

Object Oriented Programming

Prepared by Dr. Yasser Abdelhamid

IT223
Object Oriented Programming

Lecture 08



OUTLINE

- Abstraction
- Abstract classes
- Interfaces
- Multiple inheritance
- Interfaces vs abstract classes

ABSTRACTION

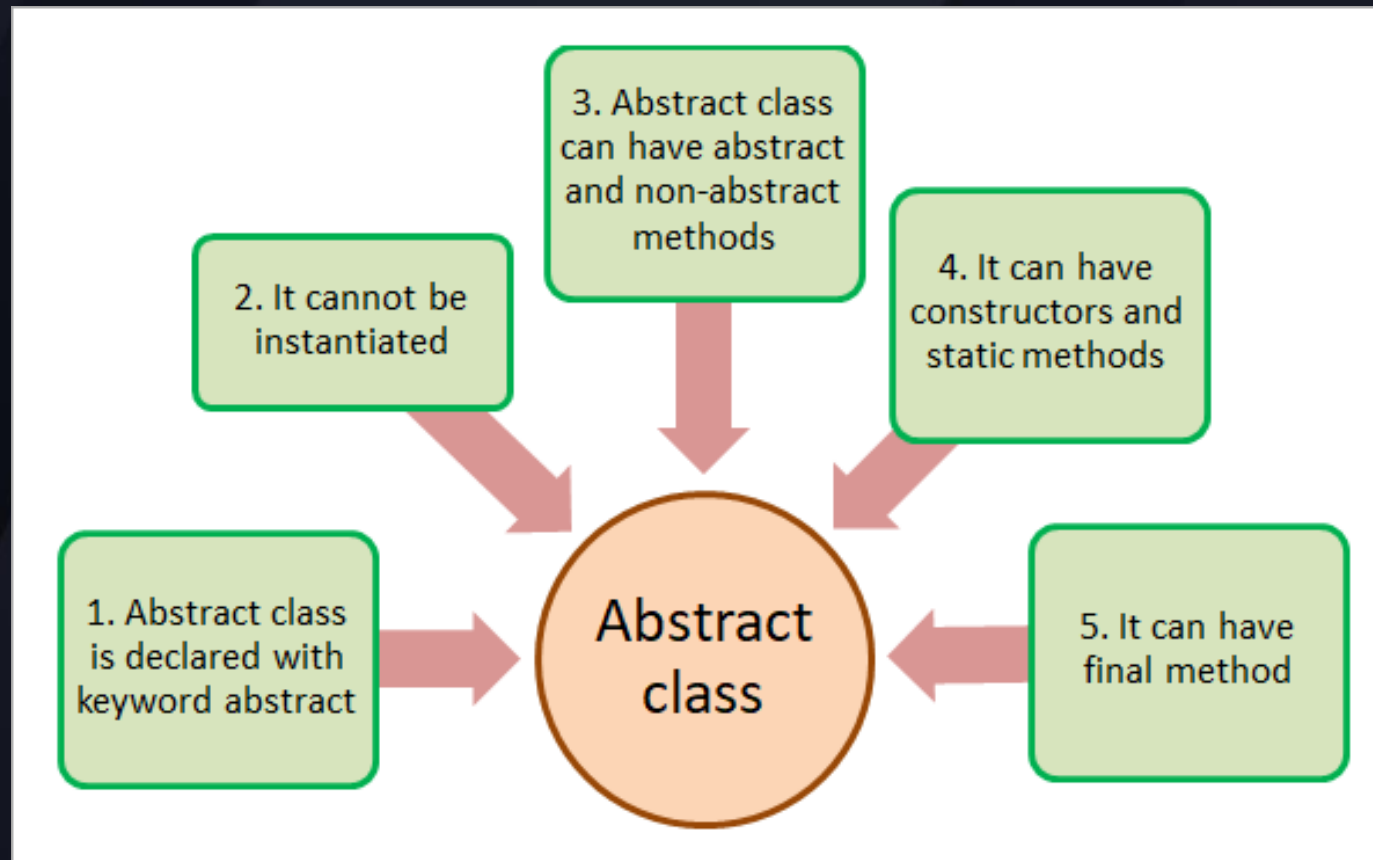
ABSTRACTION

- ❖ Sometimes we may come across a situation where we cannot provide implementation to all the methods in a class.
- ❖ We want to leave the implementation to a class that extends it.
- ❖ In such case we declare a class as **abstract**.

