JAVA Basic Concept

TUTORIAL 2

DBMS 2013-2014 Fall

TAs:Nagehan İlhan

Mahiye Uluyağmur

Inheritance

- In the Java language, classes can be *derived* from other classes, thereby *inheriting* fields and methods from those classes.
- A class that is derived from another class is called a *subclass* (child class).
- The class from which the subclass is derived is called a *superclass* (*base class* or *parent class*).

Inheritance

- Java Inheritance defines an is-a relationship between a superclass and its subclasses.
- This means that an object of a subclass can be used wherever an object of the superclass can be used.
- Class **Inheritance in java** mechanism is used to build new classes from existing classes.
- The inheritance relationship is transitive: if class x extends class y, then a class z, which extends class x, will also inherit from class y.

Inheritance

- Excepting Object, which has no superclass, every class has one and only one direct superclass (single inheritance).
- In the absence of any other explicit superclass, every class is implicitly a subclass of Object.
- Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, Object.

Inheritance

- The idea of inheritance is simple but powerful:

When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class.

Example

```
class Box {

        double width;
        double height;
        double depth;
        Box() { }
        Box(double w, double h, double d) {
                width = w;
                height = h;
                depth = d;
        }
        void getVolume() {
                System.out.println("Volume is : " + width * height * depth);
        }
}
```

Example-Cont’

```
public class MatchBox extends Box {
        double weight;
        MatchBox() {}
        MatchBox(double w, double h, double d, double m)
        {
          super(w, h, d);
          weight = m;
        }
        public static void main(String args[]) {
                MatchBox mb1 = new MatchBox(10, 10, 10, 10);
                mb1.getVolume();
                System.out.println("width of MatchBox 1 is " + mb1.width);
                System.out.println("height of MatchBox 1 is " + mb1.height);
                System.out.println("depth of MatchBox 1 is " + mb1.depth);
                System.out.println("weight of MatchBox 1 is " + mb1.weight);     }
}
```

Output

```
Output
Volume is : 1000.0
width of MatchBox 1 is 10.0
height of MatchBox 1 is 10.0
depth of MatchBox 1 is 10.0
weight of MatchBox 1 is 10.0
```

Inheritance

- *What is not possible using java class Inheritance?*

  Answer:

  1. Private members of the superclass are not inherited by the subclass and can only be indirectly accessed.

  2. Members that have default accessibility in the superclass are also not inherited by subclasses in other packages, as these members are only accessible by their simple names in subclasses within the same package as the superclass.

  3. Since constructors and initializer blocks are not members of a class, they are not inherited by a subclass.

  4. A subclass can extend only one superclass

Example

```
class Vehicle {
// Instance fields
int noOfTyres;     // no of tyres
private boolean accessories;   // check if accessorees present or not
protected String brand;  // Brand of the car

// Static fields
private static int counter;  // No of Vehicle objects created

// Constructor
  Vehicle() {
      System.out.println("Constructor of the Super class called");
      noOfTyres = 5;
      accessories = true;
      brand = "X";
      counter++;                  }
// Instance methods
      public void switchOn() { accessories = true;     }
      public void switchOff() {accessories = false;     }
      public boolean isPresent() {return accessories;                    }
      private void getBrand() {System.out.println("Vehicle Brand: " + brand);}
// Static methods
      public static void getNoOfVehicles() { System.out.println("Number of Vehicles: " + counter);       }
}
```

Example-Cont’

```
class Car extends Vehicle {
		private int carNo = 10;
		public void printCarInfo() {
		System.out.println("Car number: " + carNo);
		System.out.println("No of Tyres: " + noOfTyres); // Inherited.
		//  System.out.println("accessories: "    + accessories); // Not Inherited.
		System.out.println("accessories: " + isPresent()); // Inherited.
		//       System.out.println("Brand: "     + getBrand());  // Not Inherited.
		System.out.println("Brand: " + brand); // Inherited.
		//  System.out.println("Counter: "    + counter);     // Not Inherited.
		getNoOfVehicles(); // Inherited.
		}
}
public class VehicleDetails { // (3)

		public static void main(String[] args) {
				new Car().printCarInfo();
		}
}
```

Output

```
Output:
Constructor of the Super class called
Car number: 10
No of Tyres: 5
accessories: true
Brand: X
Number of Vehicles: 1
```

this and super keywords

- Using this and super you have full control on whether to call a method or field present in the same class or to call from the immediate superclass.
- **this** keyword is used as a reference to the current object which is an instance of the current class.
- The keyword **super** also references the current object, but as an instance of the current class's super class.

