

Week 8: Abstract Classes and Interfaces

Objectives

- At the end of the lecture, the student should be able to:
 - Define the terms abstract classes and interfaces
 - Write java codes for an abstract class
 - Write java codes for interfaces
 - Use existing abstract classes to create concrete classes
 - Write codes for a class implementing an interface
 - Differentiate between abstract classes and interfaces

Abstract Classes

- Sometimes, a defined class represents an abstract concept and, as such, should not be instantiated.
- For example, food in the real world.

Have anyone ever seen an instance of food? No. What is seen instead are instances of carrot, apple, and chocolate. Food represents the abstract concept of things that everyone can eat. It does not make sense for an instance of food to exist.

- An abstract class is just like a normal class - except that an object of that class cannot be created.
- Other classes, however, can inherit from an abstract class.

Abstract Classes (2)

- Abstract classes are specifications of what subclasses should contain and do.
- They tell the compiler that a method cannot have a realistic implementation at this point (in other words, the implementation will be carried out at an appropriate lower level in the hierarchy).

Abstract Classes (3)

- For example, the `Number` class in the `java.lang` package represents the abstract concept of numbers.
- It makes sense to model numbers in a program, but it does not make sense to create a generic number object. Instead, the `Number` class makes sense only as a superclass to classes like `Integer` and `Float`, both of which implement specific kinds of numbers and hence can be instantiated.
- An abstract class can contain fields and methods, just like any other ordinary classes.

The abstract keyword

- If an abstract class contains a method declared ‘abstract’, the subclasses must then implement these methods themselves if they are to be concrete classes.
- To make a class abstract, the keyword ‘abstract’ is placed before
 - the keyword ‘class’ in the class definition,
 - and the keyword ‘method’ in the method definition respectively.

```
abstract class Number {  
    . . .  
}
```

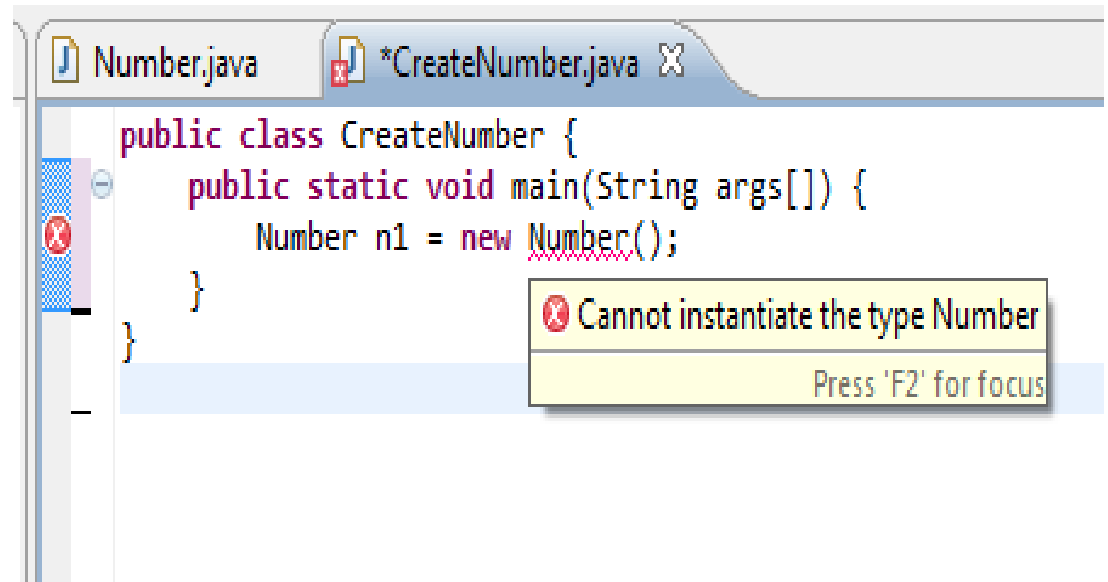
Abstract classes and objects

- An abstract class cannot be instantiated as an object. If the following is typed in Eclipse:

```
abstract class Number { /*...*/ }
```

```
Number a1 = new Number(); /*** ERROR
```

