

Module 1-12

Interfaces

static variables - review

Static members belong to the class. ***Instance members belong to an instance of the class.*** Static methods can be invoked without creating an instance of the class. Static variables and methods cannot be accessed with the **this** keyword, since they are not part of the object.

```
public class Mortgage {  
    private static double interestRate = 5.5;  
  
    public static void setInterestRate(double interestRate) {  
        Mortgage.interestRate = interestRate;  
    } //this.interestRate
```

Since static variables are shared by all instances of a *class* and not the individual *object*. Changes to the value from one object can be seen from all objects of that type. (note: static → only one copy)

static methods

If we define a method static, our class does not need to be instantiated to use it. Instead it can be accessed from the Class itself, instead of the object. *Static methods can only access other static methods or variables.*

```
public class Rectangle() {  
  
    private static int length;  
    private static int width;  
  
    public static int getArea() {  
        return length * width;  
    }  
}
```

```
private static void main(String[] args)
```

In the class using the static method:

```
Rectangle.getArea();
```

```
Rectangle rect = new Rectangle();  
rect.getArea();
```

Examples

```
Math.abs()  
Math.random()
```

```
String.join()  
String.valueOf()
```

```
Double.parseDouble()  
Integer.parseInt()
```

Lecture note : BigDecimal

- The BigDecimal class **provides operations on double numbers for arithmetic, scale handling, rounding, comparison, format conversion and hashing.**
- It can handle very large and very small floating point numbers with great precision but compensating with the time complexity a bit.
- `import java.math.BigDecimal;`

Lecture note : BigDecimal

Input : double a=0.03;

double b=0.04;

double c=b-a;

System.out.println(c);

Output :0.0099999999999999998

Input : BigDecimal _a = new BigDecimal("0.03");

BigDecimal _b = new BigDecimal("0.04");

BigDecimal _c = _b.subtract(_a);

System.out.println(_c);

Output :0.01

Objectives

- Explain what polymorphism is and how it is used with inheritance and interfaces
- Demonstrate an understanding of where inheritance can assist in writing polymorphic code
- State the purpose of interfaces and how they are used
- Implement polymorphism through inheritance
- Implement polymorphism through interfaces

Inheritance - review

Inheritance refers to a feature of **Java programming** that lets you create classes that are derived from other classes. A class that's based on another class inherits the other class. The class that is **inherited** is the parent class, the base class, or the superclass.

Inheritance - review

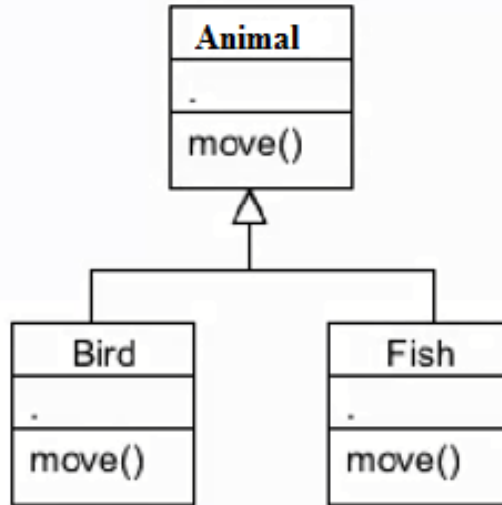
Inheritance refers to a feature of **Java programming** that lets you create classes that are derived from other classes. A class that's based on another class inherits the other class. The class that is **inherited** is the parent class, the base class, or the superclass.

Polymorphism - Overview

- **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. Any **Java** object that can pass more than one IS-A test is considered to be polymorphic. ...

Polymorphism

- Example : Animal Class



Animal Class

```
public class Animal {  
    public void move()  
    {  
        System.out.println("Moving");  
    }  
}
```

Bird Class

```
public class Bird extends Animal {  
    @Override  
    public void move()  
    {  
        System.out.println("Flying");  
    }  
}
```

Fish Class

```
public class Fish extends Animal {  
    @Override  
    public void move()  
    {  
        System.out.println("Swimming");  
    }  
}
```

AnimalApp Class

```
public class AnimalApp {  
    public static void main(String[] args) {
```

```
        Animal a = new Bird();  
        a.move();
```

```
        Animal b = new Fish();  
        b.move();  
    }
```

```
}
```

Output:

- Output:

Flying

Swimming

Notes in Inheritance

- `super()`
 - It is used by class constructors to invoke constructors of its parent class.
 - It is used inside a sub-class method definition to call a method defined in the super class. Private methods of the super-class cannot be called. Only public and protected methods can be called by the super keyword.

