



Java – Inheritance

Presented by



Inheritance

- Reusability and extendability
- Super class and Sub class
- super key word



Terminology

A class can be defined as a "subclass" of another class.

- The subclass inherits all data attributes of its superclass
- The subclass inherits all methods of its superclass
- The subclass inherits all associations of its superclass

A sub class can

- Add new functionality
- Use inherited functionality
- Override inherited functionality

superclass:

Person
- name: String
- dob: Date

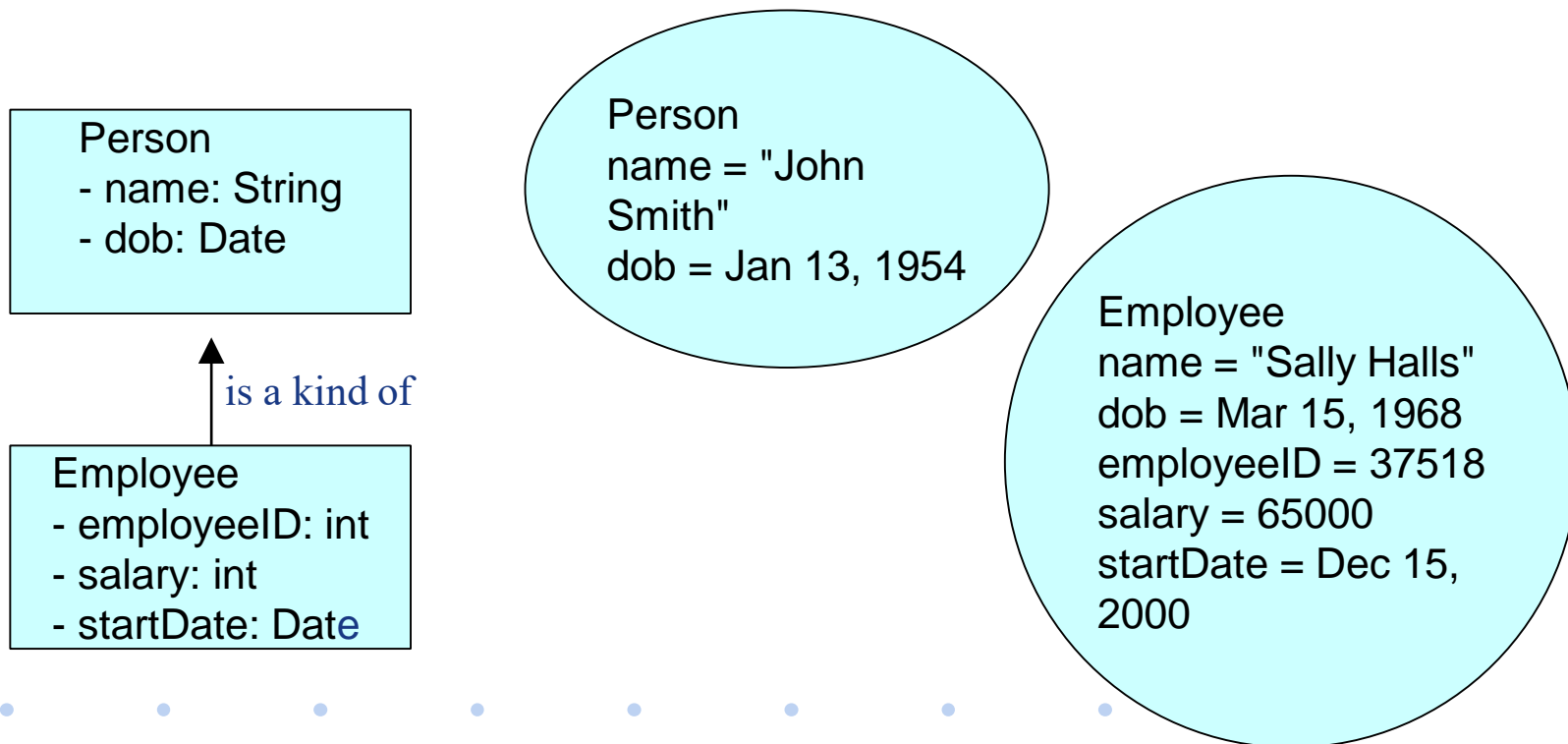
subclass:

Employee
- employeeID: int
- salary: int
- startDate: Date



What really happens?