ABSTRACTION

- ❖ Abstraction is a process of **hiding the implementation** details and **showing only functionality** to the user.
- ❖ Abstraction lets you focus on **what** the object does **instead of how** it does it.
- ❖ Abstraction can be implemented using
 - Abstract classes
 - Interfaces

ABSTRACT CLASSES



ABSTRACT CLASS

- ❖ An abstract class is declared using **abstract** keyword.
- ❖ Abstract classes may have
 - **abstract** and
 - **non-abstract methods**.
- ❖ Abstract classes **cannot be instantiated**.

ABSTRACT CLASS

- ❖ Any class which extends abstract class must **override all its abstract methods.**
- ❖ An abstract class can contain **0 or more abstract methods.**
- ❖ Abstract class declaration
 - Example :
 - `abstract class MyClass{}`

ABSTRACT METHODS

- ❖ Abstract methods are **only declared in abstract classes.**
- ❖ Declaring abstract methods in non-abstract classes results in a **compilation error.**

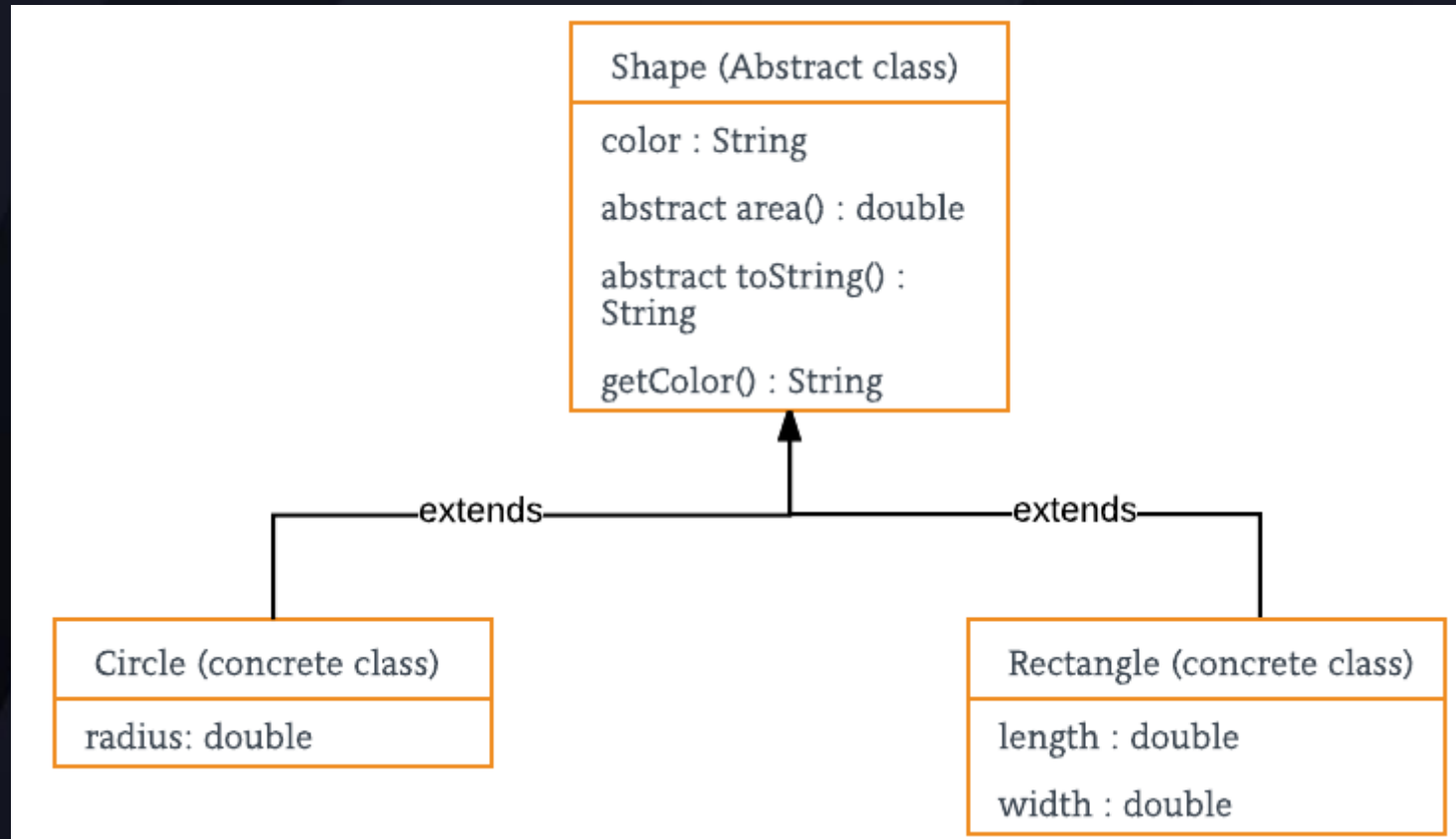
EXAMPLE

```
// Example of abstract class

abstract class Bike{
    abstract void run(); // must be overridden
}

class Honda4 extends Bike{
    void run(){System.out.println("Overriding abstract method");}
    public static void main(String args[]){
        Bike obj = new Honda4();
        obj.run();
    }
}
```