Example

```
class Counter {
        int i = 0;
        Counter increment() {i++;    return this; }
        void print() {System.out.println("i = " + i); }
        }

public class CounterDemo extends Counter {
        public static void main(String[] args) {
        Counter x = new Counter();
        x.increment().increment().increment().print();}
        }
```

Output:

```
i=3
```

Polymorphism

- The word "*polymorphism*" means "*many forms*".
- Polymorphism is the ability of an object to take on many forms.
- The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Polymorphism

```
class A {
   void whichClass() {
     System.out.println(“A’s method");
  }
}

class B extends A {
                // override kimGeldi()
   void whichClass() {
     System.out.println(“B’s method");
  }  }
}

class C extends A {
                // override kimGeldi()
   void whichClass() {
     System.out.println(“C’s method");
  }  }
}
```

```
class Dispatch {
  public static void main(String args[]) {
    A a = new A();
    B b = new B();
    C c = new C();
    A r;    //( pointer)

    r = a;
    r. whichClass()

    r = b;
    r.whichClass()

    r = c;
    r.whichClass()
  }
}
```

Overload vs. Override

```
class A {
  int i, j;

  A(int a, int b) {
    i = a;
    j = b;
  }

void show() {
System.out.println("i and j: " + i + " " + j);
  }
}
class B extends A {
  int k;
   B(int a, int b, int c) {
    super(a, b);
    k = c;
  }
// overload show()
  void show(String msg) {
    System.out.println(msg + k);
  }
}

class Override {
public static void main(String args[]) {
 B subOb = new B(1, 2, 3);
  // this calls show() in B
  subOb.show("This is k: ");
// A ‘daki show() metodunu çağırır
  subOb.show();
  }
}
```

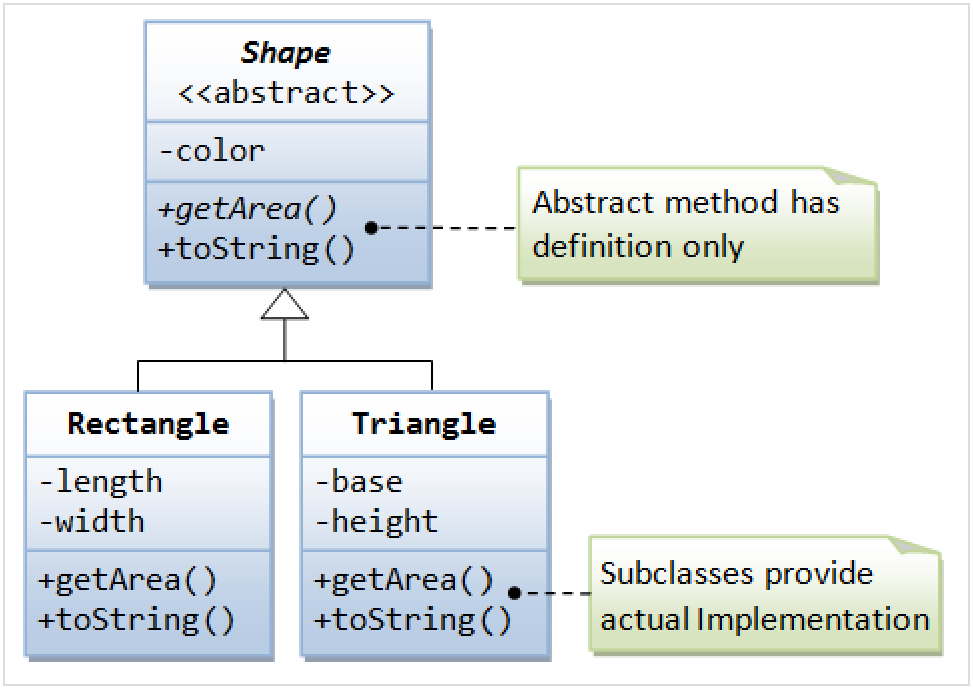
Abstract Classes&Methods

- **The abstract Method**: An abstract method is a method with only signature (i.e., the method name, the list of arguments and the return type) without implementation (i.e., the method's body).
- You use the keyword abstract to declare an abstract method.

```
abstract public class Shape {
......
public abstract double getArea();
public abstract void draw();
 }
```

Abstract Classes&Methods

- A class containing one or more abstract methods is called an abstract class.
- An abstract class must be declared with a class-modifier abstract.



