Interfaces, Inner Classes, and Packages

Interfaces

An **interface** in Java is like a class that hasn't been implemented yet. An interface contains a list of method headers, none of which are implemented. It is a way to list the features you want in a particular class without yet deciding how to implement them. Interfaces are very useful with data structures because we can list the functionality we'd like the data structure to have, and then implement them in several different ways.

Declaring an Interface

Here's a sample interface:

```
public interface ArrayCollection {
     void add(Object elem);

     void print();
}
```

This interface should be stored in the file `ArrayCollection.java`. This interface defines what functionality we'd like a collection of general elements to have. We don't know exactly how we're going to implement this collection, but we do know that we want to be able to add elements and print elements. Notice that **we do not include visibility modifiers in an interface** (they are all public).

Implementing an Interface

Of course, we do eventually have to write code for array collections, or we won't be able to use the interface. To do this, we write a class that implements the `ArrayCollection` interface. If a class implements an interface, then it must implement every method defined in the interface. The method headers in the class must look EXACTLY like the method headers in the interface.

First, we'll implement the `ArrayCollection` interface using the by storing **all** elements added in an array (up to the maximum size):

```
public class ArrayList implements ArrayCollection {
     private Object[] arr;
     private int size;

     public ArrayList(int max) {
          arr = new Object[max];
          size = 0;
     }

     public void add(Object elem) {
```

```
            //Don't add if we've reached our max size
            if (size == arr.length) return;

            arr[size] = elem;
            size++;
        }

        public void print() {
            for (int i = 0; i < size; i++) {
                System.out.println(arr[i].toString());
            }
        }
    }
```

Similarly, we can implement the `ArrayCollection` interface using by only storing each **unique** element added (up to the maximum size):

```
    public class ArraySet implements ArrayCollection {
        private Object[] arr;
        private int size;

        public ArraySet(int max) {
            arr = new Object[max];
            size = 0;
        }

        public void add(Object elem) {
            //Don't add if we've reached our max size
            if (size == arr.length) return;

            //Check if elem is already in array
            for (int i = 0; i < size; i++) {
                if (val.equals(elem)) return;
            }

            //Now add elem (no duplicates)
            arr[size] = elem;
            size++;
        }

        public void print() {
            for (int i = 0; i < size; i++) {
                System.out.println(arr[i].toString());
            }
        }
    }
```

Because `ArrayCollection` is an interface, we can't create a new `ArrayCollection` object. However, we can store either a `ArrayList` or a `ArraySet` object in an `ArrayCollection` variable:

```
ArrayCollection coll = new ArrayList();

     - OR-
ArrayCollection coll = new ArrayCollection();
```

Then, we can call the `ArrayCollection` methods:

```
coll.add(2);
coll.add(2);
```

If we created an `ArrayList` object, then the 2 would be added twice. If we created an `ArraySet` object, the 2 would only be added once (since no duplicates are allowed). Thus we can very quickly change our functionality by creating an `ArraySet` or `ArrayList`, but the code **using** the collection would remain the same.

Abstract Classes

In the example above, both the `ArraySet` and `ArrayList` implementations have the same instance variables and the same `print` method. However, their `add` methods are different. Thus it makes sense to define the instance variables and `print` methods in a parent class, but to not implement the `add` method yet.

An **abstract class** is halfway between a parent class and an interface. Abstract classes can declare instance variables, and they can also contain completed methods. However, they must contain at least one unimplemented method (like the method header in an interface). An unimplemented method is called an **abstract method**.

Declaring an Abstract Class

An abstract class is declared with the keyword "`abstract`":

```
public abstract class Name {

}
```

Any abstract methods must also be declared with the keyword `abstract`:

```
visibility abstract returnType name(args);
```

Here is `ArrayCollection` rewritten as an abstract class:

```
public abstract class ArrayCollection {
     protected Object[] arr;
     protected int size; //we can declare instance variables

     //We don't know how we're storing elements
     //So we don't know how to add yet
     public abstract void add(Object num);

     //We already know how to implement print()
     public void print() {
          for (int i = 0; i < size; i++) {
               System.out.println(arr[i].toString());
          }
     }
}
```

Note that we can now declare instance variables (`arr`, `size`) and have implemented methods (`print()`). However, the class is abstract because the `add` method is not implemented. This class represents an array of general object elements, but we don't yet know how to implement the collection. We know that we would like an add operation, but don't know how to write the code for it.

