Object-Oriented Design	**Lecture 3**
CSU 370 Fall 2007 (Pucella)	**Friday, Sep 14, 2007**

Java

We will be programming in Java in this course. Partly because it is a reasonable language, and partly because you already have seen some Java in CSU 213. In fact, I will assume you know the basics of the language: how to define a basic class, derive some objects from the class, what are methods, what are fields (which we'll also call variables to confuse things sometimes), and what are the basic types such as integers, booleans, and some fundamental operations on those types.

Now, working in a specific language such as Java will lead to terminological problems. In particular, many of the terms we use when discussing design in a language-independent way (as we have been doing until now) also have a meaning in Java, a meaning which is often quite similar to the generic language-independent meaning, but often much more specific. For instance, the term *constructor* has a specific meaning in Java which is along the lines of the meaning we gave it last lecture, but actually means something much more specific, viz. a method automatically invoked when using `new` to create a new object. Another example is *interface*, which means something specific in Java that we will cover in a couple of lectures. There are many others. Generally, the context will make it clear whether I am talking about the generic term, or a specific term that has a technical meaning in Java. It is your job to try to keep the two apart. Please ask when you are unsure.

An Implementation of Voice ADT

The plan today is to give a quick and dirty implementation of the Voice ADT, to illustrate the basic uses of Java, and also to introduce the conventions and new features we will need in this course. Recall the signature we had:

```
emptyVoice :                 -> Voice
oneNote :              Note  -> Voice
concatenated : Voice x Voice -> Voice

isEmpty :    Voice -> boolean
firstNote :  Voice -> Note
otherNotes : Voice -> Voice
```

The first step is to convert this signature into an object oriented signature, using a more Java-like syntax. First, the creators:

```
public static Voice emptyVoice ();
public static Voice oneNote (Note);
public static Voice concatenated (Voice, Voice);
```

We will see the meaning of `public` and `static` in a little bit. For the accessors, we will use the fact that every method in Java has an implicit parameter of the same type of the object to which it is associated. Since all the accessors take a voice as a parameter, we will use the implicit parameter for this.

```
public boolean isEmpty ();
public Note firstNote ();
public Voice otherNotes ();
```

We can similarly convert the specification using the above syntax, for instance:

```
emptyVoice().isEmpty() = true
oneNote(n).isEmpty() = false
concatenated(v1,v2).isEmpty() = false
```

I leave the other equations as an exercise.

Let's implement the above signature, then. We will define a class `Voice` in a file `Voice.java`. Recall that in Java, we can only put one public class (that is, a world-accessible class) per file, and the name of the file should be the same name as the class, with `.java` appended.

```
// Class implementing the Voice ADT given in lecture
public class Voice {

}
```

Let's fill in the body of this class.

Now, the main rule we will follow in this class is that the only methods in the class that we should be able to invoke are those methods that are in the signature. Now, the class can define other methods, we just have to make sure they are not accessible from outside the class.

Java's lets us restrict accessibility to methods (and to fields) using the `private` keyword. In contrast, accessible methods can be specified using the `public` keyword. (Thus, our use of `public` in the signature above.)

In particular, because a classical Java constructor (the method invoked upon a `new`) is not described in the signature, we should make it private. Intuitively, the only way we allow to create a new object of the class is via one of the creators defined in the interface. These creators are sometimes called *factory methods*, and the technique of only using creators to create new objects instead of Java constructors is called the *factory design pattern.* The Java constructor for the class is rather simple:

```
  // Voice constructor, made private
  private Voice () { }
```

It really does nothing.

But now we have a problem. We would like to write a creator such as `emptyVoice` as a public method that calls the constructor (which it can because the `emptyVoice` method is inside the class). But how will we able to call the `emptyVoice` method? Recall that invoke a method, we need an objet of that class. And we just made sure that the constructor of the class was private, that is, not invokable from outside that class. So there is no way to have an object that would let us invoke any of the creators in the first place. Ouch. That's a problem.

