Equality for Abstract Data Types

Every language has mechanisms for comparing values for equality, but it is often not the kind of equality you want. In Java, for instance, 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)`, where `!` is negation. 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,

```
Drawing obj1 = Drawing.empty();
Drawing obj2 = obj;
obj1 == obj2  ---> true
```

But:

```
Drawing obj1 = Drawing.empty();
Drawing obj2 = Drawing.empty();
obj1 == obj2   ---> false!
```

Although `obj1` and `obj2` "look the same", they are different objects. (Each invocation of `Drawing.empty()` invokes `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 drawings are equal if they contain the same sequence of lines. (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. Note that the observations we can make on drawings rely on the lines in a drawing being ordered.

For all the ADTs we are going to design, we are going to have an equality operation, capturing whatever notion of equality we deem reasonable and useful for values of the ADT, following the principle of indistinguishability. We are going to call the operation `equals()`, partly because that's a reasonable name, and partly because that's a name that Java knows about.

Java has a certain number of methods (called canonical methods) that it requires every class to implement. In fact, if you don't implement explicitly one of those canonical methods, the system assumes a default implenentation. One of those methods is the `equals()` method.[1]

The signature of the equality operation will be as follows:

```
ACCESSORS   boolean equals (Object);
```

Again, this is just convenient because it is the shape that Java expects the `equals()` method to have. (Were we to implement ADTs for other languages, we could be more flexible.) First, let's worry about the `Object` argument type. We'll come back to it later, but for the time being, just think of it as a way to indicate that `equals` can take any kind of argument as input.

What we need next is a specification for `equals`. Of course, this depends on the ADT we have. So what would be the specification of `equals` for drawings? As we said above, we want to consider two drawings equal when they have the same lines in them, in the same order. Comparing a drawing with something that is not a drawing should give us false. So let's write the specification that gives us that. If we follow the recipe outlined a couple of lectures ago, we need to say how `equals` interacts with all the creators.

```
empty().equals(obj)
  = true   if obj is a drawing && obj.isEmpty()=true
  = false  otherwise

oneLine(l).equals(obj)
  = true   if obj is a drawing && obj.restLines().isEmpty() = true
           && obj.firstLine().equals(l)
  = false  otherwise

merge(d1,d2).equals(obj)
  = true   if obj is a drawing
           && (d1.isEmpty()=d2.isEmpty()=obj.isEmpty()=true
                || (d1.isEmpty()=true and d2.equals(obj))
                || (d1.firstLine().equals(obj.firstLine()) = true
                    && merge(d1.restLines(),d2).
                                    equals(obj.restLines()) = true))
  = false  otherwise
```

(I am using Java's `&&` for AND and `||` for OR. I also assume that the `Line` ADT has an `equals()` operation that checks when two lines are equal.) A bit of a mess, but it works.

[1] By default, Java takes the `equals` method to just check for object equality, `==`, again, generally not what we want.

Actually, it turns out that we can simplify the above, and replace these three equations by a single equation. I don't recommend you necessarily do that at the beginning, at least not until you understand what is going on well. But convince yourself that you can replace the above three equations by the following simpler equation:

```
// alternative algebraic specification for equals

d.equals(obj)
  = true  if d is a drawing
          and (d.isEmpty()=obj.isEmpty()=true
               or (d.firstLine().equals(obj.firstLine())=true
                   && d.restLines().equals(obj.restLines())=true))
  = false  otherwise
```

This specification is much easier to implement. In fact, it already *is* an implementation. To implement it in Java, we first need to know how to check whether an object is a drawing. We can do it using Java's `instanceof` operator, which checks if an object is an instance of some class. Here is an implementation of `equals` for `Drawing`, more verbose than it may need to be, but at least it's clear:

```
public boolean equals (Object obj) {
  Drawing drawing;
  if (obj instanceof Drawing) {  // is obj a drawing?
    drawing = (Drawing) obj;     // cast to a Drawing
    if (this.isEmpty() && drawing.isEmpty())
      return true;
    else
      return (this.firstLine().equals(drawing.firstLine()) &&
              this.restLines().equals(drawing.restLines()));
  }
  return false;
}
```

Now, in order for `equals()` to truly behave like an equality, it has to satisfy the main properties of equality. 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)=true`

These are the three properties that `equals()` must satisfy in order for it to behave like a "good" equality method. An additional property follows from the above properties that is worth mentioning explicitly:

- `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 three properties above.

Hash Codes [Not Covered in Class]

Going hand in hand with the `equals` method in Java (this is Java specific...), is the canonical method `hashCode()`. Let me make a little detour here. 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.[2] Thus, two equal objects must have the same hash code.

[2]Try to think in the above example of a hash table what would happen if two equal objects have hash

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.

Java Programs [Not Covered in Class]

Let's write a program testing a little bit our implementation of the `Drawing` class. This will lead to an actual discussion on testing soon, but for the time being, this is just a way to re-introduce the basics of coming up with an actual Java program.

To test `Drawing`, I need a nice way to print out a representation of a drawing. So let's add an operation to the signature of the `Drawing` ADT that lets us get a string representation of a drawing for printing and debugging purposes. (We need to add it to the signature because we want the function to be available, and I told you that only operations in the signature should be available.)

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

```
public String toString ();

empty().toString() = ""
oneLine(l).toString() = l.toString()
merge(d1,d2).toString() = d1.toString() + " " + d2.toString()
```

This specification assumes that lines have a `toString` operation as well.

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

codes that end up being different mod n.

```
// Canonical method to print lines in a drawing

public String toString () {
  if (this.isEmpty())
    return "";
  return (this.firstLine().toString() + " "
          + this.restLines().toString());
}
```

Again, the name `toString` I used for this method is not random. It is a canonical method that Java understands implicitly. Here's how `toString()` is special: whenever you use an object in a context 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.[3]

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

```
// A placeholder implementation of a line

class Line {

  // The name of the line
  private String name;

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

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

  // Canonical method for printing name of a line
  public String toString () {
```

[3]Because it is a canonical method, there is a default implementation for `toString()`, something like "build a unique name from the name of the class and a hash code of the object" e.g. `Drawing@a1234`. Not the most informative.

```
    return this.name;
  }
}
```

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

```
// Simple-minded tester for the Drawing class

class DrawingTester {

  // The main testing function

  static public void main (String[] args) {
    Line l1 = Line.named("L1");
    Line l2 = Line.named("L2");
    Line l3 = Line.named("L3");
    Line l4 = Line.named("L4");
    Drawing d1 = Drawing.merge(Drawing.oneLine(l1),
                               Drawing.oneLine(l2));
    Drawing d2 = Drawing.merge(d1,Drawing.oneLine(l3));
    System.out.println("drawing = " + d2);
    System.out.println("restLines(drawing) = " + d2.restLines());
  }
}
```

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 DrawingTester.java
> java DrawingTester
drawing = L1 L2 L3
restLines(drawing) = L2 L3
```

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