, and
- You don’t duplicate initialization logic in subclasses.

2. Multi Level inheritance

```
package inheritancepack;
class Vehicle {//Level 1 - Base Class
    String brand;

    Vehicle(String brand) {
        this.brand = brand;
    }
    void start() {
            System.out.println(brand + " vehicle is starting.");
    }
}
class Car extends Vehicle {//Level 2 - Intermediate Class
    String fuelType;

    Car(String brand, String fuelType) {
        super(brand); // Call Vehicle constructor
        this.fuelType = fuelType;
    }
    void drive() {
    System.out.println(brand + " car runs on " + fuelType + ".");
    }
}
```

```java
//Level 3 - Derived Class
class ElectricCar extends Car {
    int batteryCapacity;

    ElectricCar(String brand, int batteryCapacity) {
        super(brand, "Electric"); // Call Car constructor
        this.batteryCapacity = batteryCapacity;
    }
    void charge() {
        System.out.println(brand + " electric car charging with " + batteryCapacity + " kWh
        battery.");
    }
}

public class MultilevelInheritance {

    public static void main(String[] args) {
        ElectricCar tesla = new ElectricCar("Tesla", 75);
        tesla.start(); // from Vehicle
        tesla.drive(); // from Car
        tesla.charge(); // from ElectricCar
    }
}
```

3. Hierarchical inheritance

```
package inheritancepack;
//Base class
class Account {
    String accountNumber;
    double balance;


    Account(String accountNumber, double balance) {
        this.accountNumber = accountNumber;
        this.balance = balance;
    }


    void deposit(double amount) {
        balance += amount;
        System.out.println("Deposited " + amount + ". Balance: " + balance);
    }
}
```

```java
//Derived class 1
class SavingsAccount extends Account {
    double interestRate = 0.05;


    SavingsAccount(String accountNumber, double balance) {
        super(accountNumber, balance);
    }


    void addInterest() {
        double interest = balance * interestRate;
        balance += interest;
        System.out.println("Interest added: " + interest + ". New Balance: " + balance);
    }
}
```

```java
//Derived class 2
class CurrentAccount extends Account {
    double overdraftLimit = 5000;


    CurrentAccount(String accountNumber, double balance) {
        super(accountNumber, balance);
    }


    void useOverdraft(double amount) {
        if (amount <= balance + overdraftLimit) {
            balance -= amount;
            System.out.println("Withdrew using overdraft. New Balance: " + balance);
        } else {
            System.out.println("Overdraft limit exceeded.");
        }
    }
}
```

```java
//Derived class 3
class LoanAccount extends Account {
    double emi;


    LoanAccount(String accountNumber, double balance, double emi) {
        super(accountNumber, balance);
        this.emi = emi;
    }


    void payEMI() {
        balance -= emi;
        System.out.println("EMI of " + emi + " paid. Remaining loan: " + balance);
    }
}
```

```java
public class HierarchicalInheritance {


    public static void main(String[] args) {
        SavingsAccount sa = new SavingsAccount("SA123", 10000);
        sa.deposit(2000);
        sa.addInterest();


        CurrentAccount ca = new CurrentAccount("CA123", 5000);
        ca.useOverdraft(8000);


        LoanAccount la = new LoanAccount("LA123", 100000, 5000);
        la.payEMI();
    }


}
```

Why multiple inheritances are not supported in java?

Diamond Problem

Multiple inheritances are not supported by Java because of ambiguity problem.

Suppose there are 2 classes with same method signature with different/same implementation. One class inherits these classes. A per the basic rule of inheritance, a copy of both methods should be created in the inherited class (sub class). Problem occurs now in method call, when the method is called with Inherited(sub) class object which method will be called? if you call the method by using the object of inherited class (sub class), the compiler faces an ambiguous situation not knowing which method to call. This is called *diamond problem*. That is why multiple inheritance is not supported in java.