- An Employee object inherits all of the attributes, methods and associations of Person



Example

```
public class Person{  
    private String name;  
    private Date dob;  
    .....  
}
```

```
public class Employee extends Person{  
    private int employeeID;  
    private int salary;  
    private Date startDate;  
    ...  
}
```

```
Employee anEmployee = new Employee();
```





Design different objects



Method Overriding

- **Method Overriding** allows a subclass to redefine methods of the same signature from the superclass.
- The key benefit of overriding is the ability to define/defer behavior specific to subclasses.
- An overridden method must have:
 - The same name
 - The same number of parameters and types
 - The same return type or its subtype.



Method Overriding - Example

```
public class Product {  
    private int productId;  
    private String name;  
    private double price;  
  
    // Constructors, setters, getters and  
    // other methods  
    public boolean isExpensive() {  
        return false;  
    }  
}
```

Television class overrides isExpensive() method of Product to define specific behavior of finding if Television is expensive or not.

```
public class Television extends Product {  
    private String screenType;  
    private String screenSize;  
    // Constructors, setters, getters and  
    // other methods  
  
    @Override  
    public boolean isExpensive() {  
        if( screenType.equals("CRT") &&  
            getPrice() > 20000.00) {  
            return true;  
        } else if( screenType.equals("LCD")  
            && getPrice() > 40000.00) {  
            return true;  
        } else if( screenType.equals("LED")  
            && getPrice() > 60000.00) {  
            return true;  
        }  
        return false;  
    }  
}
```



Method Overriding

- **The overridden method can't be less accessible.**
 - Access level must be the same, or more visible.
 - That means you can't, for example, override a public method and make it private.
 - Why?: the code invoking it *thinks* (at compile time) is a public method. If suddenly at runtime the JVM slammed the door shut because the overriding version called at runtime is private!!!



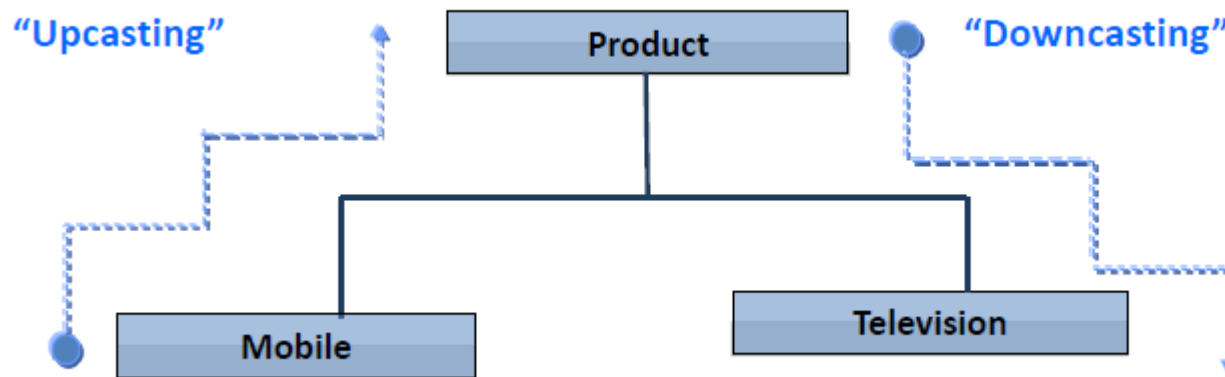
Object / Reference Casting

- To upcast a Mobile object, all you need to do is assign the object to a reference variable of type Product.

```
Product product = new Mobile();           // upcasting
```

- ```
Mobile mobile = (Mobile) product;
```

 // downcasting



# Object Typecasting – Reference Casting

Given:

```
public class A {
 public void first(){
 System.out.println("First Method");
 }
 public void second() {
 System.out.println("Second method");
 }
}

public class B extends A {
 public void second(int data) { // overloading
 System.out.println("Second method with data");
 }
}
```

What is the output of running the following statements?