- An error is displayed:



Abstract classes and methods

- Abstract classes can contain an arbitrary mix of method declarations and method definitions.
- There are three ways for a class to be abstract:
 - The class is explicitly declared abstract.
 - An encapsulated method is explicitly declared abstract.
 - The class fails to define all of the methods in the interfaces that the class implements.
- If a class has one or more methods declared abstract, then the entire class must also be declared with the abstract keyword.

Abstract Methods

- An abstract method is a method with no method statements – no implementation.

public abstract class Shape{

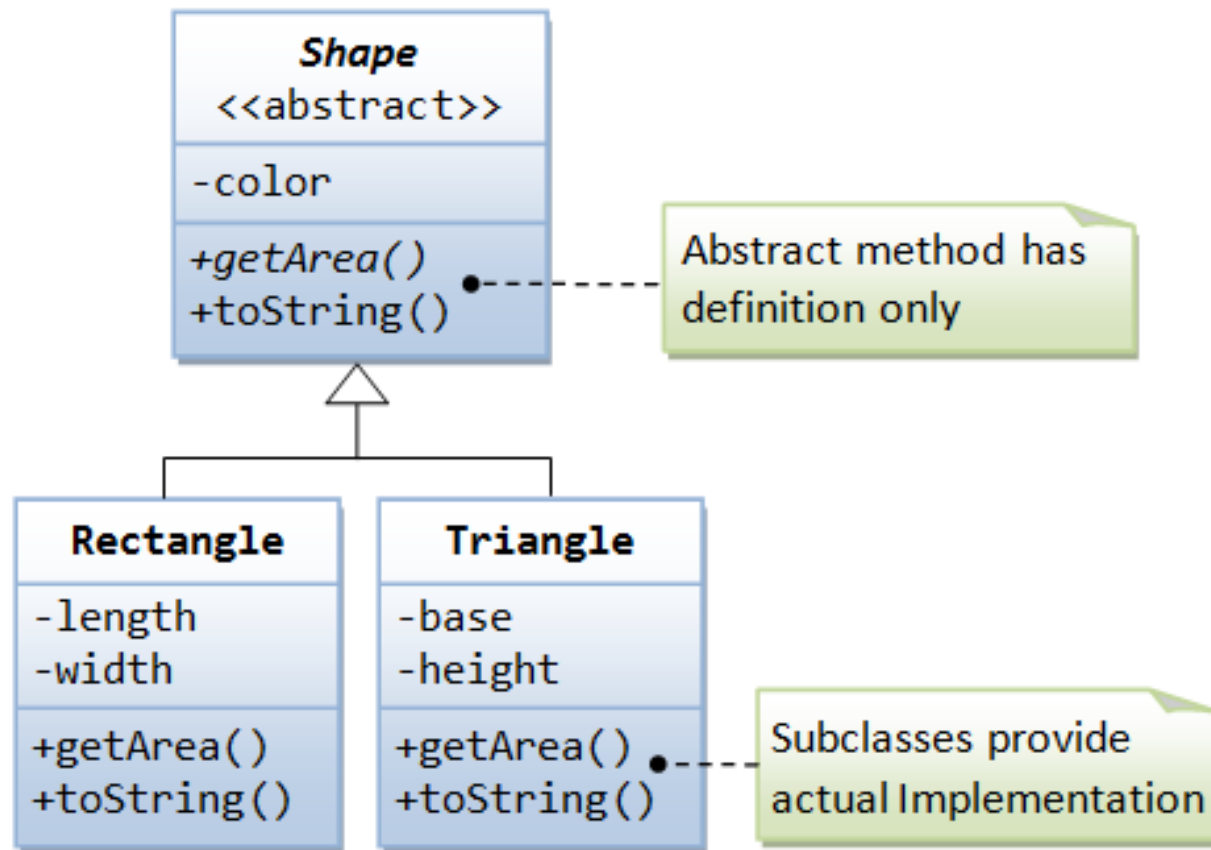
...

