Object Oriented Programming with Java

Interfaces

What Need Does an Interface Address?

Interfaces have a two-fold purpose:

Goal 1:

When we learned about inheritance, we stated that there can only be one parent class: no multiple inheritance allowed in Java

An interface is a construct that allows us to bypass this restriction. A class is allowed to “inherit” from an unrestricted number of interfaces.

What Need Does an Interface Address?

Interfaces have a two-fold purpose:

Goal 2:

Sometimes we have classes that don't have any common **properties** (e.g. `Milk`, `Car`, `Ticket`, etc.), so it wouldn't make a good OOP design to use inheritance and create a parent class for them.

But these classes have some common **behavior** (they are products we can sell) and it would make absolute sense to somehow put them under the same roof.

We can even create an array of `Sellable` items even though they're all different types, without a common parent class (other than `Object`, which is pretty useless in this scenario)

Interface: Basics

- An interface is practically a class with a set of **abstract methods**
- All interface methods are by default **public** and **abstract**
- An interface is a **type** and can be used for polymorphism
- An interface is *implemented* (sort of inherited) by a class by using the **`implements`** keyword instead of **`extends`**
- Any class can implement an interface by providing implementations of all the methods found in that interface
- A single class can implement multiple interfaces by implementing all the methods of all the *inherited* interfaces
- Interfaces do not have constructors

Interface: Advanced

- Java 8 introduced the concept of **static** and **default** methods:
  - **static** methods have a complete definition and the implementing class cannot override them. Static methods belong to the Interface class, i.e. you can only invoke them on the interface, not on the instance of the class that implements the interface
  - **default** methods, contrary to abstract methods, have a complete definition but the implementing class *can* override them. It's useful when you want to expand an interface but maintain backwards compatibility with older code that implements this interface.
- Interfaces can also have attributes which by default are **public**, **static** and **final**. The use of attributes should be avoided in most cases; use a utility class or an enumeration instead.

Declaring an Interface

We create an interface by declaring what methods must be implemented to be able to behave like this new type of thing.

All methods must be **abstract**: method signature with a semicolon instead of a body (implementation).

- ✔ We've seen abstract methods for abstract classes. These are the same thing, and can similarly be overridden with new implementations.

```
public interface Sellable
{
    public float getPrice();
    public String getSeller();
}
```

Implementing an Interface

A class that implements an interface must provide the body for all its methods

```
public class Car extends Vehicle implements Sellable {
    private float price;
    private String owner;
    private int maxSpeed;

    public Car (float price, String owner, int maxSpeed) { ... }

    public int maxSpeed() { ... }

    //implementations for all Sellable methods
    public float getPrice() {
        return price;
    }

    public String getSeller() {
        return "Last owner " + owner;
    }
}
```

Using an Interface

Now that we have implementations of our **`Sellable`** interface, we can use the Sellable **type** (assume the other classes implemented it too: **`Milk`**, **`House`**, etc.

```
public static void main (String[] args) {
  Milk milk = new Milk(...);
  House house = new House(...);

  //like superclasses, we can assign any Sellable thing to a Sellable variable
  Sellable sellable = house;

  //in the following, s, m, h could be used in each others' places
  System.out.println("House sold! "+sellable.getSeller());
  priceCheck(milk);
  priceCheck(house);
}

public static void priceCheck(Sellable sell) {
  System.out.println("checking price for "+sell+":");
  System.out.println(sell.getPrice());
}
```

Practice Problems

1. Create the `Comparison` interface which has the `compare` method that is used to compare two objects of the same type. The choice of return type and parameters is yours. Then create the `Car` class that implements the `Comparison` interface, and an executable class (e.g. `App`) that creates two `Car` objects and compares them.

2. What if you wanted to compare a `Car` object with a `Truck` object? Does the order of comparison matter? How can inheritance help in this case?

3. What if you wanted to compare a `Car` object with a `House` object? Does inheritance help here or you need something else?

4. What if you wanted to allow comparison based on different criteria?

Practice Problems

Create objects from two classes that implement an interface. Store them in variables named **a** and **b**, of the appropriate class type. Then create a variable whose type is the interface type. Call it **i**.

1. Can we store **a** or **b** into **i**?
2. Can we store **i** into **a** or **b**?
3. Can we use casting to our advantage?

4. Write a method accepting a parameter of your interface type.
5. Use it.

Examples of Interfaces

java.lang.Comparable. one method:

```
public int compareTo(Object other);
```

→ return negative : "less than", zero : "equal", positive : "greater than".
→ gives us a consistent way to sort data.

Example: The String class implements Comparable. (`compareTo` is available on Strings)

---

java.lang.Iterator. Three methods:

```
public boolean hasNext();
public Object  next   ();
public void    remove ();
```