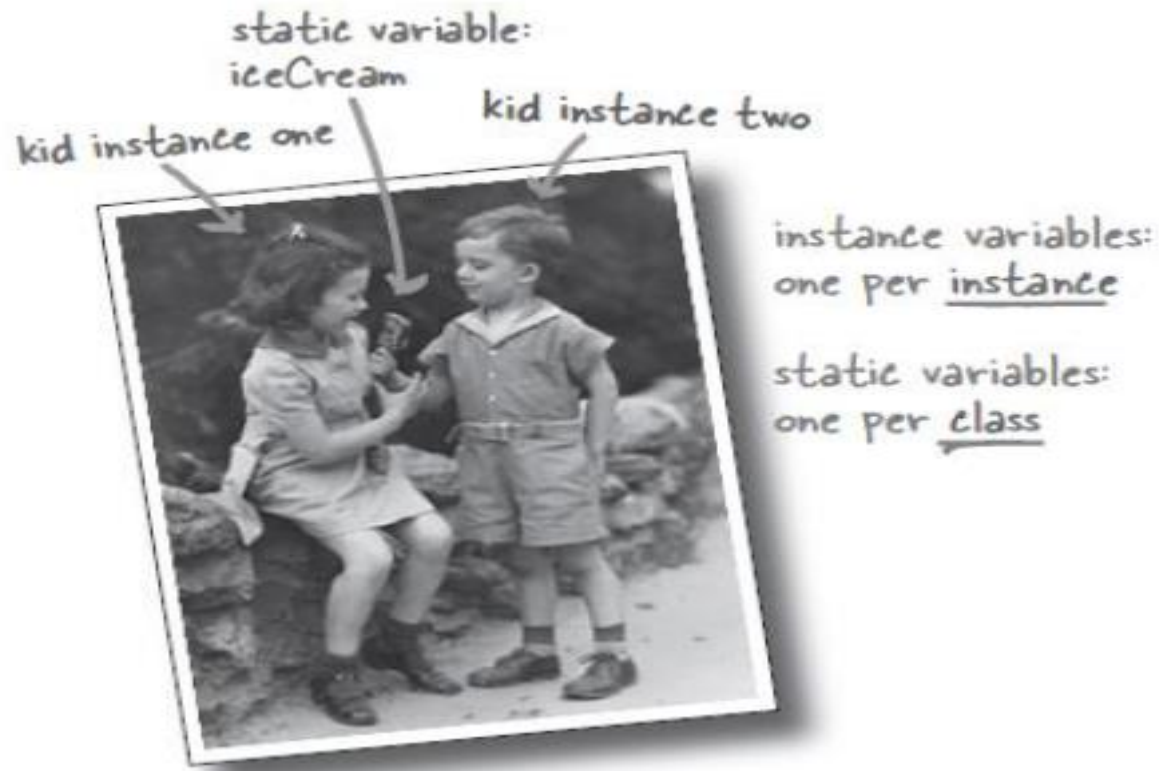
1) A obj = new B();  
 obj.second();  
 obj.second( 22 );

2) B nobj = new B();  
 nobj.second();  
 nobj.second( 22 );



# Static Members

- Static variables are shared by all instances of a class



# Static Variable

```
public class Employee {
 private String name;
 private static int count = 0;
 public Employee() {
 count++;
 }
 public Employee(String name) {
 this.name = name;
 count++;
 }
 public String getName() {
 return name;
 }
 public static int getCount() {
 return count;
 }
}
```

The static count is initialized only once when the class is loaded and not each time a new instance is created

It will keep incrementing each time the constructor is called and won't be reset to 0.



