

Module 1 Day 12

Polymorphism: Inheritance & Interfaces

Module 1 Day 12

Can you / Do You?

1. ... explain what polymorphism is and how it is used with inheritance and interfaces
2. ... understand where & how inheritance can help us write polymorphic code
3. ... state the purpose of interfaces and how they are used
4. ... implement polymorphism through inheritance (also see Day 11 Lecture Final Rich)
5. ... implement polymorphism through interfaces

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. (Day 11)
- A concrete class inheriting from an **Interface**. (Today!)
- A concrete class inheriting from an **abstract class**. (Tomorrow)

Today we will be working with Java interfaces.

Java Interfaces

An interface is a contract of behavior that exists between the interface itself and a class that implements the interface.

Interface in the real world:

- **A fast food restaurant chain requires that each franchisee place a giant corporate logo in the front of the building.**
- *While a Logo is required to exist, the franchisee is free to choose the contractors & workers it needs to actually mount the logo.*

Here, the existence of the Corporate Logo is the interface, it is expected when management comes to inspect. The contractors and workers are the code in the implementing class.

Java Interfaces

A class that implements an interface must define its own specific implementation of whatever methods the interface requires it to implement.

- The methods that the 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 it *is not an Abstract Class* (They are covered in Day 13)
- An interface cannot be instantiated; they can only be “implemented” by other classes.

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 Class>> implements <<Name of Interface>> {...}
```

- The class implementing an interface is also called a **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***; the method 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()
- These abstract methods do not have method bodies. There are no {*//code blocks.*}, and they each **end with a semicolon.**

Java Interfaces: Abstract Methods

A class implementing the Vehicle interface ***must*** provide a concrete implementation of the two abstract methods.

```
package te.mobility;  
  
public interface Vehicle {  
  
    public void honkHorn();  
    public 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 the interface abstract methods in a concrete class, the following rules are in effect:

- To fulfill the Interface's contract, the concrete class must implement the method with the exact same return type, name, and number of arguments with matching 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 in interfaces 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 {

    public 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 concrete method takes priority. The output from Car.honkHorn() 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 be public, static, and final.**

Java Interfaces: Polymorphism

Polymorphic objects are those that can assume many forms. In other words, they can pass more than one “Is-A” test.

- A child object from a parent class that implements an interface:
 - is a member of the child class
 - and a member of the parent class
 - And a member of interface “class”.
- Yesterday we instantiated a ReserveAuction with
`ReserveAuction lowReserve = new ReserveAuction ()`
lowReserve Is-A ReserveAuction **and** Auction.

Polymorphism is the ability to leverage these relationships in order to write more compact and reusable code.

Java Interfaces: Polymorphism References

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

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

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.

Java Interfaces: Polymorphism Example

Assuming that Car and Truck implements vehicle, consider a new class called RepairShop. In the real world, it is very likely that a car repair shop is able to handle more than one type of vehicle.

```
package te.main;

import te.mobility.Car;

public class RepairShop {

    public void repairVehicle(Car damagedCar) {
        System.out.println("repairing");
    }

}
```

Clearly there is an issue here, the RepairShop only accepts objects of class Car!

We can of course bypass this issue by creating yet another method that accepts Trucks.

Java Interfaces: Polymorphism Example

We can leverage interfaces to make the `repairVehicle` method much more flexible, allowing it to take in any `Vehicle`.

```
package te.main;

import te.mobility.Car;

public class RepairShop {

    public void repairVehicle(Vehicle damagedVehicle) {
        System.out.println("repairing");
    }

}
```

The method will now accept objects of class `Car`, and objects of class `Truck`.

Java Interfaces: Polymorphism Example

We can leverage interfaces to make the `repairVehicle` method much more flexible, allowing it to take in any `Vehicle`.

```
package te.main;
import te.mobility.Car;
import te.mobility.Truck;
import te.mobility.Vehicle;

public class Garage {

    public static void main(String[] args) {

        Vehicle fastCar = new Car();
        Vehicle bigTruck = new Truck();
        RepairShop repairShop = new RepairShop();

        repairShop.repairVehicle(fastCar);
        repairShop.repairVehicle(bigTruck);

    }
}
```

Both of these calls are ok to make because both `Cars` and `Trucks` are concrete classes implementing `Vehicle`.