Chapter 8
Interfaces

Hello!

The next few topics will be interfaces, enumerations, and exceptions. Interfaces and enumerations both help us introduce new types (similar to how classes did for us previously). Exceptions will be explored in more depth soon, which rely heavily upon types for us to differentiate between different exception types.

Interfaces

Java does not allow multiple inheritance; that is to say, each class has exactly one parent class. If we wanted to have multiple inheritance – say, that a Cow class that is an Animal, is Food, is Sellable at auction – we would not be allowed to extend all three of those classes. In reality, it’s not that a Cow really is multiple things at once; it’s that a Cow is one thing (an Animal), and yet we can interact with it in other specific ways: eat it, sell it, ride it, and so on.

What we really want, instead of multiple inheritance, is the chance to interact with an object in some extra way. We expect extra behaviors to be guaranteed by the objects of this class. Behaviors mean methods, so the real goal here is to be guaranteed that certain methods are available for all things that could be eaten, or all things that could be sold, or all things that could be ridden (be it a cow, a car, a wave, or a bike).

An interface is a grouping of abstract methods that any class may implement.
- A class implements an interface by overriding (implementing) every single method of the interface.
- The methods are abstract, because we expect the various classes to provide the definitions.
- We can think of an interface as a contract: any class that implements all the methods of the Fooable interface can behave like a Fooable thing.

An interface is a type.

Creating an Interface

We create an interface in similar fashion to creating a new class or enumeration (explored in the next section): we create a separate file with the same name as the interface and place the definition inside.

```java
public interface Zippable {
    public abstract boolean isClosedZ();
    public abstract void openZ();
    public abstract void closeZ();
}
```
A quick note on naming interfaces: because the purpose of an interface is to be able to interact with it in some fashion (by calling certain methods), the names might tend to be Somethingable: Sellable, Serializable, Comparable, and so on. It's just a convention, but it does help emphasize that we can interact with the objects of this class in another way.

Every single method in the interface must be abstract, so you can actually safely omit the abstract keyword here. But the abstract keyword is still required for abstract methods in classes, so you can simplify your life and always write abstract if you'd like.

Your Turn!

- Create an interface named Listenable. Give it abstract methods for listen() and ignore(). Choose what parameters or return types you feel are appropriate.
- What various classes could be Listenable? Try to come up with at least two examples.

Implementing an Interface

Now that we have an existing interface, we can cause any class to implement it by adding implements Zippable to the declaration, and then by overriding (implementing) every single method that was listed in the interface.

For our Zippable example, this means we must implement isOpenZ, openZ, and closeZ.

```java
public class Mouth implements Zippable {

    // the class has its own fields
    public boolean lipsOpen;

    // the class has its own constructors, other methods, etc.
    public Mouth (...) {...}
    public void eat(Food f) {...}

    // the class implements all methods of the Zippable interface:
    public boolean isOpenZ() { return lipsOpen; }
    public void openZ () { lipsOpen = true; }
    public void closeZ () { lipsOpen = false; }
}

public class Purse implements Zippable {

    // the usual parts of a class definition: fields, methods, etc.
    public Zipper z;
    ...

    // now, we implement all methods from Zippable:
    public boolean isOpenZ() { return z.isOpen(); }
    public void openZ () { z.open(); }
    public void closeZ () { z.close(); }
}

Your Turn!

- implement your Listenable interface with both of your example classes. (Perhaps SignificantOther, Record, Ocean, or Phone? Friend, Roman, Countryman?)
Using An Interface Implementation

Now that classes `Mouth` and `Purse` have implemented `Zippable`, we can now use the `Zippable` behavior whenever we have a `Mouth` object or a `Purse` object. Remember that we stated an interface is a type. This means we can use the interface wherever a type was required, such as at declaration time for a variable or parameter.