# Static Members

- Java is object-oriented, but once in a while you have a special case, typically a utility method (like the Math methods), where there is no need to have an instance of the class.
- The keyword `static` lets a method run *without any instance of the class*.
- A static method means "behavior not dependent on an instance variable, so no instance/object is required. Just the class."
- Example:
- The following method declared in Math class

is declared as static

```
public static int min(int a, int b) {
 return (a <= b) ? a : b;
}
```

**Math.min(42, 36);**

Use the Class name, rather than a reference variable name.



# Static methods in Inheritance

Example:

```
public class Animal {
 public static void testClassMethod() {
 System.out.println("The class" + " method in Animal.");
 }
}

public class Cat extends Animal {
 public static void testClassMethod() {
 System.out.println("The class method" + " in Cat.");
 }
}

public static void main(String[] args) {
 Animal myAnimal = new Cat();
 myAnimal.testClassMethod(); --> Animal.testClassMethod();
}
```

- The output from this program is as follows:
  - The class method in Animal.



# Static Members in Inheritance

- Answer this:

Given :

```
class A {
 public static void test() {
 System.out.println("test method of A");
 }
}

class B extends A {
 public static void test() {
 System.out.println("test method of B");
 }
}
```

What is the output?

```
A obj = new B(); // 1
obj.test(); //2
```





# Access Specifiers

| Visibility                               | Public | Protected | Default | Private |
|------------------------------------------|--------|-----------|---------|---------|
| From Within the Same Class               | Yes    | Yes       | Yes     | Yes     |
| From any class in the Same Package       | Yes    | Yes       | Yes     | No      |
| From a sub-class outside the Package     | Yes    | Yes       | No      | No      |
| From a non sub-class outside the Package | Yes    | No        | No      | No      |



# Final

- The final keyword can be used in many context.
  - Attribute
    - you cannot change the value once assigned [ constant ]
    - Example: `public static final double PI = 3.14;`
  - Method
    - You cannot override a final method
    - Example: `public final Boolean login( ) {`
  - Class
    - You cannot inherit a final class
    - Example: `public final class String {`



# Abstraction – Abstract Classes

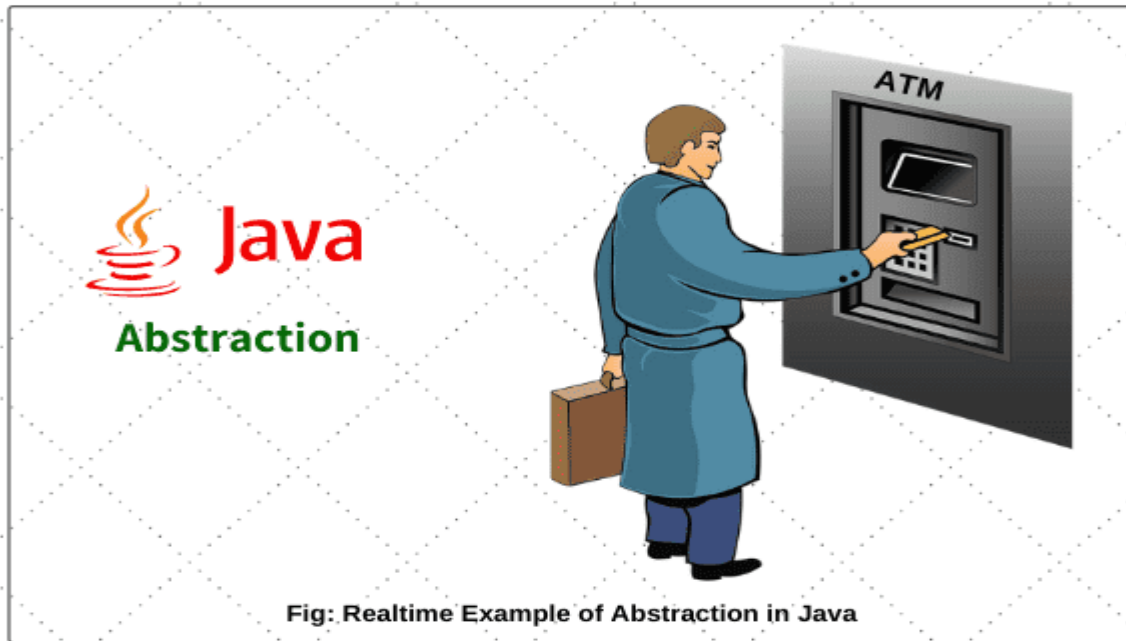
**Abstraction** - Show only essential attributes and hides unnecessary information