`InheritanceDiamondProblem.java`

```java
package inheritancepack;
class EnglishTeacher{
    public void teach() {
        System.out.println("Teach English");
    }
}

class ScienceTeacher{
    public void teach() {
        System.out.println("Teach Science");
    }
}
/*
* class MultiSubjectTeacher extends EnglishTeacher,ScienceTeacher{
*
* }
*/
public class DiamondProblem {
    public static void main(String[] args) {

    }
}
```

Learn Inheritance in OOP’s with a real life scenario

ie. Banking application

We are supposed to open two different account types, one for SAVINGS A/C and another for CURRENT A/C

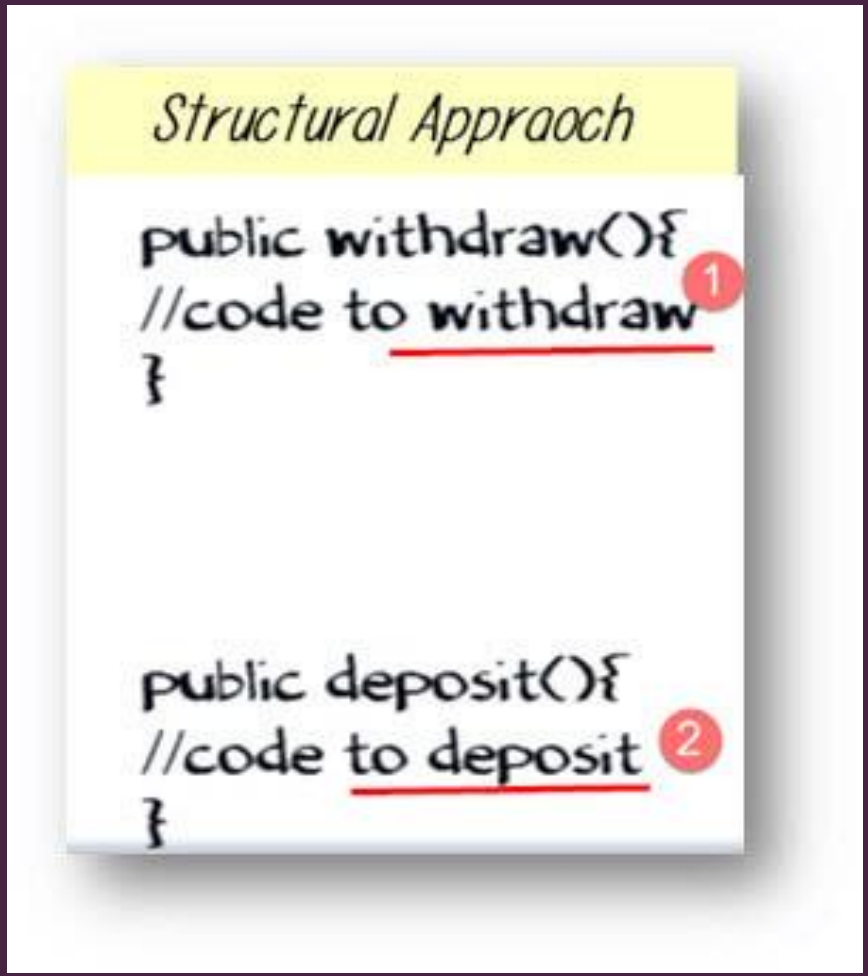
Let's compare and study how we can approach coding from a *structured* and *object-oriented programming* perspective.

Structural approach

In structured programming, we will create two functions –

1. One to withdraw
2. And the other for deposit action.

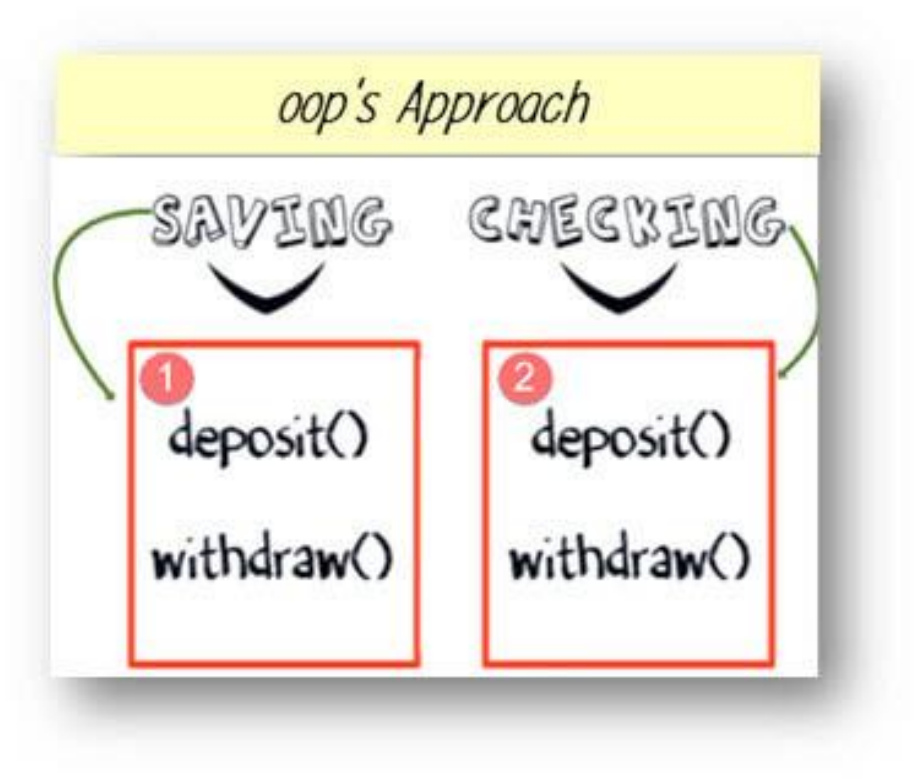
Since the working of these functions *remains same across the accounts*.