```java
Mouth m = new Mouth();
Purse p = new Purse();

m.closeZ();
p.openZ();

if (m.isOpenZ()) {
    System.out.println("my lips are not sealed! :-O");
    m.eat(new Food("potato chip")); // pretend the Food class exists...
}

// We can create a Zippable variable.
Zippable z = m;
z.openZ();

z = p;
z.closeZ();
```

We can also use the interface as a type for parameters to methods:

```java
public void closeIfNeeded(Zippable z) {
    if (z.isOpenZ()) {
        z.closeZ();
    }
}
```

- Although there is no `Zippable` class, and thus no instances (objects) exactly of type `Zippable` and no `Zippable` constructor, we can create objects of classes that do implement `Zippable`, and use references to these objects as the `Zippable` actual parameters.
- By choosing the `Zippable` type for the parameter, all we can do with it is call the methods of the `Zippable` interface on the object. We have no idea what else might be available other than those methods found in the interface.

```java
Mouth m = new Mouth();
closeIfNeeded(m);

Purse p = new Purse();
closeIfNeeded(p);
```
Your Turn!

• Create objects of the classes that implemented Listenable. Store them in variables of their own class types.
• Call the Listenable methods on these objects.
• Create a variable of type Listenable; store your various objects that are Listenable into it.
• Call the Listenable methods on your Listenable variable. This is all you can do with the Listenable variable.
• Create a method named performCustomerSupport that accepts a Listenable thing, and always ignores it.
• Create an array of Listenable objects. Use a for-each loop to listen to each thing in your array.

Implementing multiple Interfaces

A class can implement multiple interfaces: we just add the interface names in a comma-separated list after the implements keyword, and then provide all the methods of each interface that is being implemented.

```java
public class Foo implements A,B,C {
    //Foo stuff
    // A methods here
    // B methods here
    // C methods here
}
```

Example Java Interfaces

Java uses interfaces in a couple of interesting ways. Two interfaces we will consider are Comparable and Iterator.

**Comparable** is used to order values. Think of it as a way of answering the question "which one is greater?" by encoding the answer as a number. Comparable has one method:

```java
public interface Comparable {
    public int compareTo (Object other);
}
```

If an invocation (such as `a.compareTo(b)`) returns a negative number, it implies a "less than" relationship (`a<b`); if the number is positive, it indicates a "greater than" relationship (`a>b`). And if the result is zero, then it implies "equal" (`a=b`).

It is up to the class designer to decide what constitutes "greater than", and then implement the `compareTo` method accordingly. We might decide that our `Square` class will implement `Comparable` by comparing the sizes:
public class Square implements Comparable {

    public int side;

    public Square (int side) {
        this.side = side;
    }

    // implement all (1) Comparable methods.
    public int compareTo(Object so) {
        // parameter needs to be Object.
        Square s = (Square) so; // we cast it to our desired type.
        if (side<s.side) { return -1; }
        if (side>s.side) { return 1; }
        else return 0;
    }
}

Your Turn!

• Implement Comparable in any class. Consider the different ways you might want to define the relation: for Cars, is the mpg all that matters? The top speed? The maximum passengers? It's often obvious what the relation should be, but in practice whatever is the most meaningful for the program (and any future programs using this class) is what should dictate the decision.

The Iterator interface provides three methods:

public boolean hasNext(); // does this collection of values have any more values?
public Object next(); // assuming there's another value, get the next one.
public void remove(); // remove the item that the previous next() call returned.

If we were to create any type of structure where we wanted to allow some internal values to be regularly accessed, we could just implement the Iterator interface, and then the for-each loop syntax would be readily available! We will learn how to make some basic data structures later on, and so we might have a chance to implement Iterator before the course ends on a realistic data structure.

Your Turn!

• Create a class named SizeTen that has a field of type int[] which always has ten values in it (enforce this in your constructor).
• Make this class implement Iterator. You can actually ignore the remove() method when removal doesn't make sense, so just implement hasNext() and next().
• Test out your SizeTen class by writing a for-each loop:

        for (int i : mySizeTen) {
            ...use i...
        }