//abstract method – no method definition

abstract public double getArea();

}

Abstract Classes (2)



Abstract classes and polymorphism

- Abstract methods are usually declared where two or more subclasses are expected to fulfill a similar role in different ways through different implementations.
- Another class (Concrete class) has to provide implementation of abstract methods.
 - Concrete class has to implement all abstract methods of the abstract class in order to be used for instantiation
 - Concrete class uses extends keyword
- These subclasses extend the same Abstract class and provide different implementations for the abstract methods - polymorphism

Example 1: Book.java

```
public abstract class Book  
{  
String title;  
double price;  
public Book(String t)  
{  
title = t;  
}
```

```
public String getTitle(){  
return title;  
}  
public double getPrice(){  
return price;  
}  
public abstract void setPrice();  
}
```

Example 1: Fiction.java

```
public class Fiction extends Book  
{  
public Fiction(String title){  
super(title); //calling superclass constructor to initialise title  
setPrice();  
}  
public void setPrice(){  
super.price=24.99; //setting the value of price  
}  
}
```

Example 1: NonFiction.java

```
public class NonFiction extends Book  
{  
    public NonFiction(String title){  
        super(title); //calling constructor of superclass  
        setPrice();  
    }  
    public void setPrice(){  
        super.price=37.99; //setting the value of price  
    }  
}
```

Example 1: CreateBook.java

```
public class UseBook{  
    public static void main(String[] args){  
        Fiction aNovel = new Fiction("Huckelberry Finn");  
        NonFiction aManual = new NonFiction("Elements of Style");  
        System.out.println("Fiction " + aNovel.getTitle() + " costs $"  
        + aNovel.getPrice());  
        System.out.println("Non-Fiction " + aManual.getTitle() + "  
        costs $" + aManual.getPrice());  
    }  
}
```

Notes on Example 1

- The abstract class Book has a constructor
- The subclass constructors Fiction and NonFiction call the constructor of Book
- In UseBook, the methods getTitle and getPrice are used with objects of Fiction and NonFiction – these 2 classes have inherited these methods.

Example 2: Shape.java

```
abstract class Shape { // abstract class  
abstract double area(); // abstract method  
}
```

Example 2: Rectangle.java

```
public class Rectangle extends Shape {  
    double width, height;  
    Rectangle(double aWidth, double aHeight) {  
        width=aWidth;  
        height=aHeight;  
    }  
    double area() {  
        return width * height;  
    }  
}
```

