

**HOW MANY PROGRAMMERS DOES
IT TAKE TO CHANGE A LIGHT BULB?**

A still from the TV show 'The Office' featuring Steve Moss. He is in a cubicle, looking at a light bulb in his hand with a concerned expression. The background shows the office environment with cubicles and fluorescent lights.

**NONE. THAT'S A
HARDWARE PROBLEM**

Module 1-12

Interfaces

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

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 other classes
- **Polymorphism** - the ability for our code to take different forms
- **(Abstraction)** – extension of encapsulation. We can't build a car from scratch, but we know how to use (drive) it.



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.*
- A fast food restaurant must have the exact menu stipulated by the franchise.
 - *The franchise will not send its own cooks to the restaurant, it is the franchisees responsibility to hire local cooks and make sure that the food is cooked to specification.*

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 another 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

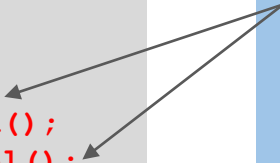
An abstract method is one that doesn't have an implementation, that is to say it has no body. Here is an example from a Vehicle Interface:

```
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();  
}
```

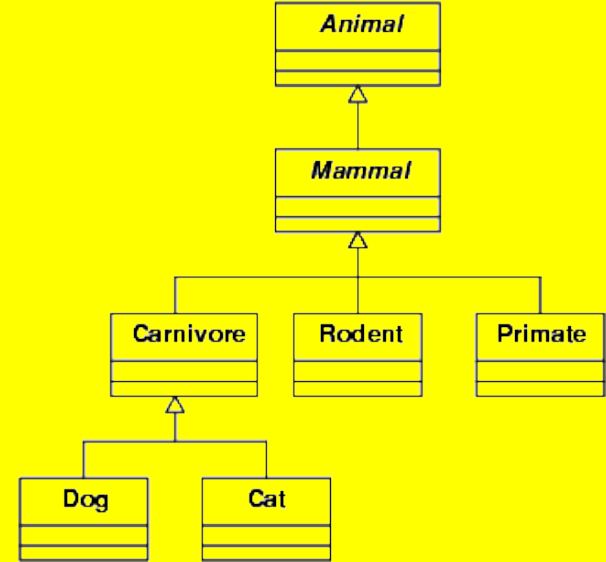
honkHorn has
been
implemented

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;  
    }  
}
```

//abstract parent class

```
abstract class Animal{  
    //abstract method  
    public abstract void sound();  
}  
//Dog class extends Animal class  
public class Dog extends Animal{  
  
    public void sound(){  
        System.out.println("Woof");  
    }  
    public static void main(String args[]){  
        Animal obj = new Dog();  
        obj.sound();  
    }  
}
```



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

Polymorphic objects are those that pass more than one “Is-A” test.

- A child object from a class that implements a parent interface is a member of the child class. It is also a member of the parent interface class.
- Consider the example we’ve worked with so far: If I instantiate a car with `Car myCar = new Car();` then myCar is a car, but it is also a vehicle.

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();
```

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.

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();

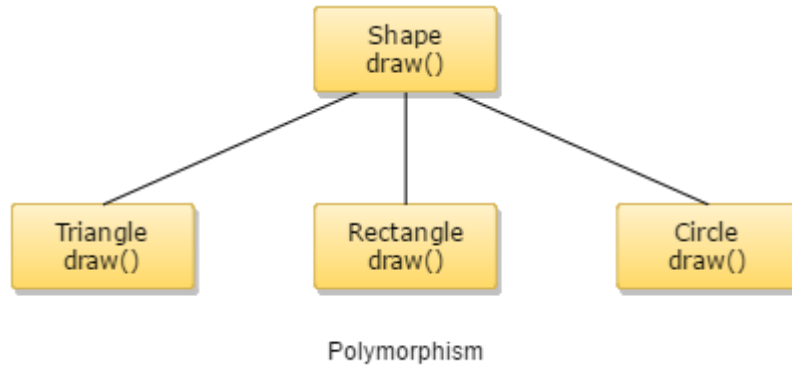
        repairShop.repairVehicle(fastCar);
        repairShop.repairVehicle(bigTruck);
    }
}
```

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



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

