Chapter 1. Interfaces

This book presents three topics:

Data structures

: Starting with the structures in the Java Collections Framework (JCF), you will learn how to use data structures like lists and maps, and you will see how they work.

Analysis of algorithms

: I present techniques for analyzing code and predicting how fast it will run and how much space (memory) it will require.

Information retrieval

: To motivate the first two topics, and to make the exercises more interesting, we will use data structures and algorithms to build a simple web search engine.

Here's an outline of the order of topics:

- We'll start with the `List` interface and you will write classes that implement this interface two different ways. Then we'll compare your implementations with the Java classes `ArrayList` and `LinkedList`.
- Next I'll introduce tree-shaped data structures and you will work on the first application: a program that reads pages from Wikipedia, parses the contents, and navigates the resulting tree to find links and other features. We'll use these tools to test the "Getting to Philosophy" conjecture (you can get a preview by reading *http://thinkdast.com/getphil*).
- We'll learn about the `Map` interface and Java's `HashMap` implementation. Then you'll write classes that implement this interface using a hash table and a binary search tree.
- Finally, you will use these classes (and a few others I'll present along the way) to implement a web search engine, including a crawler that finds and reads pages, an indexer that stores the contents of web pages in a form that can be searched efficiently, and a retriever that takes queries from a user and returns relevant results.

Let's get started.

Why Are There Two Kinds of List?

When people start working with the Java Collections Framework, they are sometimes confused about `ArrayList` and `LinkedList`. Why does Java provide two implementations of the `List` interface? And how should you choose which one to use? I will answer these questions in the next few chapters.

I'll start by reviewing `interface`s and the classes that implement them, and I'll present the idea of "programming to an interface".

In the first few exercises, you'll implement classes similar to `ArrayList` and `LinkedList`, so you'll know how they work, and we'll see that each of them has pros and cons. Some operations are faster or use less space with `ArrayList`; others are faster or smaller with `LinkedList`. Which one is better for a particular application depends on which operations it performs most often.

Interfaces in Java

A Java `interface` specifies a set of methods; any class that implements this `interface` has to provide these methods. For example, here is the source code for `Comparable`, which is an `interface` defined in the package `java.lang`:

```
public interface Comparable<T> {

    public int compareTo(T o);

}
```

This `interface` definition uses a type parameter, `T`, which makes `Comparable` a **generic type**. In order to implement this `interface`, a class has to

- Specify the type `T` refers to, and

- Provide a method named `compareTo` that takes an object as a parameter and returns an `int`.

For example, here's the source code for `java.lang.Integer`:

```
public final class Integer extends Number implements
Comparable<Integer> {



    public int compareTo(Integer anotherInteger) {

        int thisVal = this.value;

        int anotherVal = anotherInteger.value;

        return (thisVal<anotherVal ? -1 :
(thisVal==anotherVal ? 0 : 1));

    }



    // other methods omitted

}
```

This class extends `Number`, so it inherits the methods and instance variables of `Number`; and it implements `Comparable<Integer>`, so it provides a method named `compareTo` that takes an `Integer` and returns an `int`.

When a class declares that it implements an `interface`, the compiler checks that it provides all methods defined by the `interface`.

As an aside, this implementation of `compareTo` uses the "ternary operator", sometimes written `?:`. If you are not familiar with it, you can read about it at *http://thinkdast.com/ternary*.

List Interface

The Java Collections Framework (JCF) defines an `interface` called `List` and provides two implementations, `ArrayList` and `LinkedList`.

The `interface` defines what it means to be a `List`; any class that implements this `interface` has to provide a particular set of methods, including `add`, `get`, `remove`, and about 20 more.

`ArrayList` and `LinkedList` provide these methods, so they can be used interchangeably. A method written to work with a `List` will work with an `ArrayList`, `LinkedList`, or any other object that implements `List`.

Here's a contrived example that demonstrates the point:

```
public class ListClientExample {

    private List list;



    public ListClientExample() {

        list = new LinkedList();

    }



    private List getList() {

        return list;

    }



    public static void main(String[] args) {

        ListClientExample lce = new ListClientExample();

        List list = lce.getList();
```

```
            System.out.println(list);

        }

    }
```

`ListClientExample` doesn't do anything useful, but it has the essential elements of a class that **encapsulates** a `List`; that is, it contains a `List` as an instance variable. I'll use this class to make a point, and then you'll work with it in the first exercise.