Structural Approach
public withdraw(){
//code to withdraw
}
1
public deposit(){
//code to deposit
}
2

OOP's approach

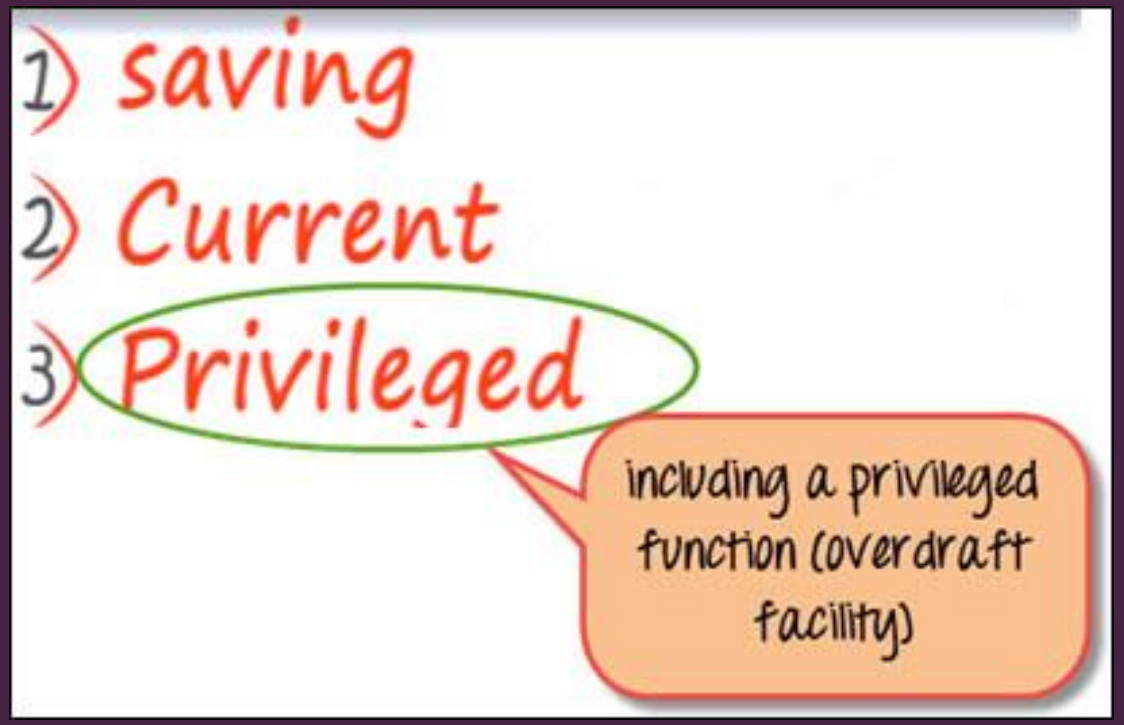
While using the OOPs programming approach. We would create two classes.

- Each having implementation of the deposit and withdraw functions.
- This will redundant extra work.