Example 2: Circle.java

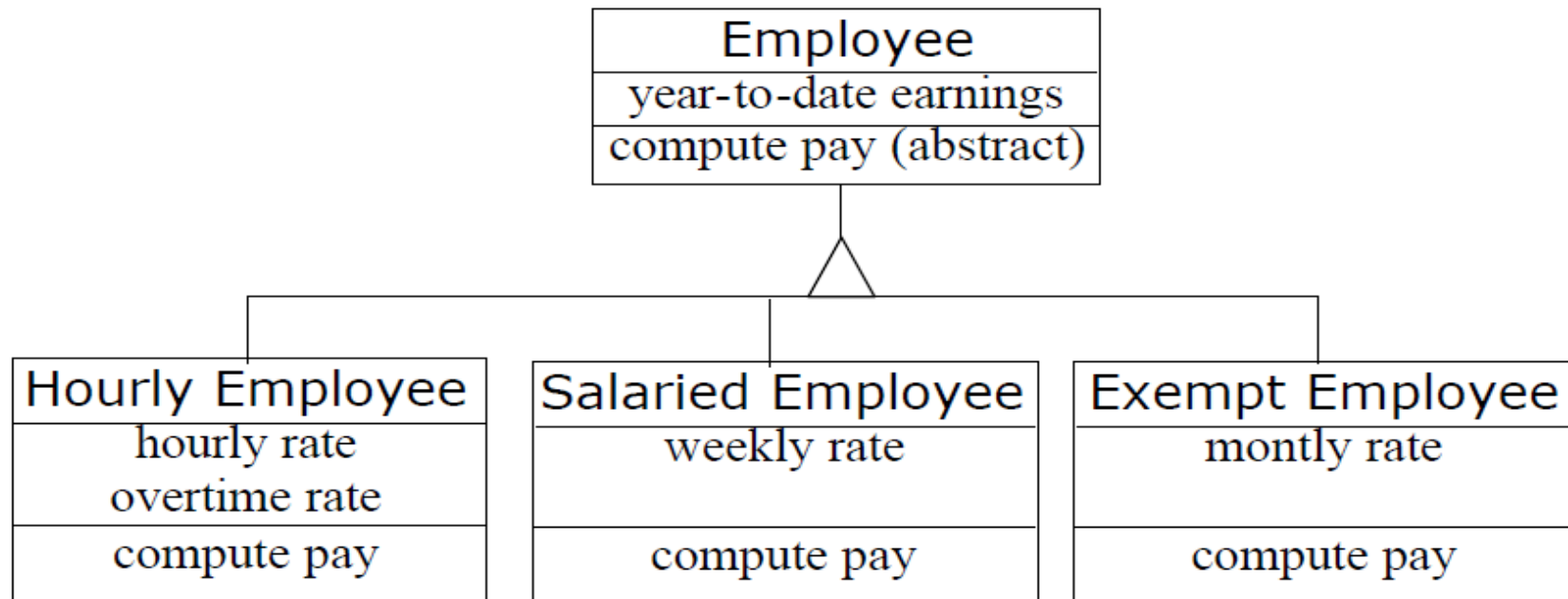
```
class Circle extends Shape {  
    double radius;  
    Circle(double aRadius) {  
        radius = aRadius;  
    }  
    double area() {  
        return Math.PI * radius * radius;  
    }  
}
```

ShapeTest.java

```
public class ShapeTest {  
    public static void main (String[]  
    arg) {  
        Shape[] shapes = new Shape[3];  
        //array of shape  
        shapes[0] = new Circle(100.0);  
        shapes[1] = new  
        Rectangle(150.0, 200.0);  
        shapes[2] = new  
        Rectangle(250.0, 300.0);
```

```
        double total_area = 0;  
        for (int i = 0; i < shapes.length;  
        i++)  
            total_area += shapes[i].area();  
        System.out.println("The total area  
        for 3 shapes is =  
        " + total_area);  
    }  
}
```

Example 3: Abstract class Employee



Notes

- A class can be declared abstract, even though none of its methods are declared abstract.
- A constructor cannot be abstract

Interfaces

- They define a standard and public way of specifying the behavior of classes
- All methods of an interface are abstract methods
 - Defines the signatures of a set of methods, without the body (implementation of the methods)
- A concrete class must implement the interface (all the abstract methods of the Interface)
- They allows classes, regardless of their locations in the class hierarchy, to implement common behaviors

Interfaces

- ▶ An interface is a *contract* (or a protocol, or a common understanding) of what the classes can do.
- ▶ When a class implements a certain interface, it promises to provide implementation to all the abstract methods declared in the interface.
- ▶ Interface defines a set of common behaviors. The classes implement the interface agree to these behaviors and provide their own implementation to the behaviors.