Extending an Abstract Class

Extending an abstract class is exactly like extending an ordinary class:

```
public class ChildName extends ParentName {

}
```

The only difference is that this class must implement every abstract method from the parent class.

Here is how we can implement the abstract class `ArrayCollection` by allowing duplicates.

```
public class ArrayList extends ArrayCollection {
     //We inherit size and arr

     public ArrayList(int max) {
          arr = new int[max];
          size = 0;
     }

     //add is no longer abstract
     public void add(Object elem) {
          if (size == arr.length) return;
```

```
            arr[size] = elem;
            size++;
        }

        //We inherit print()
    }
```

Here are some examples of using the `ArrayList` class:

```
    ArrayList ac = new ArrayList(10);

    //ArrayList extends ArrayCollection
    ArrayCollection c = new ArrayList(5);

    ac.add(1);
    ac.add(2);

    c.add("a");
    c.add("b");
    c.add("c");

    ac.print();     //ArrayList inherits print(), prints 1, 2
    c.print();      //prints a, b, c
```

Inner Classes

An inner class is a class declared INSIDE another class. This is done when we want the details of the inner class to only be visible to the class it's inside. In this case, we declare the inner class private. Here's the format:

```
    public class OuterClass {
        //contents of OuterClass

        private class InnerClass {
            //contents of InnerClass
        }
    }
```

This code should be stored in the file `OuterClass.java`. No other class (besides `InnerClass` and `OuterClass`) knows that `InnerClass` exists.

For example, suppose we want to store the names and ages of a bunch of people. We can have an inner class called Person, that stores a single person's name and age. Then, the outer class can have an array of type Person:

```
public class People {
	private int size = 0;
	private Person[] list = new Person[100];

	//Add a new person, if there is room in the array
	public void add(String name, int age) {
		//If there is room left
		if (size < list.length) {
			//Create a new person, and add it to the array
			Person p = new Person(name, age);
			list[size] = p;
			size++;
		}
	}

	//Return the age of the given person
	//Return -1 if the person isn't in our array
	public int getAge(String name) {
		//loop through all people
		for (int i = 0; i < size; i++) {
			//if this person's name matches what we're looking for
			if (list[i].name.equals(name)) {
				//return this person's age
				return list[i].age;
			}
		}

		//We haven't returned yet, so we must have not
		//found the name in our array.  Return -1.
		return -1;
	}

	private class Person {
		public String name;
		public int age;

		public Person(String name, int age) {
			this.name = name;
			this.age = age;
		}
	}
}
```

Here's how we might use the `People` class:

```
People p = new People();
p.add("Fred", 18);
```

```
p.add("James", 20);
System.out.println(p.getAge("Fred"));     //prints 18
```

However, we can't do:

```
Person pers = new Person("Amy", 26);
```

from outside the `People` class, since we can't see `Person` outside of `People`.

Packages

When you start writing industrial-sized software programs, you will start using hundreds – perhaps thousands – of classes. If all these classes were stored in the same directory, it would be very difficult to keep straight what anything did. A Java **package** allows us to store related classes and interfaces in a separate directory.

The biggest program we will write in this class will have about 10 classes, which is not big enough to need packages. However, we will separate our files into packages anyway, so as to get practice using them.

Putting Code in a Package

When you decide to place a class or interface in a package, first place that class or interface in a directory with the package name. Next, add the line:

```
package packageName;
```

to the top of the file, where `packageName` is the name of the package. (Note: most development environments handle packages for you. If you are using something like Eclipse, find an "Add Package" option and supply the name of the package. Next, try to add a class or interface to that package. The development environment will handle all the details for you.)

Using a Package

If you want to use code from a package called `packageName`, add:

```
import packageName.*;
```

to the file where you want to use the code. This is exactly like importing Java packages. Now, you will be able to use all the classes and interfaces in that package.