The `ListClientExample` constructor initializes `list` by **instantiating** (that is, creating) a new `LinkedList`; the getter method called `getList` returns a reference to the internal `List` object; and `main` contains a few lines of code to test these methods.

The important thing about this example is that it uses `List` whenever possible and avoids specifying `LinkedList` or `ArrayList` unless it is necessary. For example, the instance variable is declared to be a `List`, and `getList` returns a `List`, but neither specifies which kind of list.

If you change your mind and decide to use an `ArrayList`, you only have to change the constructor; you don't have to make any other changes.

This style is called **interface-based programming**, or more casually, "programming to an interface" (see *http://thinkdast.com/interbaseprog*). Here we are talking about the general idea of an interface, not a Java `interface`.

When you use a library, your code should only depend on the interface, like `List`. It should not depend on a specific implementation, like `ArrayList`. That way, if the implementation changes in the future, the code that uses it will still work.

On the other hand, if the interface changes, the code that depends on it has to change, too. That's why library developers avoid changing interfaces unless absolutely necessary.

Exercise 1

Since this is the first exercise, we'll keep it simple. You will take the code from the previous section and **swap the implementation**; that is, you will replace the `LinkedList` with an `ArrayList`. Because the code programs to an interface, you will be able to swap the implementation by changing a single line and adding an `import` statement.

Start by setting up your development environment. For all of the exercises, you will need to be able to compile and run Java code. I developed the examples using Java SE Development Kit 7. If you are using a more recent version, everything should still work. If you are using an older version, you might find some incompatibilities.

I recommend using an interactive development environment (IDE) that provides syntax checking, auto-completion, and source code refactoring. These features help you avoid errors or find them quickly. However, if you are preparing for a technical interview, remember that you will not have these tools during the interview, so you might also want to practice writing code without them.

If you have not already downloaded the code for this book, see the instructions in "Working with the Code".

In the directory named `code`, you should find these files and directories:

- `build.xml` is an Ant file that makes it easier to compile and run the code.
- `lib` contains the libraries you'll need (for this exercise, just JUnit).
- `src` contains the source code.

If you navigate into `src/com/allendowney/thinkdast`, you'll find the source code for this exercise:

- `ListClientExample.java` contains the code from the previous section.
- `ListClientExampleTest.java` contains a JUnit test for `ListClientExample`.

Review `ListClientExample` and make sure you understand what it does. Then compile and run it. If you use Ant, you can navigate to the `code` directory and run `ant ListClientExample`.

You might get a warning like:

```
List is a raw type. References to generic type List<E>

should be parameterized.
```

To keep the example simple, I didn't bother to specify the type of the elements in the `List`. If this warning bothers you, you can fix it by replacing each `List` or `LinkedList` with `List<Integer>` or `LinkedList<Integer>`.

Review `ListClientExampleTest`. It runs one test, which creates a `ListClientExample`, invokes `getList`, and then checks whether the result is an `ArrayList`. Initially, this test will fail because the result is a `LinkedList`, not an `ArrayList`. Run this test and confirm that it fails.

NOTE: This test makes sense for this exercise, but it is not a good example of a test. Good tests should check whether the class under test satisfies the requirements of the *interface*; they should not depend on the details of the *implementation*.

In the `ListClientExample`, replace `LinkedList` with `ArrayList`. You might have to add an `import` statement. Compile and run `ListClientExample`. Then run the test again. With this change, the test should now pass.

To make this test pass, you only had to change `LinkedList` in the constructor; you did not have to change any of the places where `List` appears. What happens if you do? Go ahead and replace one or more appearances of `List` with `ArrayList`. The program should still work correctly, but now it is "overspecified". If you change your mind in the future and want to swap the interface again, you would have to change more code.

In the `ListClientExample` constructor, what happens if you replace `ArrayList` with `List`? Why can't you instantiate a `List`?