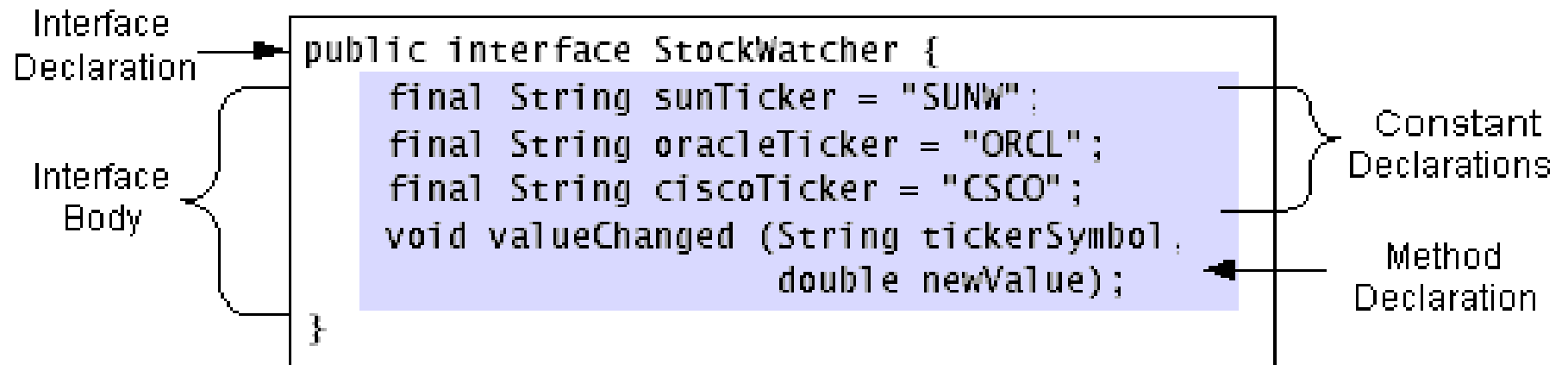


Interfaces in Java

- Java uses the implements keyword to indicate that a class implements one or more interfaces. A class must conform with any interface it implements. This means every method specified in the interface must be implemented.
- An interface is defined with the keyword interface:

```
interface MyInterface {  
    public void m1(); //no implementation provided  
    public String m2(); //no implementation provided  
}
```

Interfaces in Java



Interfaces

- An interface cannot be instantiated as an object.
- An interface can contain
 - method declarations rather than definitions
 - static and final fields, which represent constants
e.g: `public static final double pi = 3.1415;`
- All interface members must be declared public.

Example 4: Relation Interface

*// Note that Interface contains just set of method
// signatures without any implementations.
// No need to say abstract modifier for each method
// since it assumed.*

```
public interface Relation {  
    public boolean isGreater( Object a, Object b);  
    public boolean isLess( Object a, Object b);  
    public boolean isEqual( Object a, Object b);  
}
```

Example 5: OperateCar Interface

```
public interface OperateCar {  
    // constant declarations, if any and method signatures  
    int turn(Direction direction, double radius, double startSpeed, double  
        endSpeed);  
    int changeLanes(Direction direction, double startSpeed,  
        double endSpeed);  
    int signalTurn(Direction direction, boolean signalOn);  
    int getRadarFront(double distanceToCar, double speedOfCar);  
    int getRadarRear(double distanceToCar, double speedOfCar);  
}
```

Classes and Interfaces

- A class implements an interface:

```
class C implements Runnable {  
    //implementation of all abstract methods  
}
```

- A class can implement multiple interfaces. The interface identifiers are separated by commas.

```
public class ComputerScienceStudent implements  
    PersonInterface, AnotherInterface, Thirdinterface{  
    // Implementation of all abstract methods of all interfaces.  
}
```

Classes and Interfaces (2)

- By convention, the `implements` clause follows the `extends` clause if it exists.

public class ComputerScienceStudent


extends Student

implements PersonInterface, AnotherInterface,
Thirdinterface{

*// Implementation of all abstract methods of **all** interfaces.*

}

Example 6 – interface Flyer and class Airplane

```
interface Flyer {  
    public void takeOff();  
    public void land();  
    public void fly();  
}  
  
class Airplane implements Flyer  
{  
    public void takeOff() {   
        System.out.println("Hang on...  
airplane is ready to take  
off..");  
    }  
}
```

```
    public void land() {  
        System.out.println("Hold on...  
airplane is ready to land..");  
    }  
  
    public void fly() {  
        System.out.println("Relax... we  
are flying ...");  
    }  
}
```


