CS61B Lecture 11

Friday, February 14, 2020

`Comparable`

The Java library provides an interface to describe `Object`s that have a natural order on them, such as `String` and `Integer`. For now, we will use the old Java 1.4 version:

```
public interface Comparable { // For now, the Java 1.4 version
    /** Returns value <0, == 0, or > 0 depending on whether THIS is
     *  <, ==, or > OBJ. Exception if OBJ not of compatible type. */
     int compareTo(Object obj);
}
```

We might use this in a general-purpose `max` function for example:

```
/** The largest value in array A, or null if A empty. */
public static Comparable max(Comparable[] A) {
    if (A.length == 0) { return null; }
    Comparable result; result = A[0];
    for (int i = 1; i < A.length; i += 1) {
        if (result.compareTo(A[i]) < 0) result = A[i];
        return result;
    }
}
```

Now `max(S)` will return the maximum value in `S` if `S` is an Object that implements `Comparable`.

Implementing `Comparable`

Here's how we can write a class that implements the `Comparable` interface:

```
/** A class representing a sequence of ints. */
class IntSequence implements Comparable {
    private int[] myValues;
    private int myCount;
    ...
    public int get(int k) { return myValues[k]; }

    @Override
    public int compareTo(Object obj) {
        IntSequence x = (IntSequence) obj; // Blows up if obj not an IntSequence
        for (int i = 0; i < myCount && i < x.myCount; i += 1) {
            if (myValues[i] < x.myValues[i]) { return -1;
            } else if (myValues[i] > x.myValues[i]) {
                return 1;
            }
            return myCount - x.myCount; // <0 iff myCount < x.myCount
        }
    }
```

```
}
```

It is also possible to add an interface retroactively: if `IntSequence` did not implement `Comparable`, but did implement `compareTo` without `@Override`, we could write:

```
class ComparableIntSequence extends IntSequence implements Comparable {
    ...
}
```

Java would then "match up" the `compareTo` in `IntSequence` with that in `Comparable`.

Java Generics

The `Comparable` we just showed earlier was the old Java 1.4 version. The current version uses a newer feature of Java: generic types.

```
public interface Comparable<T> {
    int compareTo(T x);
}
```

Here, `T` is like a formal parameter in a method, except it has a value of a type. We can then use this value throughout our class, and Java will substitute every instance of this parameter with whatever we decide to call it on:

```
class IntSequence implements Comparable<IntSequence> {
    ...

    @Override
    public int compareTo(IntSequence x) {
        for (int i = 0; i < myCount && i < x.myCount; i += 1) {
        if (myValues[i] < x.myValues[i]) { ... }
        return myCount - x.myCount;
    }
}
```

In this case, every instance of `T` is substituted with the type we passed in: `IntSequence`.

`Reader`

The Java class `java.io.Reader` abstracts sources of charcters. Here, we will give you an interface version (it is, in actuality, an abstract class, but let's say it is an interface for illustration purposes).

```
public interface Reader { // Real java.io.Reader is abstract class

    /** Release this stream: further reads are illegal */
    void close();

    /** Read as many characters as possible, up to LEN,
     *  into BUF[OFF], BUF[OFF+1],..., and return the
     *  number read, or -1 if at end-of-stream. */

    int read(char[] buf, int off, int len);
```

```
    /** Short for read(BUF, 0, BUF.length). */
    int read(char[] buf);

    /** Read and return single character, or -1 at end-of-stream. */
    int read();
}
```

Generic Partial Implementation

According to the specifications, some of the methods of `Reader` are related. We can express this with a **partial implementation**, which leaves key methods unimplemented and provides default bodies for others.

However, the result is still abstract: we still cannot use `new` on it.

```
/** A partial implementation of Reader. Concrete
 *  implementations MUST override close and read(,,).
 * They MAY override the other read methods for speed. */
public abstract class AbstractReader implements Reader {
    // Next two lines are redundant.
    public abstract void close();
    public abstract int read(char[] buf, int off, int len);
    public int read(char[] buf) {
        return read(buf,0,buf.length);
    }
    public int read() {
        return (read(buf1) == -1) ? -1 : buf1[0];
    }
    private char[] buf1 = new char[1];
}
```

Implementing `Reader`

The class StringReader reads characters from a String:

```
public class StringReader extends AbstractReader {
    private String str;
    private int k;
    /** A Reader that delivers the characters in STR. */
    public StringReader(String s) {
        str = s; k = 0;
    }
    public void close() {
        str = null;
    }
    public int read(char[] buf, int off, int len) {
        if (k == str.length())
        return -1;
        len = Math.min(len, str.length() - k);
        str.getChars(k, k+len, buf, off);
        k += len;
        return len;
    }
}
```

