Object-Oriented Design — **Lecture 5**
CSU 370 Fall 2007 (Pucella) — **Friday, Sep 21, 2007**

The Java Type System (continued)

The `Object` Class

All classes subclass the `Object` class. (By default, this is the superclass used when you do not specify an `extends` class in your class definition.)

The `Object` class is essentially defined as follows.

```
public class Object {
  public String toString () {...}
  public boolean equals (Object obj) {...}
  public int hashCode () {...}
  public Object clone () {...}
}
```

These methods the so-called canonical methods I mentioned earlier.

Thus, every object defines a `toString()` method, etc, etc.

Because of the way Java works (we will see in a bunch of lectures), when a canonical method is not defined in a class you create, a default implementation is provided.

The default for `toString()` is something like "build a unique name from the name of the class and a hash code of the object" e.g. `Voice@a1234`. You generally want to replace this by something more informative, as we already saw in the `Voice` example a few lectures ago.

Let's look at the other canonical methods

Object Equality

The built-in operator `==` is used to check for equality. Now, for primitive types, `==` behaves like you would expect, that is, `1==1` and `!(1==2)`. Similarly, `true==true`, but `!(true==false)`.

But what happens with objects? `obj1==obj2` returns true exactly when two objects are the *same* actual object. In other words, `==` compares object identity.

For example,

```
Voice obj = emptyVoice();
```

```
  Voice obj2 = obj;
  obj2 == obj  ---> true
```

But:

```
  Voice obj = emptyVoice();
  Voice obj2 = emptyVoice();
  obj2 == obj   ---> false!
```

Although `obj` and `obj2` "look the same", they are different objects. (Each invocation of `emptyVoice()` performs a `new`, which creates a different object every time.)

Object identity is useful, but it is rarely what we want. In particular, I may want to say that two voices are equal if they contain the same sequence of notes. (This is a little bit like in set theory, where two sets are considered equal if they have the same elements, or considering two lists equals if they have the same elements in the same order.) This goes back to the principle of indistinguishability, which can be paraphrased here as: if two objects behave the same (i.e., yield the same observations) no matter the situation, then they should be considered equal.

The `equals()` method is used to represent this kind of equality. Whenever you define a new class, you will want to define an `equals()` method that compares another method with it for equality. Now, you get to choose what equality means for your new class. As I said, by default, if you do not supply an `equals()` method, a default is provided, which in this case is simply something like:

```
  public boolean equals(Object obj) {
    return this==obj;
  }
```

and therefore the default `equals()` method is just identity checking. Again, generally not what we want.

For voices, we want to implement something more useful. For instance, a reasonable notion of equality for voices is to take two voices to be equal if they have the same sequence of notes. Assume that the `Note` class has an implementation of `equals()` that checks when two notes are the same note.

```
  public boolean equals (Object obj) {
    Voice voice;
    if (obj instanceOf Voice) {  // gotta be at least a Voice to be equal
      voice = (Voice) obj;  // cast to a Voice
      if (this.isEmpty() && voice.isEmpty())
        return true;
      else
        return (this.firstNote().equals (voice.firstNote()) &&
                this.otherNotes().equals (voice.otherNotes()));
    }
    return false;
  }
```

(This should really go in the signature of the ADT I am writing; cf Homework 2)

Note that in order to come up with this code I made an assumption on the behavior of the `Note` class – that its `equals()` method indeed checks that two notes are the same note (not necessarily the same object though). Here, when writing the `Voice` class, I am acting as a client of the `Note` class, and relying on its specification.

Now, in order for `equals()` to truly behave like an equality, it has to satisfy the main properties of equality. Does anybody know what they are? What are the characteristics of equality?

- **Reflexivity**: `obj1.equals(obj1)=true`
- **Symmetry**: if `obj1.equals(obj2)==true`, then `obj2.equals(obj1)==true`
- **Transitivity**: if `obj1.equals(obj2)==true` and `obj2.equals(obj3)==true`, then `obj1.equals(obj3)`

These are the three properties that `equals()` must satisfy in order for it to behave like a "good" equality method. An additional property that can be thought of as following from the above properties but that is worth mentioning explicitly, is that

- `obj.equals(null)` is always false

Now, Java does not enforce any of those properties! It would be cool if it did, and in fact, it can be considered a nontrivial research project to figure out how to get the system to analyze your code to make sure the above is true. (Because, after all, note that you can write absolutely anything in the `equals()` method... so you need to be able to check properties of some arbitrary code — in fact, you can prove it is impossible to get, say, Eclipse, or any compiler to tell you the answer. Stick around for theory of computation to see why that is.)

It is an implicit behavioral specification that `equals()` satisfies the four properties above.

Hash codes

The last canonical method we look at is `hashcode()`. Intuitively, the hash code of an object is an integer representation of the object, that serves to identify it. The fact that an hash code is an integer makes it useful for data structures such as hash tables.