The main purpose of abstraction is hiding the unnecessary details from the users.

It helps in reducing programming complexity and efforts.



# Abstraction – Real world scenarios



- We all use an ATM machine for cash withdrawal, money transfer, retrieve min-statement, etc in our daily life.
- But we don't know internally what things are happening inside ATM machine.



# Abstraction – Realworld scenarios

Internal details of Car



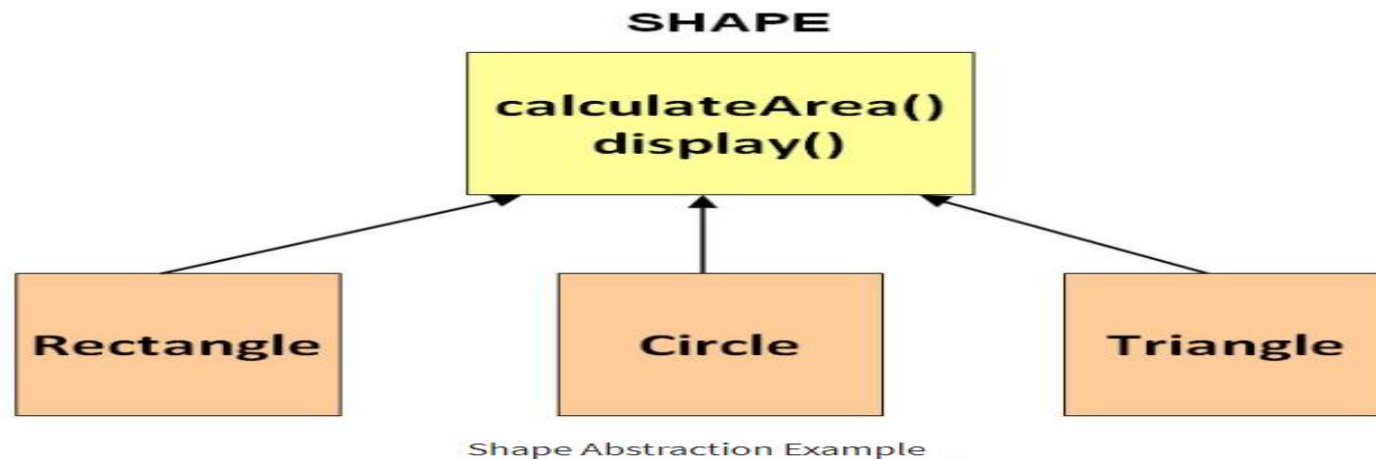
Abstraction of Car



- The car owner only needs to know about some basic things that are necessary for him, i.e., to drive a car accelerator, clutch, brakes.
- He doesn't need to know about the inner mechanism of the car.
- This is abstraction.