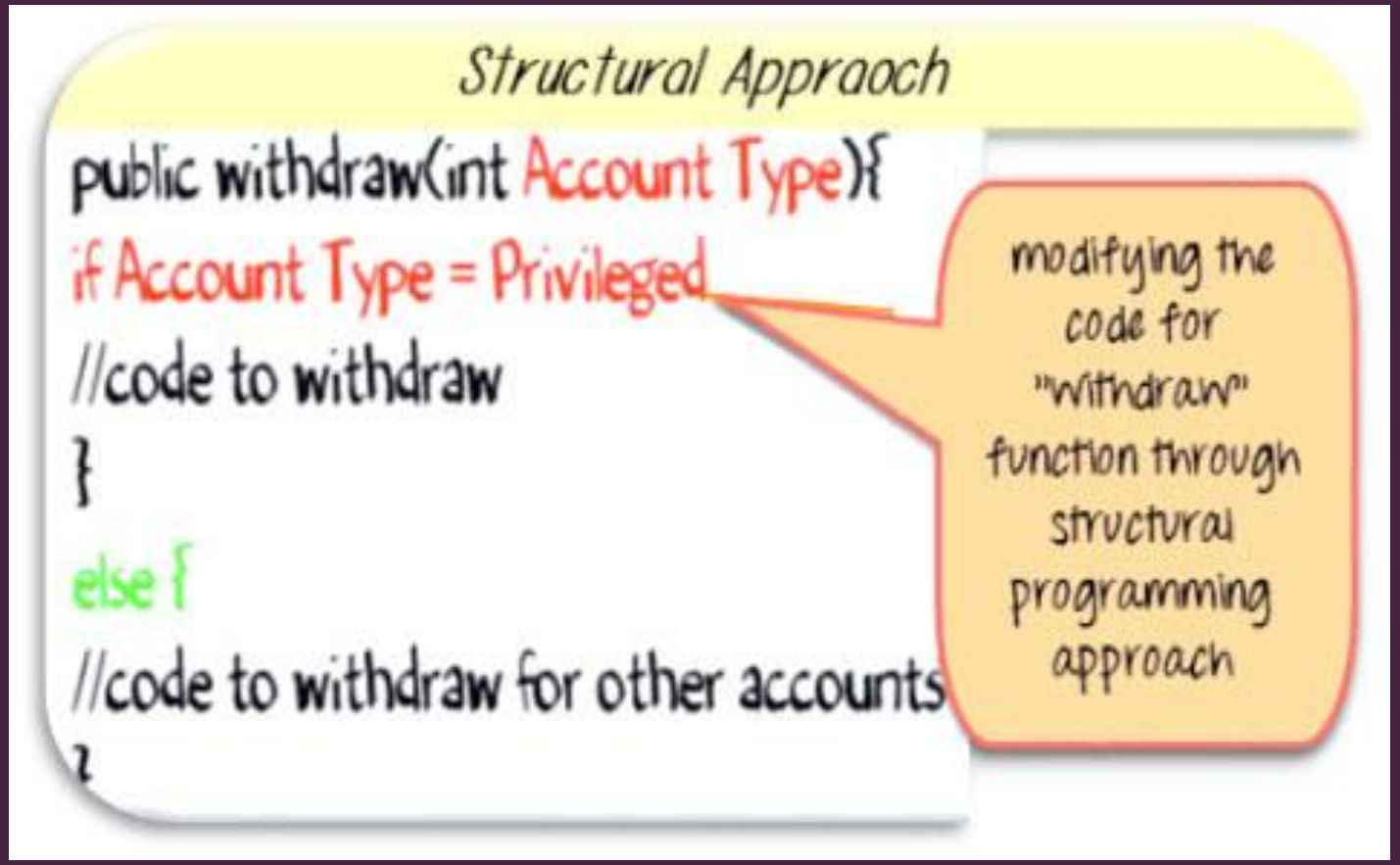
Change Request in Software

Now there is a change in the requirement specification for something that is so common in the software industry. You are supposed to add functionality *PRIVILEGED BANKING ACCOUNT* with Overdraft Facility. For a background, overdraft is a facility where you can withdraw an amount more than available the balance in your account.