Suppose you wanted to implement a data structure to represents sets of objects. The main operations you want to perform on sets is adding and removing objects from the set, and checking whether an object is in the set. The naive approach is to use a list, but of course, checking membership in a list is proportional to the size of the list, making the operation expensive when sets become large. A more efficient implement is to use a hash table. A hash table is just an array of some size n, and each cell in the array is a list of objects. To insert an object in the hash table, you convert the object into an integer (this is what the hash code is used for), convert that integer into an integer i between 0 and $n-1$ using modular arithmetic (e.g., if $n = 100$, then 23440 is 40 (mod 100)) and use i as an index into the array. You attach the object at the beginning of the list at position i. To check for membership of an object, you again compute the hash code of the object, turn it into an integer i between 0 and $n-1$ using modular arithmetic, and look for the object in the list at index i. The hope is that the lists in each cell of the array are much shorter than an overall list of objects would be.

In order for the above to work, of course, we need some restrictions on what makes a good hash code. In particular, let's look again at hash tables. Generally, we will look for the object in the set using the object's `equals()` method — after all, we generally are interested in an object that is indistinguishable in the set, not for that exact same object.

This means that two equal objects must have the same hash code, to ensure that two equal objects end up in the same cell.[1] Thus, two equal objects must have the same hash code. Formally:

- For all objects `obj1` and `obj2`, if `obj1.equals(obj2)` then `obj1.hashCode() == obj2.hashCode()`.

Generally, hash codes are computed from data local to the object (for instance, the value of its fields). Another property of the `hashCode` that is a little bit more difficult to formalize is that the returned hash codes should "spread out" somehow; given two unequal objects of the same class, their hash codes should be "different enough". To see why we want something like that, suppose an extreme case, that `hashCode()` returns always value 0. (Convince yourself that this is okay, that is, it satisfies the property given in the bullet above!) What happens in the hash table example above? Similarly, suppose that `hashCode()` always returns either 0 or 1. What happens then?

We will see more uses of hash codes when we look at the Java Collections framework later in the course.

[1]Try to think in the above example of a hash table what would happen if two equal objects have hash codes that end up being different mod n.

Primitive Types and Corresponding Class Types

Primitive types are not class types. In particular, values of type int are not objects. That's a bit of a pain. This means that not everything is an object, and this introduces some heterogeneity in Java.

In particular, every object has a `toString()` method that can be used to get a string representation of the object. However, you cannot (a priori...) do `v.toString()` if `v` is an integer variable. An integer is not an object. This is especially problematic in some of the data structures defined in the Java Collections framework, because they are structures parameterized over a class type; for instance, we may have a queue data structure parameterized by a class type, so that one can construct a queue to store `Voice` objects, or a queue to store `Person` objects, etc. A bit similar to array, where `Person`[] constructs an array of `Person` objects. Now, we can use `int[]` because Java treats arrays specially, but it turns out we cannot construct a queue holding integers, because integers are not objects.

Bummer. Well, to restore some homogeneity, Java replicates the primitive types as class types. More precisely, for every primitive type, there is a corresponding wrapper class type. For example, corresponding to the `int` type, there is a wrapper class `Integer`. An instance of `Integer` is constructed by invoking the constructor with an integer. Thus,

```
  Integer t = new Integer (3);
```

This constructs a *boxed* integer 3. (The terminology boxed is standard; it is meant to convey the image of putting the integer 3 into a box, here an object.) The class `Integer` defines all the canonical methods, so that we can write `t.toString()` to obtain the string `"3"`, and also some methods to access the boxed integer, such as `t.intValue()`, which returns the integer stored in the boxed integer `t` as an `int`.

This moving back and forth between the boxed and the unboxed representation is so useful that it is actually performed implicitly by Java. Therefore, the following code:

```
  int i;
  Integer t = 3;
  i = t;
```

is implicitly treated as:

```
  int i;
  Integer t = new Integer (3);
  i = t.intValue();
```

This implicit boxing and unboxing can also happen at method calls, of course.

Similar wrapper classes exist for all the other primitive types. Please refer to the documentation.

Enum Types

The last bit of Java trivia I want to cover is enumeration classes.

Suppose you want to modify the `Person` class so that it also stored the sex of the person in question. No problem, we add a new field to store the field, add a new argument to the class constructor, and add a new public method to return the sex of the person. But how do we represent the sex of a person?

The cleanest way is to define new constants to represent `Male` and `FEMALE`. And these will be constants of type `Sex`, so that we do not confuse them with other constants that may exist in the system.

The syntax is as follows:

```
  public enum Sex { MALE, FEMALE };
```

(By convention, constants are all caps.) Now we can add a field and methods to the `Person` class:

```
  Sex s;

  public Sex getSex() {
    return this.s;
  }
```

We can compare values of type `Sex`, that is, we can check if `obj.getSex()==Sex.MALE`. (Note the use of the enum name qualifier.)

There is actually nothing mysterious happening here. `Sex` is just a class, and `MALE` and `FEMALE` are just objects of the `Sex` class constructed automatically; moreover, and this is the key point, they are the *only* objects of the `Sex` class that can ever be created.

You can think of the above enum declaration as an abbreviation for the more verbose but (roughly) equivalent code:

```
  public class Sex {
    private Sex() {}
    public static final Sex MALE = new Sex();
    public static final Sex FEMALE = new Sex();
  }
```

Note that the constructor is private, meaning that no one from outside the class can create new objects of the class. And there are only two objects created by the class, stored in the static fields `MALE` and `FEMALE`. (Do not worry about the `final` annotation on the methods; we will see what it means a bit later.)