The key is to somehow make sure that the creator methods are available even when no objects of the class exist. How can we do that? Well, it helps to think of the world of the program to be split in two. One part, the part that correspond to the source code, is the static part. Static means non-moving, which we take to be non-executing, and the static part of the program correspond to what information about the program we have before anything executes. In Java, the only thing we know before the program executes are what classes are defined. Thus, the classes belong to the static world. These classes exist, in some sense, even before programs start executing. The other part is the dynamic part, which corresponds to program execution. During executions, instances of the classes, that is, objects, get created, updated, destroyed. Thus, objects belong to the dynamic world.

To make sure that a method is available even when no object is present, we need to associate it with the *class* itself, rather than with the objects. That is, we need to move it to the static world. To do this, we use the `static` qualifier. Any method which is qualified to be a static method is a method that is available even when no object of the class has been created. To invoke a static method, instead of writing *object.method*(), we use *class.method*().[1] Creators will be static methods (hence why I wrote `static` in the signature above for the creators).

Before implementing the creators, though, we need to figure out how to represent a voice. This is a quick and dirty implementation, and I will use a quick and dirty solution. There is a pedagogical reason for doing that, which I will come back to at the end of the class. We will see a better way of implementing it later on, and you will be able to appreciate how much better that way is when we get to it.

We will represent a voice using two fields, one to hold the first note in the voice, and the other to hold the rest of the voice, minus the first note.

```
  private Note first;    // holds the first note of a voice
  private Voice rest;    // holds the rest of the notes of a voice
```

[1]There are some rules about static methods; in particular, a static method cannot refer to methods or fields that are defined in objects, which makes sense because a static method exists even when there are no objects around.

An empty voice will be represented by having both fields hold the null value. Because these fields are not part of the signature, they will be made private, and therefore not accessible from outside a voice object. (Thankfully, the accessors will be able to extract information from those fields.)

First, the creators.

```
  // Creator for an empty voice
  public static Voice emptyVoice () {
    Voice obj = new Voice ();
    obj.first = null;
    obj.rest = null;
    return obj;
  }

  // Creator for a voice made up of one note only
  public static Voice oneNote (Note n) {
    Voice obj = new Voice ();
    obj.first = n;
    obj.rest = emptyVoice ();
    return obj;
  }

  // Creator for a voice made up of two voices in succession
  public static Voice concatenated (Voice v1, Voice v2) {
    Voice obj;
    if (v1.isEmpty ())
      return v2;
    obj = new Voice ();
    obj.first = v1.firstNote ();
    obj.rest = concatenated (v1.otherNotes(),v2)
    return obj;
  }
```

Then, the accessors.

```
  // Predicate for an empty voice
  public boolean isEmpty () {
    return (this.first == null);
  }

  // Accessor for the first note of a voice
  public Note firstNote () {
    if (this.isEmpty())
      throw new Error ("firstNote() of empty voice");
    return this.first;
  }

  // Accessor for the rest of the notes of a voice
  public Voice otherNotes () {
    if (this.isEmpty())
      throw new Error ("otherNotes() of empty voice");
    return this.rest;
  }
```

A couple of things to notice. First off, I explicit use `this.` as a qualifier when invoking a method of the current object, or accessing a field of the current object. Also, when noticing an error, I throw an exception (in this case, the `Error` exception). We will see exceptions in more detail later on in the course.

I have also followed a couple of conventions for naming, which you should follow as well. The Java compiler will not enforce them, but your brain will learn to recognize them and use them to spot some errors some times. Class names are capitalized, like `Voice`, and later on `VoiceTester`. Method names and variable names are capitalized but for the first letter, like `oneNote`, or `first`.

All code should be commented. Not putting any comments is a sin that I will not permit you to indulge in. Every class should have a comment at the top indicating the purpose of the class, and every method and variable should have a comment indicating the role of the method or variable.

Tester and the canonical `toString` method

I would like to write a simple tester for the above code. To do that, I need a nice way to print out a representation of a voice. So let's add a method to the interface that let's us get a string representation of a voice for printing and debugging purposes. (We need to add it to the interface because we want the function to be available, and I told you that only operations in the interface should be available.)