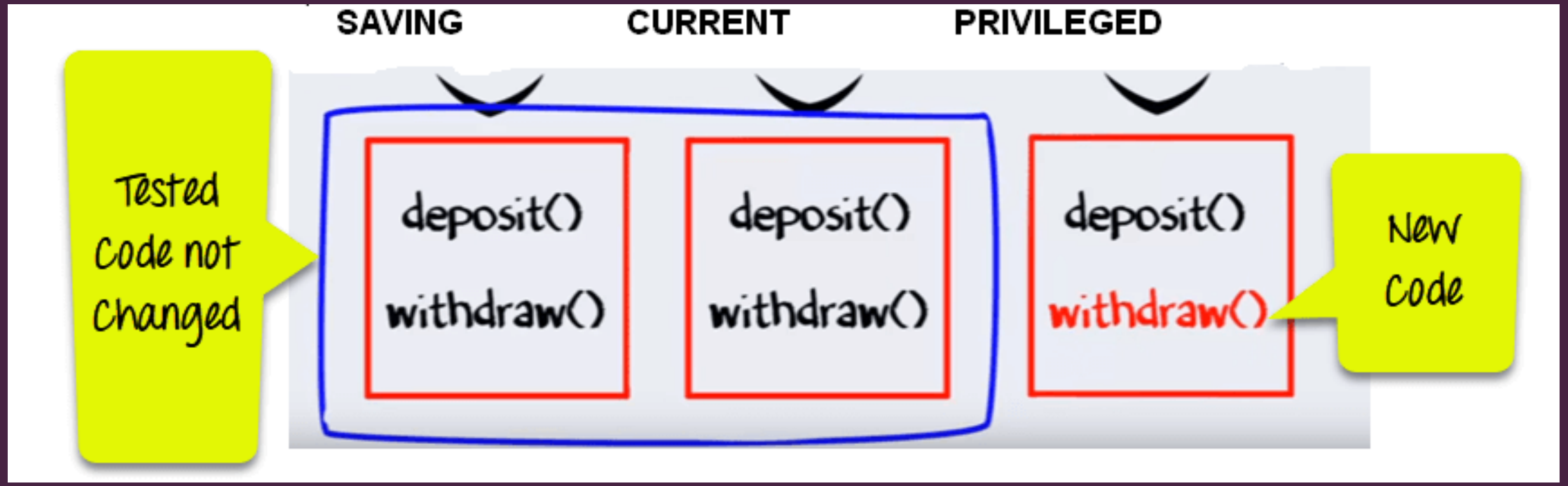
Structural approach

Using functional approach, I have to modify my withdraw function, which is already tested and baselined. And add a method like below will take care of new requirements.



OOP's approach

Using OOP's approach, you just need to write a new class with unique implementation of withdraw function. We never touched the tested piece of code.