EXAMPLE



EXAMPLE SHAPE CLASS

```
public abstract class Shape {  
    private String name;  
    public Shape() {  
        this.name = "Unknown shape";  
    }  
    public Shape(String name) {  
        this.name = name;  
    }  
    public String getName() {  
        return this.name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    // Abstract methods  
    public abstract void draw();  
    public abstract double getArea();  
    public abstract double getPerimeter();  
}
```

EXAMPLE RECTANGLE CLASS

```
public class Rectangle extends Shape {  
    private double width;  
    private double height;  
    public Rectangle(double width, double height) {  
        // Set the shape name as "Rectangle"  
        super("Rectangle");  
        this.width = width;  
        this.height = height;  
    }  
    ...  
}
```

EXAMPLE RECTANGLE CLASS (CONT.)

```
// Provide an implementation for inherited abstract draw() method
public void draw() {
    System.out.println("Drawing a rectangle...");
}

// Provide an implementation for inherited abstract getArea() method
public double getArea() {
    return width * height; }

// Provide an implementation for inherited abstract getPerimeter() method
public double getPerimeter() {
    return 2.0 * (width + height); }
}
```

EXAMPLE CIRCLE CLASS

```
public class Circle extends Shape {  
    private double radius;  
    public Circle(double radius) {  
        super("Circle");  
        this.radius = radius;  
    }  
    ...  
}
```

EXAMPLE CIRCLE CLASS (CONT.)

```
// Provide an implementation for inherited abstract draw() method
public void draw() {
    System.out.println("Drawing a circle...");
}
// Provide an implementation for inherited abstract getArea() method
public double getArea() {
    return Math.PI * radius * radius;
}
// Provide an implementation for inherited abstract getPerimeter() method
public double getPerimeter() {
    return 2.0 * Math.PI * radius;
}
}
```

USING SHAPE ABSTRACT CLASS

```
Shape s = new Rectangle(2.0, 5.0);
```

```
/* Late binding
```

```
    s.getArea() will call the getArea() method  
    of the Rectangle class.
```

```
*/
```

```
double area = s.getArea();
```

EXAMPLE: HANDLING ARRAY OF SHAPES

```
public class ShapeUtil {  
    public static void drawShapes(Shape[] list) {  
        for (int i = 0; i < list.length; i++) {  
            // Draw a shape, no matter what it is  
            list[i].draw(); // Late binding  
        }  
    }  
    ...  
}
```

