Module 1-13

Abstract Classes

Today’s Objectives

1. Abstract Methods Classes
2. Final Methods and Classes
3. Protected and Default Access Modifiers
4. Static Methods
5. Software Development Lifecycle (if time allows)

Java Interfaces - Review

1. An interface is **a fully abstract class**. It includes a group of abstract methods (methods without a body). We use the interface keyword to create an interface in Java.

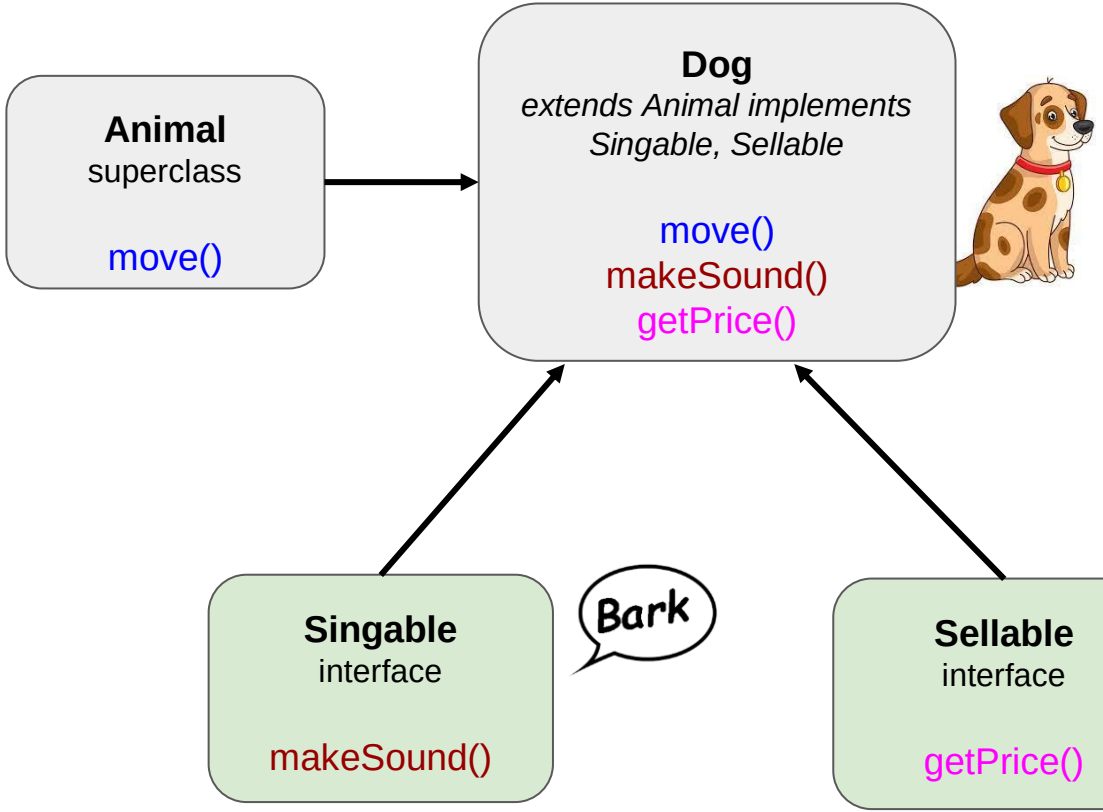
2. An Interface in Java programming language is defined as **an abstract type used to specify the behavior of a class**. An interface in Java is a blueprint of a class...

3. An interface is **a completely "abstract class" that is used to group related methods with empty bodies**:

Object instanceof

Dog turkey = new Dog();

turkey *instanceof* Animal
turkey *instanceof* Dog
turkey *instanceof* Singable
turkey *instanceof* Sellable



An Object is an instanceof a data type if it can be safely cast to that data type.

Abstract Class

An abstract class cannot be instantiated and exists solely for the purpose of inheritance and polymorphism.

Like a combination of an interface and a superclass.

1. Can extend it like a superclass
2. Can inherit implementation from it like a superclass
3. Can provide method signatures that must be implemented like an interface
4. A class can only extend either 1 abstract class or 1 superclass, but may implement multiple interfaces.

Abstract Method

A method signature with the abstract modifier. Can only exists in an abstract class, and must be overridden in any implementation class that extends the abstract class.

```
public abstract int doSciene(int x, int y)
```

Reasons to use an Abstract Class

1. To prevent a superclass from being instantiated.
    a. For example: Having a Generic “Feline” as an object may not make sense, so by making Feline abstract it forces the user of the class to instantiate the more concrete HouseCat or Lion objects that are subclasses of Feline.
2. When you need to have the ability to inherit functionality from a superclass and force a subclass to implement subclass specific methods.
    a. ***Abstract classes should only represent a IS-A relationship*** (a Car IS-A Vehicle)
    b. This same need can also be accomplished by using a combination of a superclass and interfaces.

Final Methods and Classes

The **final** modifier can be used with methods and classes to control how they are used.