Principles of Object-Oriented Programming (OOP)

- **Encapsulation** - the concept of hiding values or state of data within a class, limiting the points of access
- **Inheritance** - the practice of creating a hierarchy for classes in which descendants obtain the attributes and behaviors from their parent classes
- **Polymorphism** - the ability for our code to take different forms
- **(Abstraction)** – extension of encapsulation. We don't build a car from scratch, but we know how to use (drive) it.



Polymorphism!

Three Main Inheritance Scenarios

Recall from the previous discussion that there are three main ways inheritance can be implemented.

- A **concrete class** (all the classes we have seen so far) inheriting from another concrete class.
- A concrete class inheriting from an **abstract class**.
- A concrete class inheriting from an **Interface**. ←

Today we will be working with Java interfaces.



Java Interfaces

An interface is best thought of as a contract between the interface itself and a particular class.

Consider the following real world examples:

- A fast food restaurant franchise might stipulate that the franchisee must place a giant logo in the front of the building.
 - *The franchisee is free to choose whatever contractors or workers it needs to actually mount the logo.*

Java Interfaces

In layman's term/another way to explain it:

1. An interface can be considered as an interest group or even a country club.

e.g. class FarmAnimal implements Singable {..}

→ class FarmAnimal joins the Singable club and becomes its member.

2. If we have,

```
public class FarmAnimal implements Singable {..}
```

```
public class Chicken extends FarmAnimal{..}
```

```
public class Pig extends FarmAnimal{..}
```

```
public class Cow extends FarmAnimal{..}
```

```
public class Tractor implements Singable{..}
```

Java Interfaces

```
Singable[] singables = new Singable[] {new Cow(), new Chicken(), new Pig(), new Tractor()};
```

```
for (Singable singable : singables) {  
    String name = singable.getName();  
    String sound = singable.getSound();  
    System.out.println("Old MacDonald had a farm, ee, ay, ee, ay, oh!");  
    System.out.println("And on his farm he had a " + name  
        + ", ee, ay, ee, ay, oh!");  
    System.out.println("With a " + sound + " " + sound + " here");  
    System.out.println("And a " + sound + " " + sound + " there");  
    System.out.println("Here a " + sound + " there a " + sound  
        + " everywhere a " + sound + " " + sound);  
    System.out.println();  
}
```

Java Interfaces

A class that chooses to implement an interface will define whatever method the interface asks it to implement.

- The methods that the child class needs to implement are defined in the Interface through **abstract methods**.
- An interface itself is an example of a class that is “abstract in nature” though there is actually something called an **Abstract Class** (this will be the subject of tomorrow’s lecture)..
 - Therefore, an interface cannot be instantiated, and can only be “implemented” by some other class.

Java Interfaces: Declaration

- The declaration for an Interface is as follows:

```
public interface <<Name of the Interface>> {...}
```



- A class implementing an Interface must have the following convention:

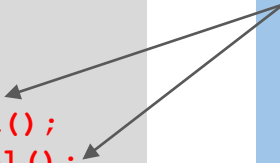
```
public class <<Name of (Child) Class>> implements <<Name of Interface>> {...}
```

- The class implementing an interface is also referred to as the **concrete class**.
- You cannot instantiate Interfaces, you can only instantiate the classes that implement an interface.

Java Interfaces: Abstract Methods

An abstract method is one that doesn't have an implementation, that is to say it has no body. Here is an example from a Vehicle Interface:

```
package te.mobility;  
  
public interface Vehicle {  
  
    public void honkHorn();  
    public void checkFuel();  
  
}
```



- The Interface Vehicle has two abstract methods: **honkHorn()** and **checkFuel()**.
- Note that these abstract methods do not have a body, there is no {...}, and it **ends with a semicolon**.

Java Interfaces: Abstract Methods

A class that implements Vehicle must provide a concrete implementation of the two abstract methods.

```
package te.mobility;  
  
public interface Vehicle {  
  
    public void honkHorn();  
    double checkFuel();  
}
```

honkHorn has
been
implemented

checkFuel has
been
implemented

```
package te.mobility;  
  
public class Car implements Vehicle {  
  
    private double fuelLeft;  
    private double tankCapacity;  
  
    @Override  
    public void honkHorn() {  
        System.out.println("beeeep");  
    }  
  
    @Override  
    public double checkFuel() {  
        return (fuelLeft / tankCapacity) * 100;  
    }  
}
```

Java Interfaces: Abstract Method Rules

When implementing abstract methods on a concrete class, the following rules are observed:

- To fulfill the Interface's contract, the concrete class must implement the method with the exact same return type, exact same name, and exact same number of arguments (with correct data types).
- The access modifier on the implementation cannot be more restrictive than that of that parent Interface.
 - For example - the concrete class cannot implement the method as private if the abstract class has marked it as public.
- All abstract methods are assumed to be public.