# Abstraction – Realworld scenarios

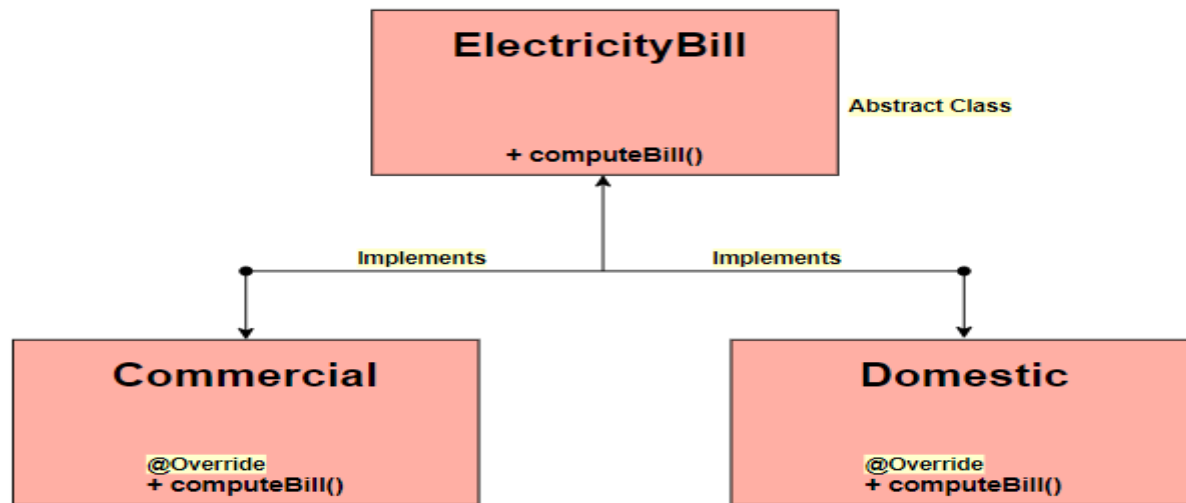


- The Shape class is created to save on common attributes and methods shared by the three classes Rectangle, Circle, and Triangle.
- The method calculateArea() is one such method shared by all three sub classes and present in Shape class, where the formulae are different.

• \*\*\* Show ShapesApp



# Abstraction – Realworld scenarios



- The electricity bill calculations are different for commercial usage and domestic usage.
- The method `computeBill()` is an abstract method shared by two sub classes.
- \*\* Show  
AbstractClassesElectricityBillApp



# Abstract Methods

- An abstract method is an generic method which would be useful for subclasses.
- An abstract method has no body!
- An abstract method means the method must be overridden.

“All subtypes of this type have **THIS** method”

```
public abstract class Product {

 private int productId;
 private String name;
 private double price;

 // remaining code
 public abstract boolean isExpensive() ;
}
```

The method `isExpensive()` is made abstract because it is generic. We cannot provide logic in this method to find if product is expensive or not.

Concrete classes extending from Product [Television, Mobile, ..] will override this method and provide appropriate logic to find if its expensive or not.





# Abstract Methods

- Any class which extends an abstract class has to override all abstract methods declared in super class, else that class should also be declared as abstract.
- This enforces all derived classes to provide a common signature [public boolean isExpensive() ]

Television class overrides public boolean isExpensive() method which is declared abstract in Product class

```
public class Television extends Product {
 private String screenType;
 private String screenSize; // in inches

 @Override
 public boolean isExpensive() {
 //CRT Television is expensive if cost is more than 10,000/-
 if(screenType.equals("CRT") && getPrice() > 20000.00) {
 return true;
 } else if(screenType.equals("LCD") && getPrice() > 40000.00) {
 return true;
 } else if(screenType.equals("LED") && getPrice() > 60000.00) {
 return true;
 }
 return false;
 }
}
```