Using `Reader`

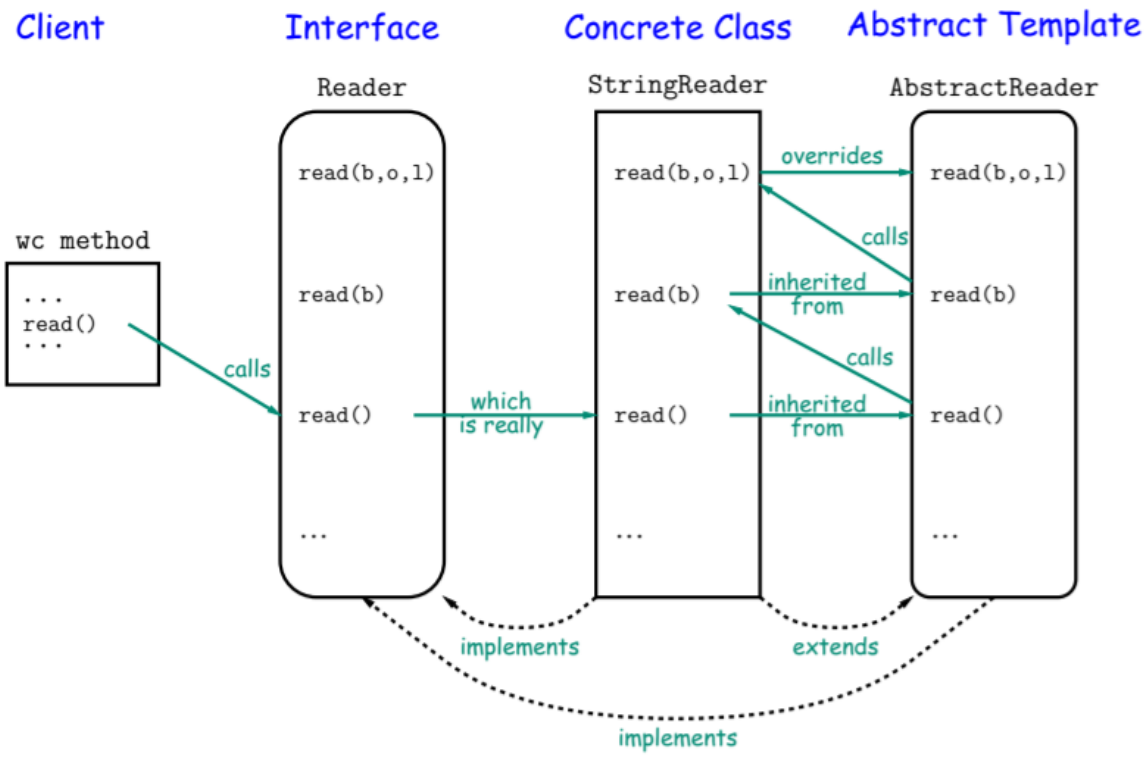
Consider this method, which counts words:

```
/** The total number of words in R, where a "word" is
 * a maximal sequence of non-whitespace characters. */
int wc(Reader r) {
    int c0, count;
    c0 = ' '; count = 0;
    while (true) {
        int c = r.read();
        if (c == -1) return count;
        if (Character.isWhitespace((char) c0)
        && !Character.isWhitespace((char) c))
        count += 1;
        c0 = c;
    }
}
```

This method works for any `Reader`:

```
wc(new StringReader(someText)) // # words in someText
wc(new InputStreamReader(System.in)) // # words in standard input
wc(new FileReader("foo.txt")) // # words in file foo.txt.
```

How It Fits Together



Conclusion

- The `Reader` interface class served as a specification for a whole set of readers.
- Ideally, most client methods that deal with `Reader`s, like `wc`, will specify type `Reader` for the formal parameters, not a specific kind of `Reader`, thus assuming as little as possible.

- And only when a client creates a new `Reader` will it get specific about what subtype of `Reader` it needs.
- That way, client's methods are as widely applicable as possible.
- Finally, `AbstractReader` is a tool for implementors of non-abstract `Reader` classes, and not used by clients.
- Alas, Java library is not pure. E.g., `AbstractReader` is really just called `Reader` and there is no interface. In this example, we saw what they should have done!
- The `Comparable` interface allows definition of functions that depend only on a limited subset of the properties (methods) of their arguments (such as "must have a `compareTo` method").