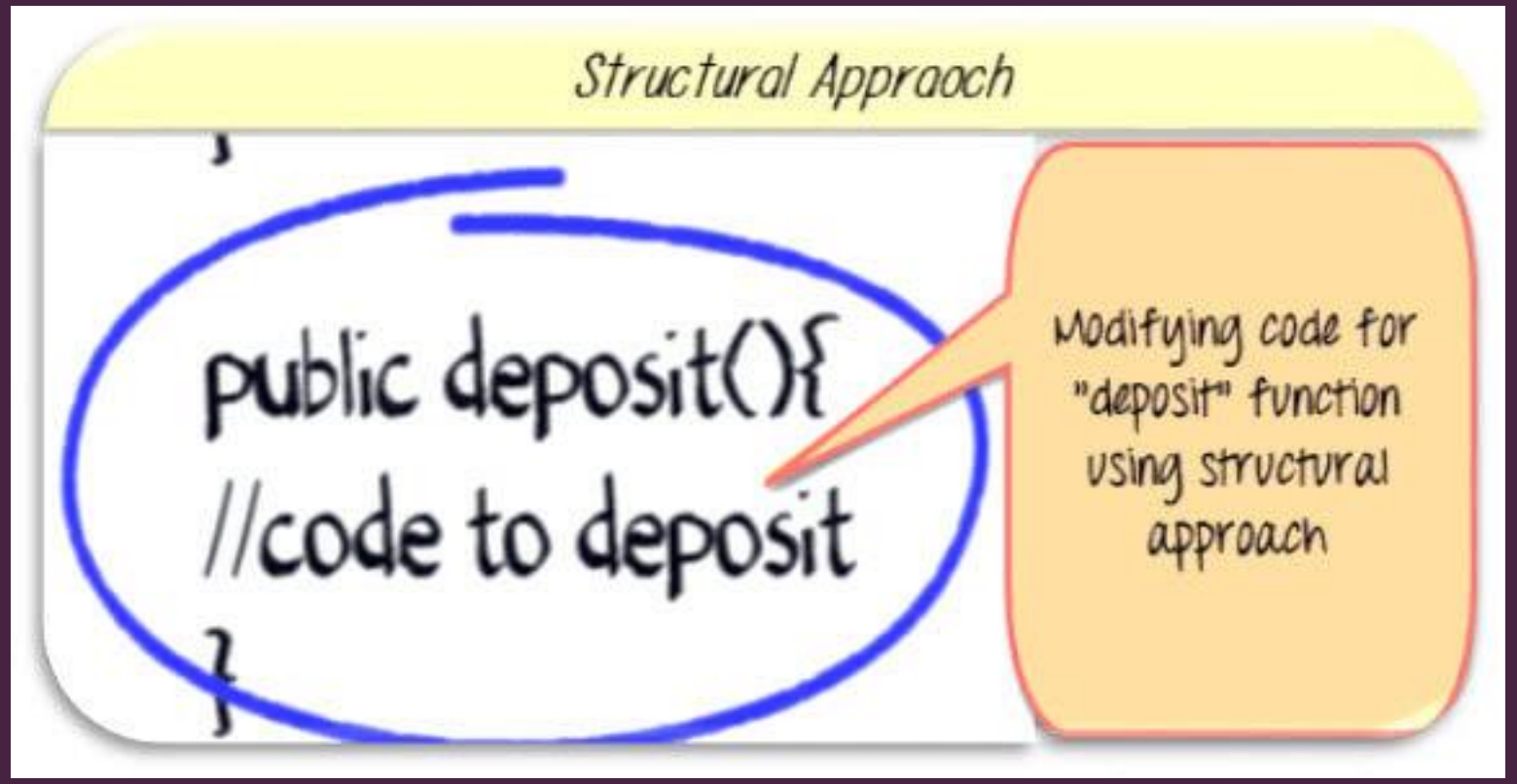


Another Change Request

What if the requirement changes further? Like to add *CREDIT CARD ACCOUNT* with its own unique requirement of deposits.

Structural approach

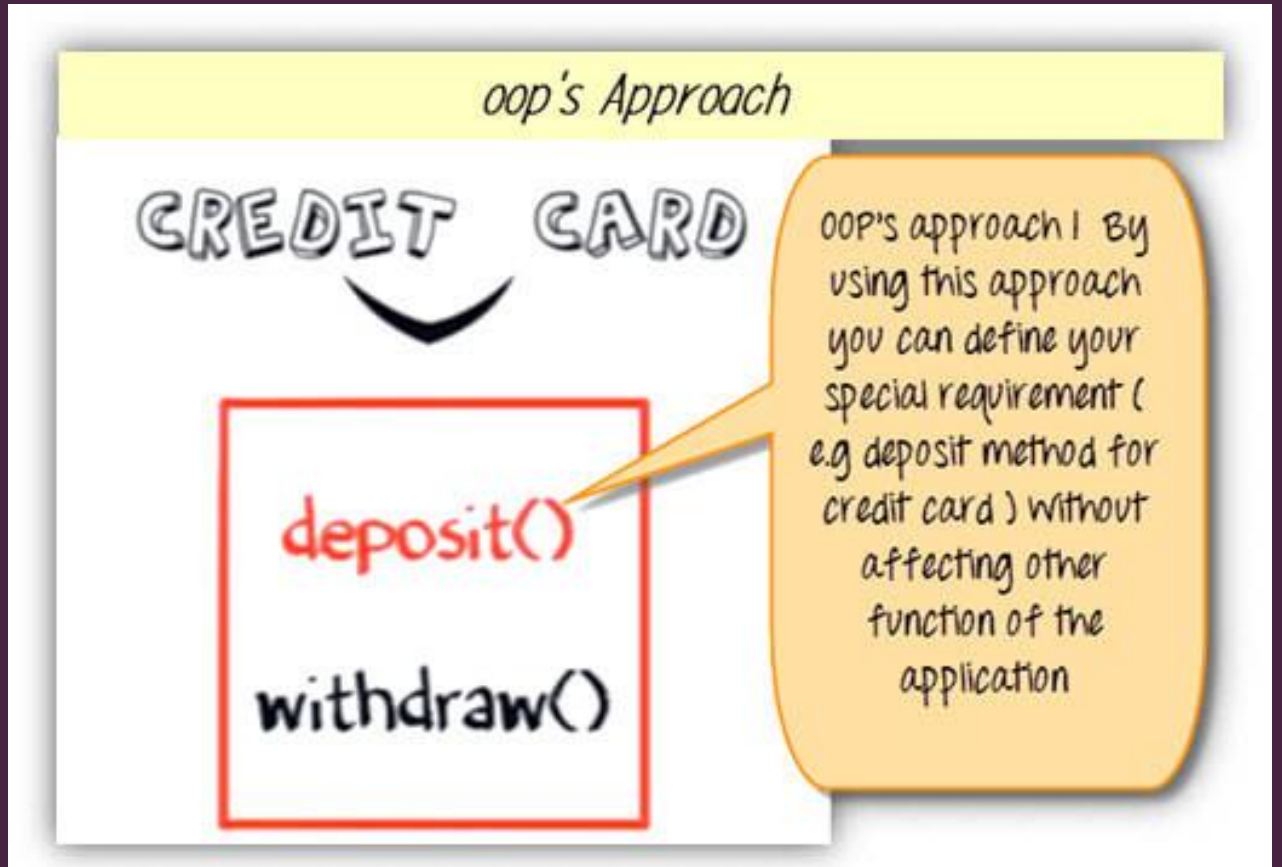
Using structural approach you have to change tested piece of deposit code again.