Java Interfaces: Default Methods

Looking at our Vehicle interface, we could define the default method as follows:

```
package te.mobility;

public interface Vehicle {

    double checkFuel(String units);

    default void honkHorn() {
        System.out.println("beep");
    }
}
```

An instance of a concrete class that implements Vehicle can just call honkHorn now through the instantiated object, i.e. myCar.honkHorn();

Java Interfaces: Default Methods

A concrete class can override the default method by implementing its own version of the method:

```
package te.mobility;

public interface Vehicle {

    public double checkFuel(String units);

    default void honkHorn() {
        System.out.println("interface");
    }
}
```

For an instance of car, if honkHorn is invoked, this one takes priority. The output will be "concrete."

```
package te.mobility;

public class Car implements Vehicle {

    private double fuelLeft;
    private double tankCapacity;

    public void honkHorn() {
        System.out.println("concrete");
    }

    @Override
    public double checkFuel(String units) {
        return (fuelLeft / tankCapacity) * 100;
    }
}
```

Java Interfaces: Data Members

It is possible for interfaces to have data members, if they do, **they are assumed to public, static, and final.**



Java Interfaces: Polymorphism References

Interfaces allow us to create references based on the interface, but instantiate an instance of the concrete class instead.

```
Vehicle vehicle = new Car();
```

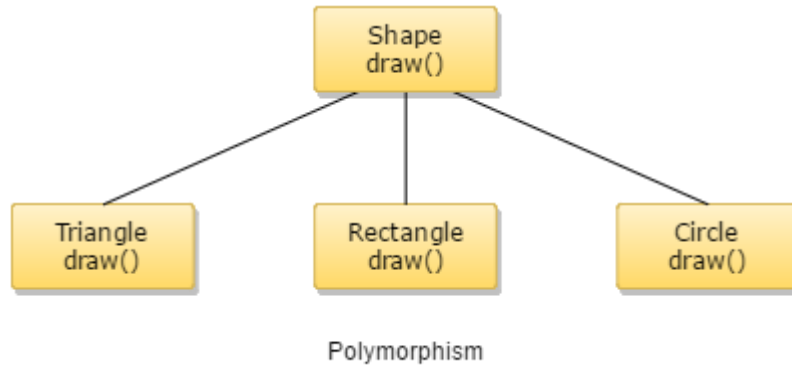
Have we seen this before? Think about Lists, Maps, and Sets.

To the left of the equal sign is the reference, note it is of the interface type.

To the right of the equal sign is the instantiation of the object, note it is of the concrete class.

Objectives

- Should be able to explain what polymorphism is and how it is used with inheritance and interfaces



Objectives

- Should be able to explain what polymorphism is and how it is used with inheritance and interfaces
- Should be able to demonstrate an understanding of where inheritance can assist in writing polymorphic code

Interfaces and Polymorphism

Java interfaces are a way to achieve polymorphism. Polymorphism is a concept that takes some practice and thought to master. Basically, polymorphism means that an instance of a class (an object) can be used as if it were of different types. Here, a type means either a class or an interface.

<http://tutorials.jenkov.com/java/interfaces.html#interfaces-and-polymorphism>

Objectives

- Should be able to explain what polymorphism is and how it is used with inheritance and interfaces
- Should be able to demonstrate an understanding of where inheritance can assist in writing polymorphic code
- Should be able to state the purpose of interfaces and how they are used

A *Java interface* is a bit like a **Java class**, except a Java interface can only contain method signatures and fields. A Java interface is not intended to contain implementations of the methods, only the signature (name, parameters and exceptions) of the method. However, it is possible to provide default implementations of a method in a Java interface, to make the implementation of the interface easier for classes implementing the interface.

<http://tutorials.jenkov.com/java/interfaces.html#interfaces-and-polymorphism>

Objectives

- Should be able to explain what polymorphism is and how it is used with inheritance and interfaces
- Should be able to demonstrate an understanding of where inheritance can assist in writing polymorphic code
- Should be able to state the purpose of interfaces and how they are used
- Should be able to implement polymorphism through inheritance



Objectives

- Should be able to explain what polymorphism is and how it is used with inheritance and interfaces
- Should be able to demonstrate an understanding of where inheritance can assist in writing polymorphic code
- Should be able to state the purpose of interfaces and how they are used
- Should be able to implement polymorphism through inheritance
- Should be able to implement polymorphism through interfaces