So here is the signature, and the specification that you should add to the interface.

```
  public String toString ();

  emptyVoice().toString = ""
  oneNote(n).toString = n.toString()
  concatenated(v1,v2).toString = v1.toString() + " " + v2.toString()
```

This specification assumes that notes have a `toString` method attached to them as well.

It is straightforward to transform this specification into a method to add to the `Voice` class.

```
// Canonical method to print notes in the voice
public String toString () {
  if (this.isEmpty())
    return "";
  return (this.first.toString() + " " + this.rest.toString());
}
```

The name `toString` we used for this method is not random. It is one of those special names that Java understands implicitly. (There are several like that, if you read the Java documentation carefully.) We call those *canonical methods*. Here's how `toString` is special: whenever you use an object in a place where Java expects a string—for example, invoking `System.out.println` which expects a string and prints it out on the console—then Java will invoke the `toString` method of that object if it is defined instead of giving you back a type error.

With this in mind, we can now write a small testing program. Of course, we need an implementation for notes. Here is something simple that gets us off the ground, in a file `Note.java`. We can do nothing with notes except create them and print out their name.

```
// A simple implementation of a note
class Note {

  // Private constructor, taking the name of the note as argument
  private Note (String s) {
    this.name = s;
  }

  // The name of the note
  private String name;

  // Creator for a note, taking the name as an argument
  public static Note named (String s) {
    Note obj = new Note(s);
    return obj;
  }

  // Canonical method for printing name of a note
  public String toString () {
    return this.name;
  }
}
```

With this, we can write a very simple minded tester in `VoiceTester.java`:

```
// Simple tester for the Voice class
class VoiceTester {

  // The main testing function
  static public void main (String[] args) {
    Note nDo = Note.named("do");
    Note nRe = Note.named("re");
    Note nMi = Note.named("mi");
    Note nFa = Note.named("fa");
    Voice v1 = Voice.concatenated (Voice.oneNote (nFa),
                                   Voice.oneNote (nMi));
    Voice v2 = Voice.concatenated (v1,Voice.oneNote (nRe));
    System.out.println("v1 = " + v2);
    System.out.println("otherNotes(v1) = " + v2.otherNotes());
  }
}
```

Can you spot where I made us of the implicit invokation of the `toString` canonical functions?

Compiling and running this code gives us the expected result:

```
> javac VoiceTester.java
> java VoiceTester
v1 = fa mi re
otherNotes(v1) = mi re
```

As I said, this is very simple-minded testing. Just running a test or two to make sure we can create simple voices. In a few lectures, we will be considering testing much more carefully.

Static Variables

Before stopping today, let me talk about another feature of the Java object system which comes in useful, and is related to static methods.

Consider the problem of keeping track of how many voice objects are created during the course of an execution. In fact, suppose we augment the interface with an operation `totalVoices`:

```
public static int totalVoices ();
```

We need the method to be static, because it needs to be invokable even when no objects are defined (and in that case return the value 0, of course).

But where do we store the information? It cannot be in any object, because it needs to hold a value (0) even when there are no objects around. The answer, which may be obvious, is to somehow store the information in a field that is kept in the class itself, transcending all the instances of that class. This field, because it exists in the class itself instead of objects of that class, will be called a *static field*, static variable, or *class variable*, variously.

We add the following field to the `Voice` class:

```
// Class variable holding total # of voices created
private static integer totalVoices = 0;
```

Note that we provide an initial value for the field.

The corresponding accessor is similarly simple:

```
// Accessor for total number of voices created
public static integer totalVoices() {
  return totalVoices;
}
```

Finally, we need to update the class variable whenever a `Voice` object is created. The variable can be updated either in each of the creators, or we can just do it once in the constructor:

```
// Voice constructor, made private, and updating the class variable
private Voice () {
  totalVoices++;
}
```

Note that, in contrast with everything else we have seen until now, this variable is *mutated* during execution. There are some reasons why doing this for class variables is not as problematic as for fields in normal objects. But for now, just be aware that updating a class variable is a mutation.