EXAMPLE: HANDLING ARRAY OF SHAPES(CONT.)

```
public static void printShapeDetails(Shape[] list) {  
    for (int i = 0; i < list.length; i++) {  
        // Gather details about the shape  
        String name = list[i].getName(); // Late Binding  
        double area = list[i].getArea(); // Late binding  
        double perimeter = list[i].getPerimeter(); // Late binding  
        // Print details  
        System.out.println("Name: " + name);  
        System.out.println("Area: " + area);  
        System.out.println("Perimeter: " + perimeter);  
    }  
}
```

LATE BINDING

- ❖ In Java, late binding, also known as **dynamic binding** or **runtime polymorphism**, refers to the process of determining the appropriate method or function to call **at runtime**, based on the actual type of the object rather than the reference type.
- ❖ Late binding allows for more flexibility and extensibility in object-oriented programming.

TESTING SHAPEUTIL CLASS

```
public class ShapeUtilTest {  
    public static void main(String[] args) {  
        // Create some shapes, draw, and print their details  
        Shape[] shapeList = new Shape[2]; Array of two shapes  
        shapeList[0] = new Rectangle(2.0, 4.0); // Upcasting  
        shapeList[1] = new Circle(5.0); // Upcasting  
        // Draw all shapes  
        ShapeUtil.drawShapes(shapeList);  
        // Print details of all shapes  
        ShapeUtil.printShapeDetails(shapeList);  
    }  
}
```

EXERCISE: BANK ACCOUNTS APPLICATION

- ❖ A bank account has a name, interest rate, and interest interval.
- ❖ Create an abstract class that implements bank accounts class.
- ❖ Create concrete bank account classes for the following types of accounts:
 - Savings account, interest rate 12%, monthly
 - Platinum account, interest rate 17%, quarterly.
- ❖ Create test class to use these classes.

INTERFACES

INTERFACES

- ❖ Interfaces are another way to **achieve abstraction** in Java.
- ❖ Interfaces cannot be used to create objects.
- ❖ Classes **implement** interfaces.
- ❖ On implementation of an interface, you must **override all of its methods**.
- ❖ An interface **cannot contain a constructor**.

INTERFACES IMPLEMENT ABSTRACTION

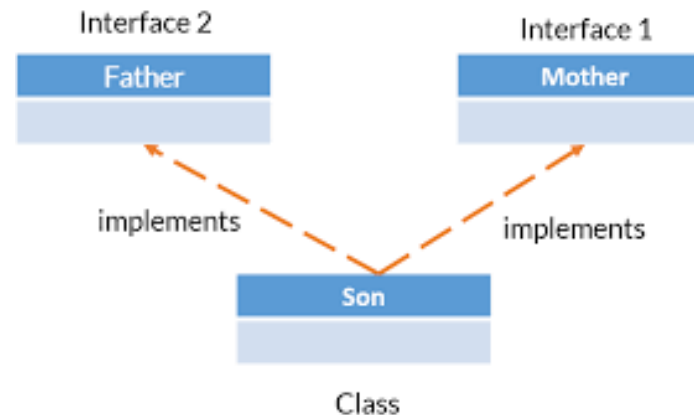
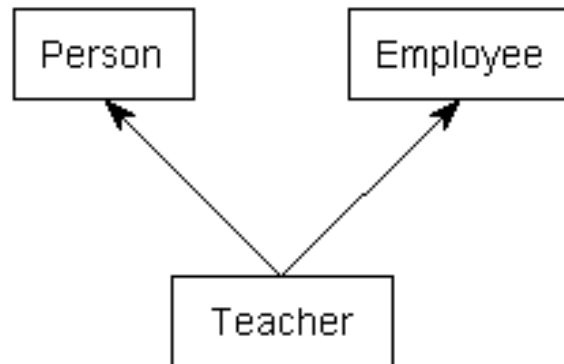