OOP's approach

But using object-oriented approach, you will just create a new class with its unique implementation of deposit method (highlighted red in the image below).

So even though the structural programming seems like an easy approach initially, OOP's wins in a long term.

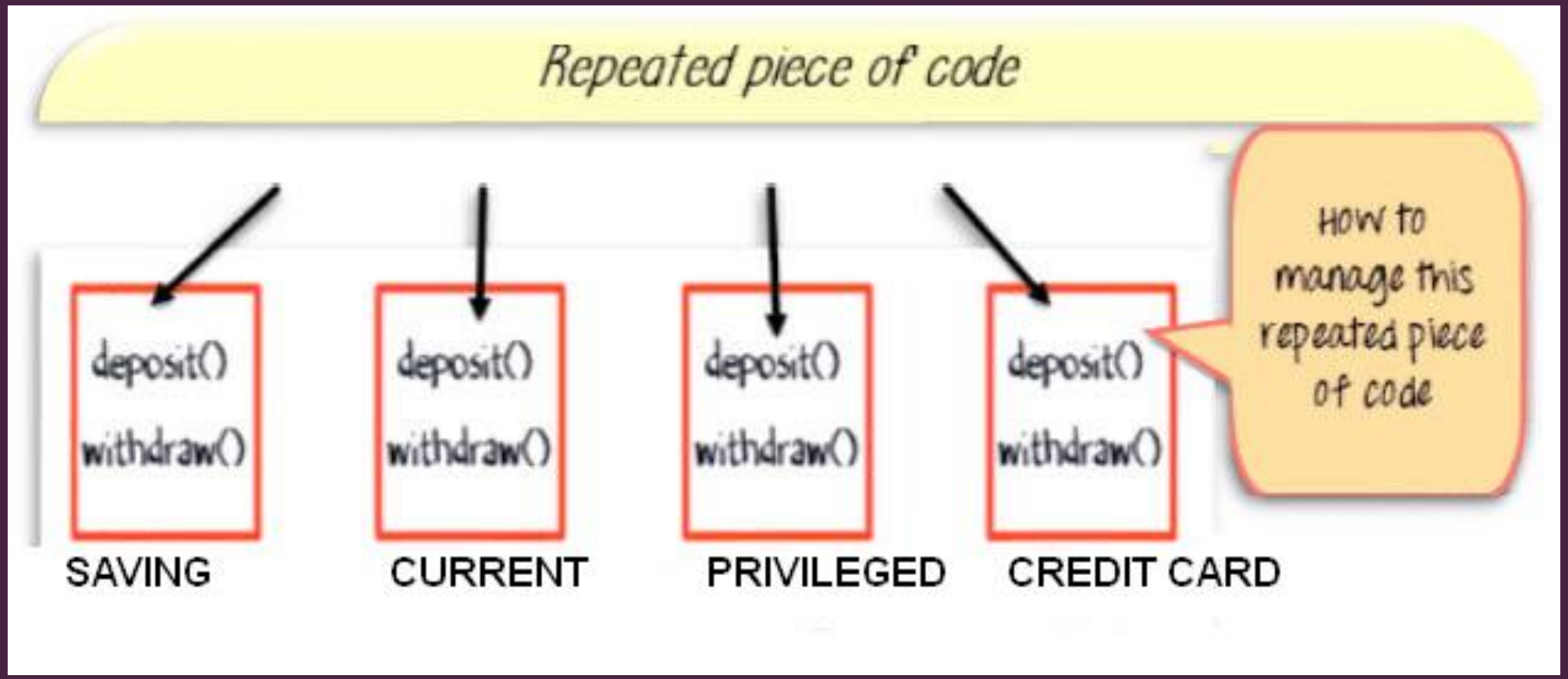


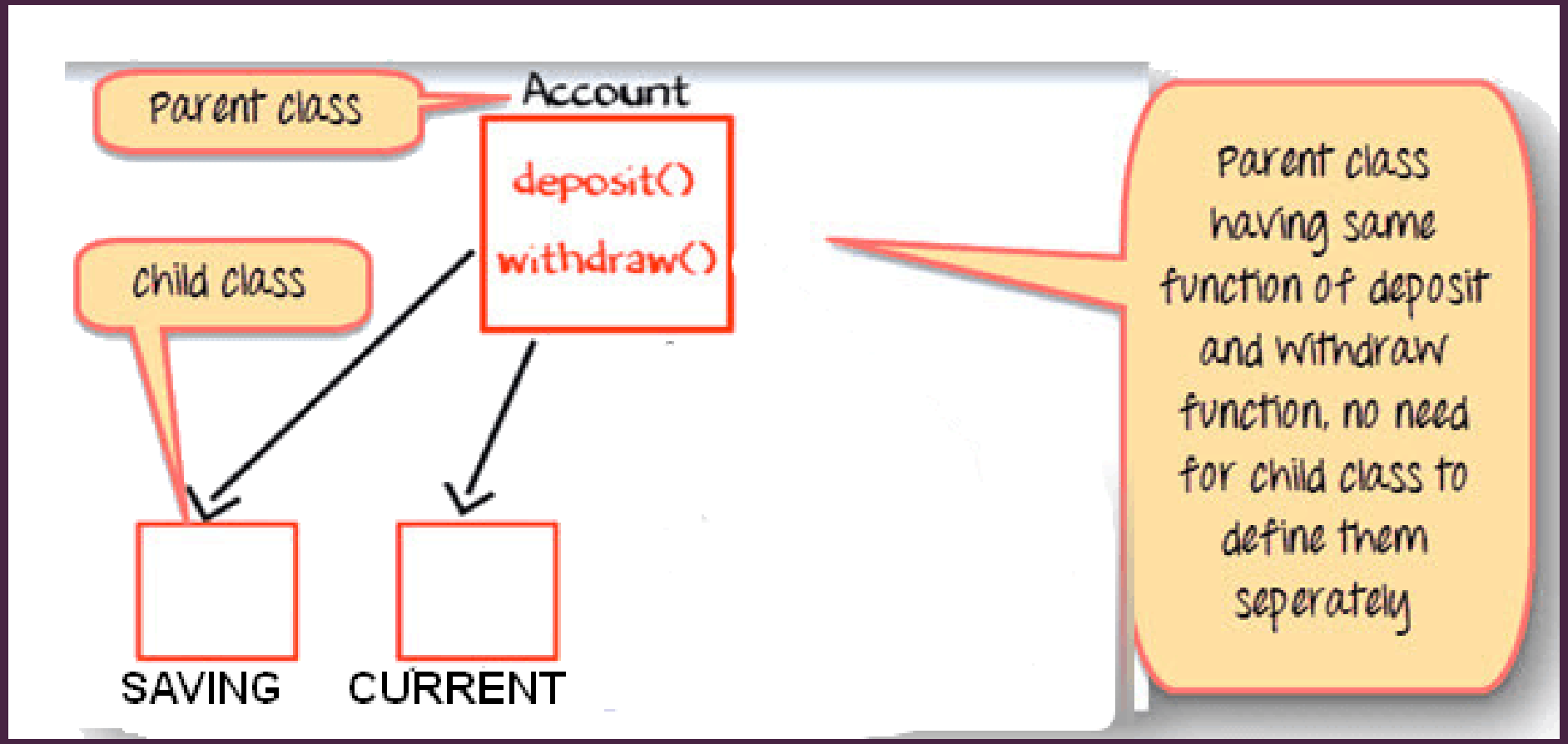
Advantage of Inheritance in OOPs

But one may argue that across all classes, you have a repeated pieces of code.

To overcome this, you create a parent class, say “Account" and implement the same function of deposit and withdraw. And make child classes inherited "account" class. So that they will have access to withdraw and deposit functions in account class.

The functions are not required to be implemented individually. This is *Inheritance in java*.


Repeated piece of code
deposit()
withdraw()
deposit()
withdraw()
deposit()
withdraw()
deposit()
withdraw()
How to manage this repeated piece of code
SAVING
CURRENT
PRIVILEGED
CREDIT CARD


Parent class
Account
deposit()
withdraw()
child class
SAVING
CURRENT
Parent class having same function of deposit and withdraw function, no need for child class to define them seperately

Thank you ☺ Happy coding ☺