Example 6 – class Bird

```
class Bird implements Flyer {  
    public void takeOff() {  
        System.out.println("Wings ready... Tweety to take off...");  
    }  
    public void land() {  
        System.out.println("Tweety is ready to land ...");  
    }  
    public void fly() {  
        System.out.println("Tweety is flying ...");  
    }  
}
```

Example 7

```
public interface Product  
{  
    static final String MAKER = "My Corp";  
    public int getPrice(int id);  
}
```

Example 7 – class Shoe

// Class Shoe which implements the Product Interface

public class Shoe implements Product{

public int getPrice(int id){

if (id == 1)

return(5);

else

return(10);

}

public String getMaker(){

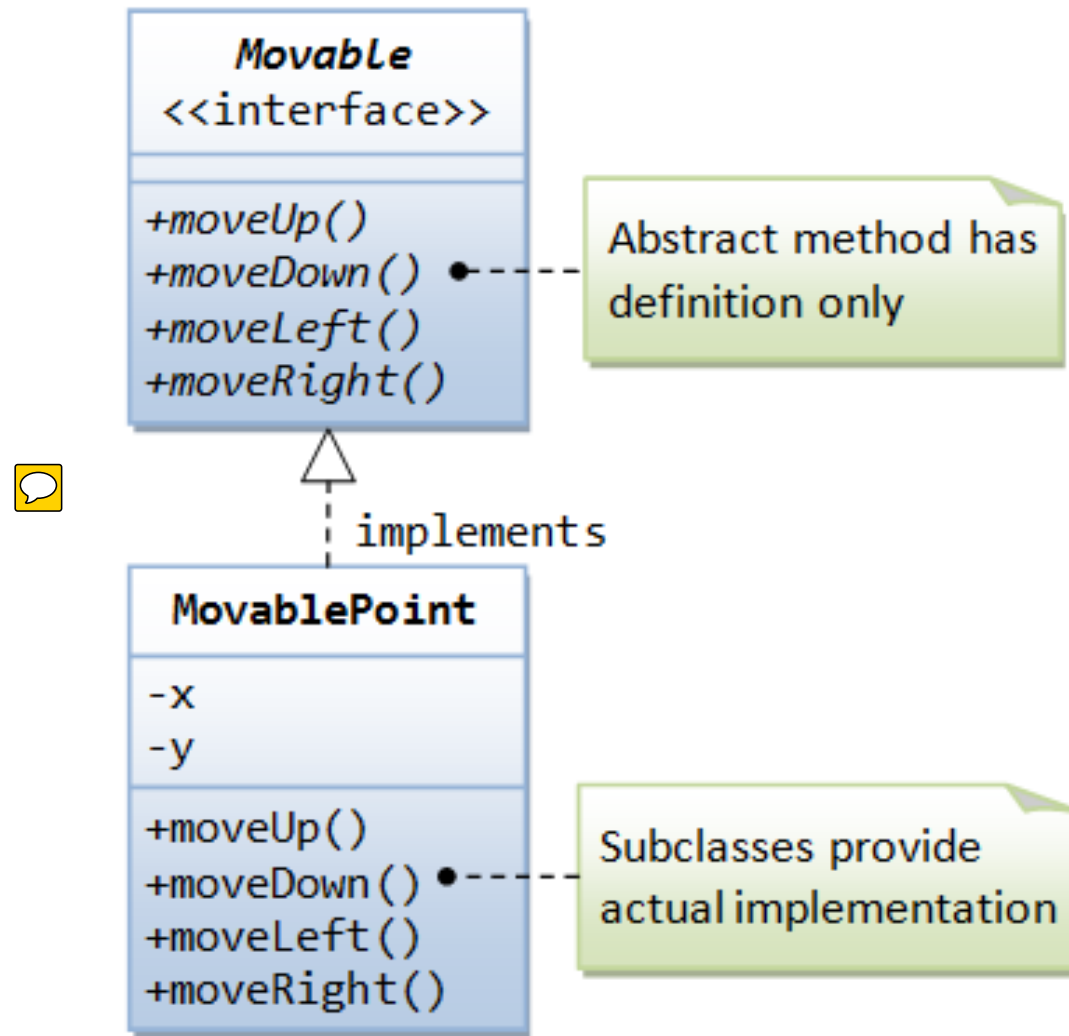
return(MAKER);