ATM Machine

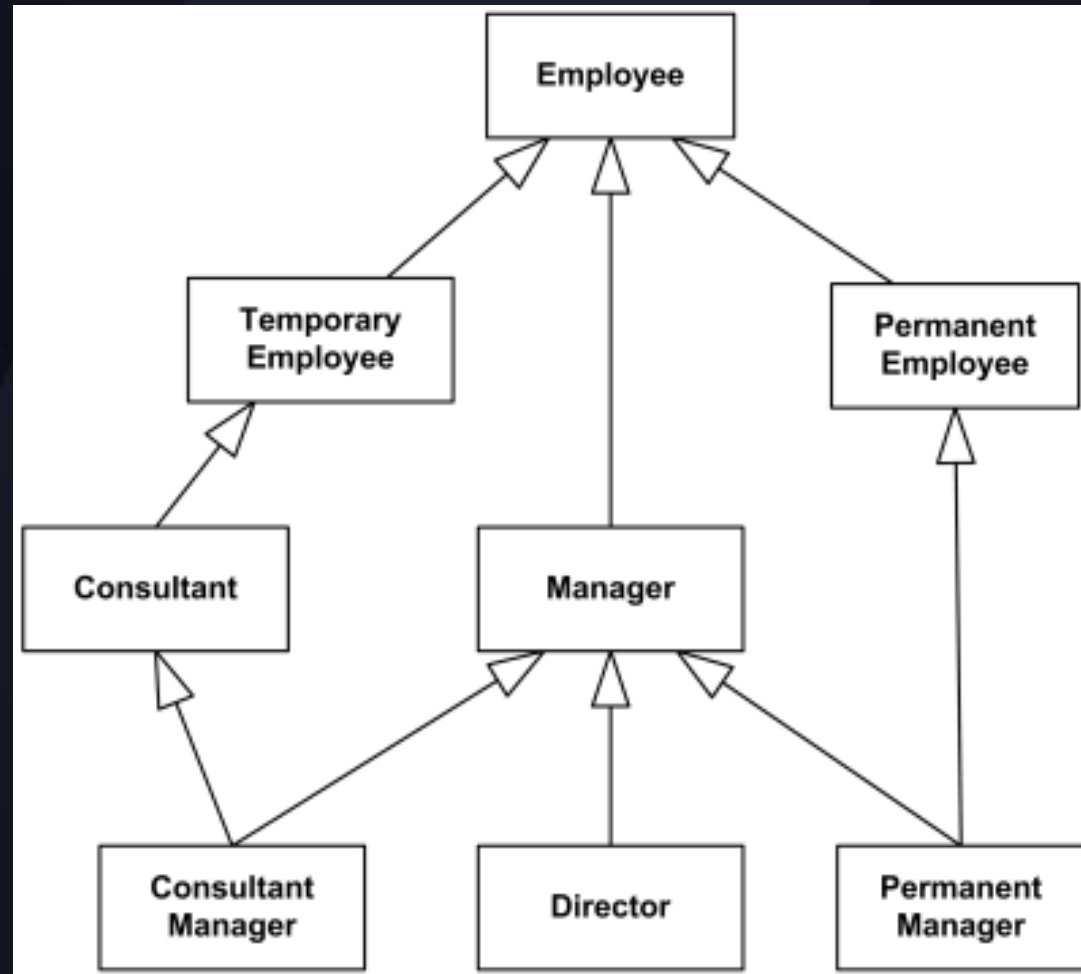


MULTIPLE INHERITANCE

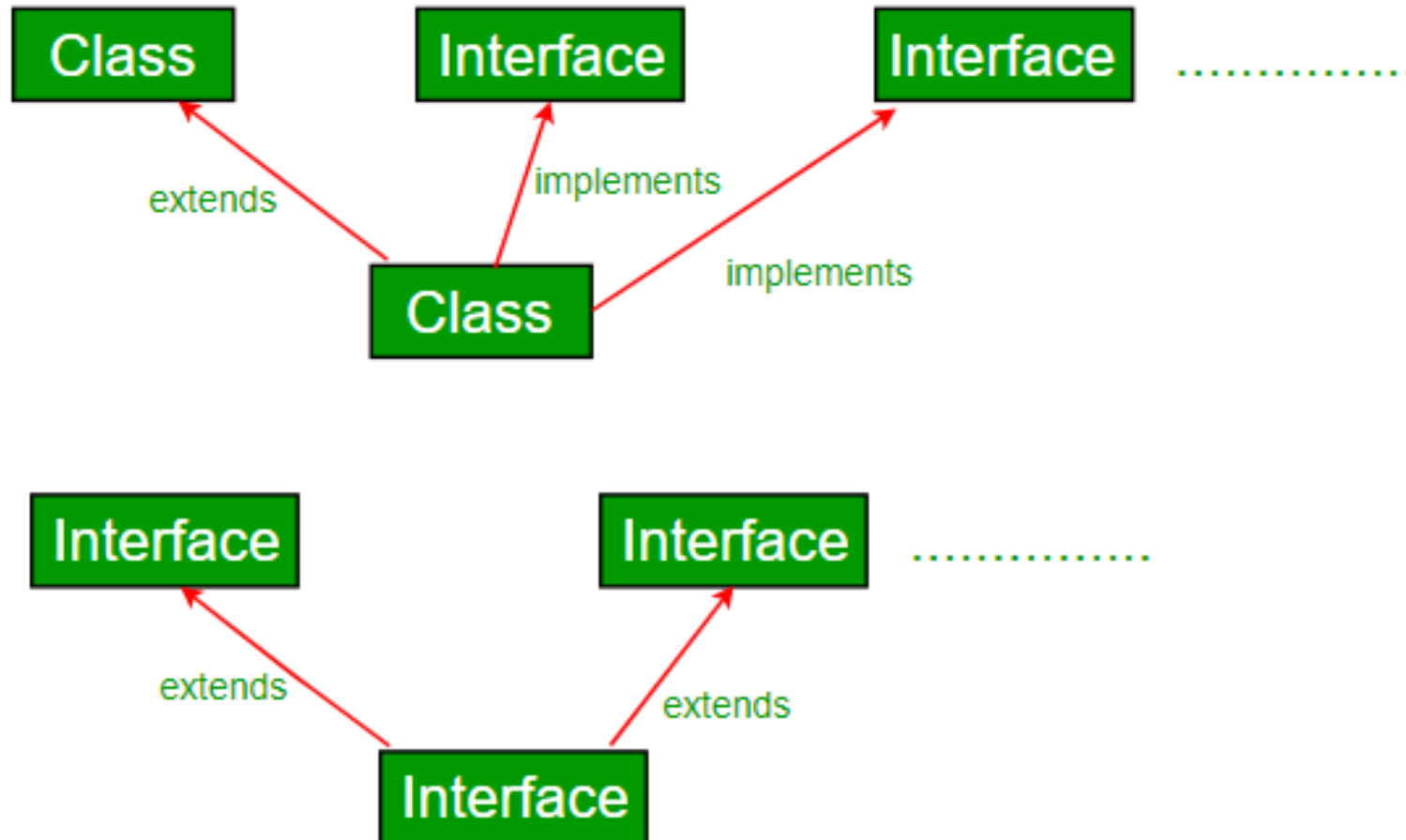
- ❖ Multiple inheritance can be achieved using interfaces.
- ❖ A class **can implement multiple interfaces**.



EXAMPLE



INTERFACES VS CLASSES



INTERFACES VS ABSTRACT CLASSES

Abstract Class

1. *abstract* keyword
2. Subclasses *extends* abstract class
3. Abstract class can have implemented methods and 0 or more abstract methods
4. We can extend only one abstract class



Interface

1. *interface* keyword
2. Subclasses *implements* interfaces
3. Java 8 onwards, Interfaces can have default and static methods
4. We can implement multiple interfaces



DECLARING AN INTERFACE

- ❖ Like a class, an interface can have a **public** or package-level (**default**) scope.
- ❖ The keyword **interface** is used to declare an interface.
- ❖ An interface declaration is implicitly **abstract**.

```
<modifiers> interface <interface-name> {  
    Constant-Declaration  
    Method-Declaration  
    Nested-Type-Declaration  
}
```

DECLARING INTERFACE MEMBERS

- ❖ An interface can have three types of members:
 - Constant fields
 - Abstract, static, and default methods
 - Static types (nested interfaces and classes)
- ❖ interface **cannot have mutable instances and class variables.**

CONSTANT FIELDS DECLARATIONS

- ❖ All fields in an interface are implicitly **public**, **static**, and **final**.
- ❖ You can access the fields in an interface using the dot notation in the form of:
 - **<interface-name>.<field-name>**.
- ❖ You **must initialize fields** at the time of declaration.