A final method cannot be Overridden.

```
public final String myMethod(){ }
```

A final Class cannot have subclasses.

```
public final class myClass { }
```

Superclass vs Interface vs Abstract

An **interface** provides method signatures without implementation that creates a contract of what must have a subclass specific implementation. *Can represent either an HAS-A or IS-A relationship.*

A **superclass** provides default implementation that can be inherited by a subclass, so it can guarantee a default implementation, but not a subclass specific one. *Can only represent an IS-A relationship.*

An **abstract class** allows for default implementation to be inherited and/or to provide method signatures for methods that must have a subclass specific implementation. *Can only represent an IS-A relationship.*

Protected and Default Access Modifiers

- **Private** - accessible on in the class
  - can be applied to methods and member variables
- **Protected** - accessible in the class and in any subclasses in the inheritance tree.
  - can be applied to methods and member variables
  - In Java, Protected it also available to any class in the same package, but this use is discouraged.
- **Default (no access modifier)** - accessible to any class or subclass in the same package.
  - can be applied to methods and member variables
- **Public** - accessible everywhere
  - can be applied to methods, member variables, classes, and interfaces)

Java Accessors Cheatsheet

toString method

```
class Student {
   private int id;
   private String name;

   public Student(int id, String name) {
      this.id = id;
      this.name = name;
   }
}
// Driver class to test the Student class
public class Demo {
   public static void main(String[] args) {
      Student s = new Student(101, "James Bond");
      System.out.println("The student details are: "+s);
   }
}
```

```
Output:
The student details are:
Student@28d93b30
```

toString method - Override

```
class Student {
   private int id;
   private String name;

   public Student(int id, String name) {
      this.id = id;
      this.name = name;
   }
}
// Driver class to test the Student class
public class Demo {
   public static void main(String[] args) {
      Student s = new Student(101, "James Bond");
      System.out.println("The student details are: "+s);
   }
}
```

```
@Override
 public String toString() {
   return id + " " + name;
 }
```

```
Output:
The student details are:
101 James Bond
```

Module 1-13

Second Deck

Abstract Classes

Objectives

- Should be able to define and use `abstract` in the context of a class and a method
- Should be able to define and use `final` in the context of a class and a method
- Should be able to explain the differences between public, private, and protected access

If else if chain vs. switch statement

```
…
System.out.println(“Movie ticket prices: “);
System.out.println(“1. Adult - $14.00”);
System.out.println(“2. Child - $8.00”);
 System.out.println(“3. Senior - $11.00”);
System.out.print(“Enter choice: “);
int choice = Integer.parseInt(input.nextLine());
       if (choice == 1) {
          total = quantity * 14;
       } else if (choice == 2) {
           total = quantity * 8;
       }  else if (choice == 3) {
            total = quantity * 11;
       }  else {
            System.out.println(“Invalid entry”);
       }

  …
```

```
…
System.out.println(“Movie ticket prices: “);
System.out.println(“1. Adult - $14.00”);
System.out.println(“2. Child - $8.00”);
 System.out.println(“3. Senior - $11.00”);
System.out.print(“Enter choice: “);
int choice = Integer.parseInt(input.nextLine());
switch (choice) {
       case 1:
            total = quantity * 14;
            break;
       case 2:
            total = quantity * 8;
            break;
       case 3:
            total = quantity * 11;
            break;
       default:
            System.out.println(“Invalid entry”);
  }

  …
```

Making Animals Sleep

Abstract Classes

Abstract Classes combine some of the features we've seen in interfaces along with inheriting from a concrete class.

- Abstract methods can be extended by concrete classes.
- Abstract classes can have abstract methods
- Abstract classes can have concrete methods
- Abstract classes can have constructors
- Abstract classes, like Interfaces, cannot be instantiated

Abstract Classes : Declaration

We use the following pattern to declare abstract classes.

- The abstract class itself:

  public **abstract** class **<<Name of the Abstract Class>>** {...}

- The child class that inherits from the abstract class:

  public class **<<Name of Child Class>>** **extends** **<<Name of Abstract Class>>**

Abstract Classes Example

```
package te.mobility;

public abstract class Vehicle {

     private int numberOfWheels;
     private double tankCapacity;
     private double fuelLeft;

     public Vehicle(int numberOfWheels) {
        this.numberOfWheels = numberOfWheels;
     }
     public double getTankCapacity() {
        return tankCapacity;
     }

     public abstract Double calculateFuelPercentage();


     public double getFuelLeft() {
       return fuelLeft;
     }
}
```

We need to implement the constructor

We need to implement the abstract method

extends is used.

```
package te.mobility;

public class Car extends Vehicle {

     public Car(int numberOfWheels) {
        super(numberOfWheels);
     }

     @Override
     public Double calculateFuelPercentage() {

        return super.getFuelLeft() /
           super.getTankCapacity() * 100;
     }

}
```

Abstract Classes: final keyword

Declaring methods as **`final`** prevent them from being overriden by a child class.

```
package te.mobility;

public abstract class Vehicle {
...

    public final void refuelCar() {
       this.fuelLeft = tankCapacity;
...
}
```

```
package te.mobility;

public class Car extends Vehicle
{

    @Override
    public void refuelCar() {

    }

}
```

This override will cause an error, as the method is marked as final.

Multiple Inheritance – not allowed

- Java does not allow multiple inheritance of concrete classes or abstract classes. The following **is not allowed**:

  public class Car **extends Vehicle, MotorVehicles** {...}

Where Vehicle and MotorVehicles are classes or abstract classes

- Java does allow for the implementation of multiple interfaces:

  public class Car **implements IVehicle, IMotorVehicle** {...}

Where IVehicle and IMotorVehicle are interfaces

ABSTRACT CLASSES VS INTERFACES

ABSTRACT CLASS

- Defines methods & properties
- Can contain method bodies
- Can contain properties
- Cannot be instantiated
- Are inherited

INTERFACES

- Defines methods & properties
- No method bodies
- No properties
- Cannot be instantiated
- Are implemented