}

}



Example 8



Example 8

Interface Moveable.java

```
public interface Movable {  
    // abstract methods to be implemented by the subclasses  
    public void moveUp();  
    public void moveDown();  
    public void moveLeft();  
    public void moveRight();  
}
```



Example 8

```
public class MovablePoint implements
Movable {
    // Private member variables
    private int x, y; // (x, y) coordinates of
the point

    // Constructor
    public MovablePoint(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public String toString() {
        return "Point at (" + x + "," + y + ")";
    }
}
```

// Implement abstract methods defined
in the interface Movable

@Override

```
public void moveUp() {
    y--;
}
```



@Override

```
public void moveDown() {
    y++;
}
```

@Override

```
public void moveLeft() {
    x--;
}
```

@Override

```
public void moveRight() {
    x++;
}
}
```

Example 8

We can also upcast subclass instances to the Movable interface, via polymorphism, similar to an abstract class.

```
public class TestMovable {  
    public static void main(String[] args) {  
        Movable m1 = new MovablePoint(5, 5); // upcast  
        System.out.println(m1);  
        m1.moveDown();  
        System.out.println(m1);  
        m1.moveRight();  
        System.out.println(m1);  
    }  
}
```



Interfaces (4)

- Interfaces are not part of the class hierarchy, however they may have relationships amongst themselves.
- One interface can extend (subclass) one or more interfaces.

```
public interface PersonInterface {  
    void doSomething();  
}
```

```
public interface StudentInterface extends PersonInterface {  
    void doExtraSomething();  
}
```


Abstract Classes v/s Interfaces

1. An Abstract class is a class which contains one or more abstract methods. An abstract class can contain no abstract methods also i.e. abstract class may contain concrete methods. A Java Interface can contain only method declarations and public static final constants.
2. Abstract class definition begins with the keyword “abstract” keyword followed by Class definition. An Interface definition begins with the keyword “interface”.
3. Abstract classes are useful in a situation when some general methods should be implemented and specialization behavior should be implemented by subclasses. Interfaces are useful in a situation when all its properties need to be implemented by subclasses

Abstract Classes v/s Interfaces (2)

4. All variables in an Interface are by default - public static final while an abstract class can have instance variables.
5. An interface is used in situations when a class needs to extend another class apart from the abstract class. In such situations it is not possible to have multiple inheritance of classes. An interface on the other hand can be used when it is required to implement one or more interfaces. Abstract class does not support Multiple Inheritance whereas an Interface supports multiple Inheritance.
6. An Interface can only have public members whereas an abstract class can contain private as well as protected members.

Abstract Classes v/s Interfaces (3)

7. A class implementing an interface must implement all of the methods defined in the interface, while a class extending an abstract class need not implement any of the methods defined in the abstract class if implemented already.
8. In an interface a new feature (method) is to be added, then those methods **MUST** be implemented in all of the classes which implement that interface. However, in the case of an abstract class, the method can be simply implemented in the abstract class and the same can be called by its subclass.

Abstract Classes v/s Interfaces (4)

9. Interfaces are often used to describe the peripheral abilities of a class, and not its central identity, E.g. an Automobile class might implement the Recyclable interface, which could apply to many otherwise totally unrelated objects.