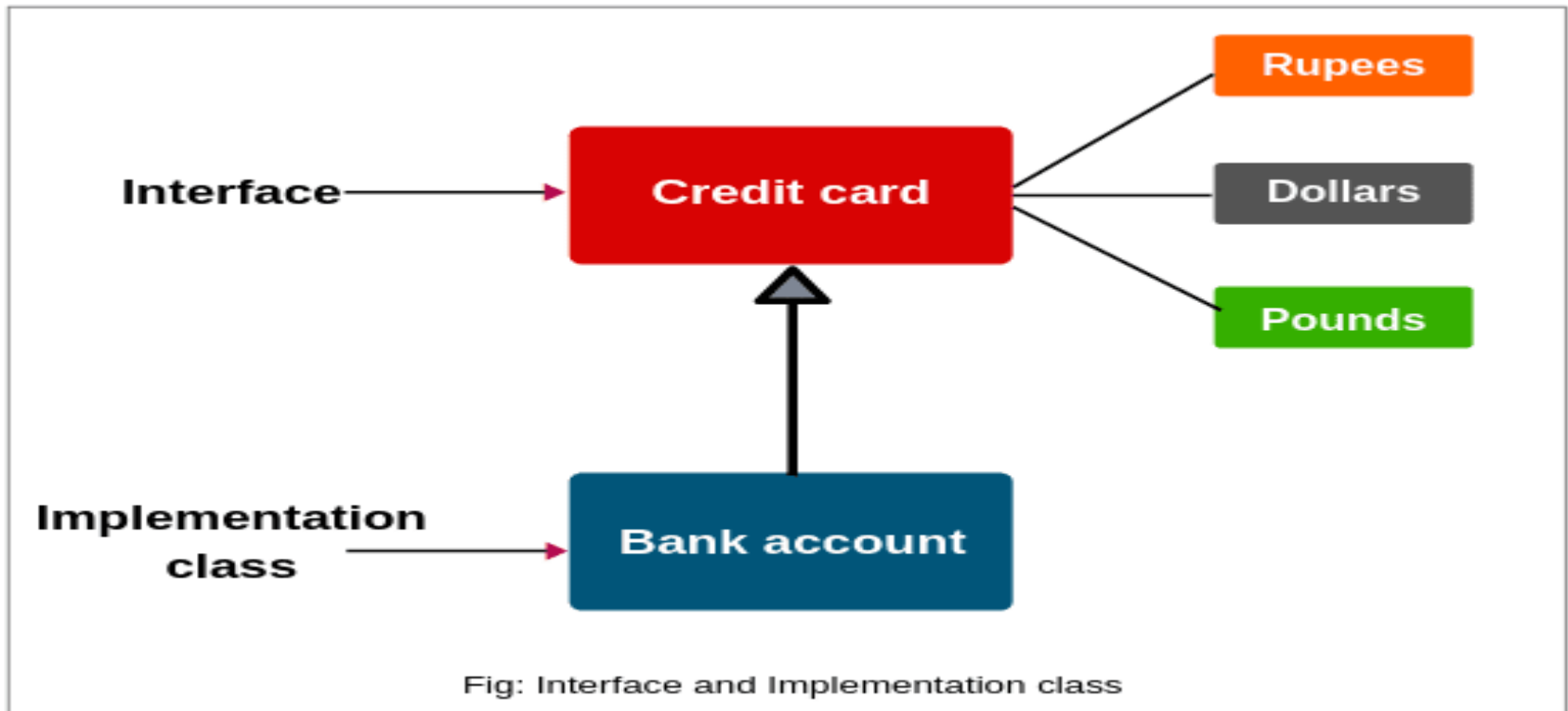


# Interfaces

- The **interface** is another way to achieve abstraction in JAVA.
- An interface is used for full abstraction.
- An interface is a blueprint of a class.
- An interface can have methods and variables like class, but the methods of an interface are by default abstract.
- It means the interface can contain only an **abstract method**(Method without body).



# Interfaces – Realworld scenarios



# Interfaces – Realworld scenarios

- Suppose you have a credit card.
  - In a shop, you can easily pay in rupees by using your credit card.
  - If you go to another shop where dollars are accepted, you can also pay in dollars.
  - The same credit card can be used to pay in pounds also.
- \*\* Show InterfacesBankingApp



# Interface Extension

- You can add new methods to an interface by using inheritance, and you can also combine several **interfaces** into a new **interface** with inheritance.

```
interface Fight {
 public void fight();
}
interface Dance extends Fight {
 public void dance();
}

class Actor implements Dance {
 @Override
 public void fight() {
 // fight implementation
 }

 @Override
 public void dance() {
 // dance implementation
 }
}
```

Dance is a simple extension of Fight!!!

Actor implements Dance, and hence he knows to dance and fight