CONSTANT FIELDS DECLARATIONS

```
public interface Choices {  
    int YES = 1;  
    int NO = 2;  
}
```



```
public interface Choices {  
    public static final int YES = 1;  
    public static final int NO = 2;  
}
```

```
public class ChoicesTest {  
    public static void main(String[] args) {  
        System.out.println("Choices.YES = " + Choices.YES);  
        System.out.println("Choices.NO = " + Choices.NO);  
    }  
}
```

INTERFACE METHODS

- ❖ Interface methods are by default **abstract** and **public**.
- ❖ The absence of static and default modifiers makes a method abstract.

INTERFACE METHODS

```
public interface Player {  
    void play();  
    void stop();  
    void forward();  
    void rewind();  
}
```



```
public interface Player {  
    public abstract void play();  
    public abstract void stop();  
    public abstract void forward();  
    public abstract void rewind();  
}
```

STATIC METHODS DECLARATIONS

- ❖ From **Java 8**, you can have static methods in interfaces.
- ❖ A static method contains the static modifier.
- ❖ They are implicitly **public**.

STATIC METHODS DECLARATIONS

Static method definition

```
public interface Walkable {  
    // An abstract method  
    void walk();  
  
    // A static convenience method  
    public static void letThemWalk(Walkable[] list) {  
        for (int i = 0; i < list.length; i++) {  
            list[i].walk();  
        }  
    }  
}
```

Using static methods of interfaces

```
Walkable[] w = new Walkable[4];  
w[0] = new Person("Jack");  
w[1] = new Duck("Jeff");  
w[2] = new Person("John");  
w[3] = new Cat("Jace");  
  
// Let everyone walk  
Walkable.letThemWalk(w);
```

DEFAULT METHODS DECLARATIONS

- ❖ A default method provides a **default implementation** for the classes that implements the interface, but does not override the default method.

DEFAULT METHODS DECLARATIONS

```
interface SaySomething{
    // Default method
    default void defaultMsg(){
        System.out.println("Working...");
    }
    // Abstract method
    void sayMsg(String msg);
}

public class Person implements SaySomething{
    public void sayMsg(String msg){    // implementing abstract method
        System.out.println(msg);
    }
    public static void main(String[] args) {
        Person p = new Person();
        p.defaultMsg(); // calling default method
        p.sayMsg("Working hard..."); // calling abstract method
    }
}
```

MULTIPLE INHERITANCE

```
interface SaySomething{
    // Default method
    default void defaultMsg(){
        System.out.println("Working...");
    }
    // Abstract method
    void sayMsg(String msg);
}

// second interface
interface DoSomething{
    // Default method
    default void defaultAction(){
        System.out.println("Jumping...");
    }
    // Abstract method
    void doMsg(String msg);
}
```

```
class Person implements SaySomething, DoSomething {
    // implementing first interface
    public void sayMsg(String msg){    // implementing abstract method
        System.out.println(msg);
    }
    // can we override interface default method?
    public void defaultMsg(){
        System.out.println("Working... Working... Working...");
    }
    // implementing second interface
    public void doMsg(String msg){    // implementing abstract method
        System.out.println(msg);
    }
    // can we override interface default method?
    public void defaultAction(){
        System.out.println("Jumping... Jumping... Jumping...");
    }
    public static void main(String[] args) {
        Person p = new Person();
        p.defaultMsg(); // calling default method from first interface
        p.sayMsg("Working hard..."); // calling abstract method
        p.defaultAction(); // calling default method from second interface
        p.doMsg("Jumping hard...");
    }
}
```

SUMMARY

- Abstraction
- Abstract classes
- Interfaces
- Multiple inheritance
- Interfaces vs abstract classes



Thank You

www.midocean.ae