Abstract Classes&Methods

Shape.java

```
abstract public class Shape {
  // Private member variable
  private String color;

  // Constructor
  public Shape (String color) {
    this.color = color;
  }

  @Override
  public String toString() {
    return "Shape of color=\"" + color + "\"";
  }
  // All Shape subclasses must implement a method called getArea()
  abstract public double getArea();
}
```

Abstract Classes&Methods

```
public class TestShape {
   public static void main(String[] args) {
      Shape s1 = new Rectangle("red", 4, 5);
      System.out.println(s1);
      System.out.println("Area is " + s1.getArea());

      Shape s2 = new Triangle("blue", 4, 5);
      System.out.println(s2);
      System.out.println("Area is " + s2.getArea());

      // Cannot create instance of an abstract class
      Shape s3 = new Shape("green");   // Compilation Error!!
   }
}
```

Abstract Classes&Methods

- An abstract class is *incomplete* in its definition, since the implementation of its abstract methods is missing.
- Therefore, an abstract class *cannot be instantiated.*
- To use an abstract class, you have to derive a subclass from the abstract class.
- In the derived subclass, you have to override the abstract methods and provide implementation to all the abstract methods.
- The subclass derived is now complete, and can be instantiated.
- (If a subclass does not provide implementation to all the abstract methods of the superclass, the subclass remains abstract

Interface

- An interface is a collection of abstract methods.
- A class implements an interface, thereby inheriting the abstract methods of the interface.
- An interface is not a class.
- Writing an interface is similar to writing a class, but they are two different concepts.

Interface

- A class describes the attributes and behaviors of an object.
- An interface contains behaviors that a class implements.
- Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

Interface

- An interface is similar to a class in the following ways:
  - ❑ An interface can contain any number of methods.
  - ❑ An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.

Interface

- An interface is different from a class in several ways:
  - ❑You cannot instantiate an interface.
  - ❑An interface does not contain any constructors.
  - ❑All of the methods in an interface are abstract.
  - ❑An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
  - ❑An interface is not extended by a class; it is implemented by a class. An interface can extend multiple interfaces.

Interface

- The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface:

```
/* File name : NameOfInterface.java */
import java.lang.*;
//Any number of import statements

public interface NameOfInterface
{
   //Any number of final, static fields
   //Any number of abstract method declarations\
}
```

Interface

- Interfaces have the following properties:
  - ❑An interface is implicitly abstract. You do not need to use the **abstract** keyword when declaring an interface.
  - ❑Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
  - ❑Methods in an interface are implicitly public.
- Example:

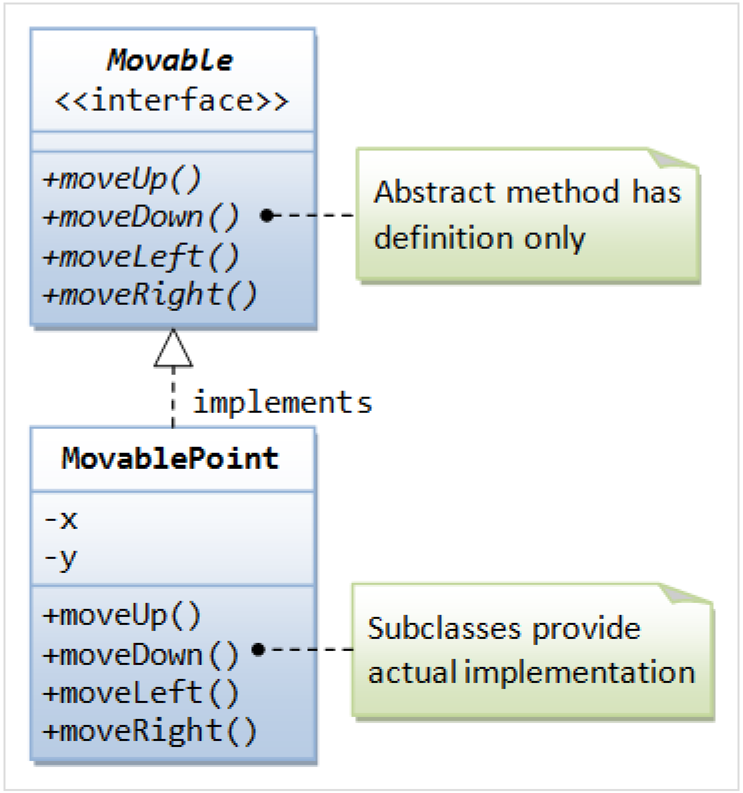
```
/* File name : Animal.java */
interface Animal {
   public void eat();
   public void travel();
}
```

Implementing Interfaces

- When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface.
- If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.
- A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

Implementing Interfaces

Movable Interface and its Implementation



Implementing Interfaces

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();
}
```

To use an interface, again, you must derive subclasses and provide implementation to all the abstract methods declared in the interface. The subclasses are now complete and can be instantiated.

Implementing Interfaces

MovablePoint.java

```
public class MovablePoint implements
Movable {
  // Private membet 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++;
  }
}
```

Implementing Interfaces

TestMovable.java

```
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);
  }
}
```

Implementing Multiple interfaces

- Java does not support *multiple inheritance* to avoid inheriting conflicting properties from multiple superclasses.
- A subclass, however, can implement more than one interfaces.

```
public class Circle extends Shape implements Movable, Displayable
 {  // One superclass but implement multiple interfaces
    .......
}
```