4 Equality for Abstract Data Types

Every language has mechanisms for comparing values for equality, but it is rarely the kind of equality you want. In Java or C++, 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` is true, and `1==2` is false. Similarly, `true==true` is true, but `true==false` is false.

But what happens with objects? `obj1==obj2` is true exactly when two objects are the *same* actual object. When an object is created, it gets allocated somewhere in memory. We consider two objects to be the same actual object if they live at the same address in memory. In other words, `==` checks for *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()` calls `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. The following principle can be used to help you devise a suitable notion of equality for an ADT for which you've already determined the creators and accessors, and can be paraphrased as follows: if two objects behave the same (i.e., yield the same observations) no matter the situation, then they should be taken to be equal considered equal. More formally:

> **The Principle of Indistinguishability**: If two values `v` and `w` of an ADT are such that for every accessor `m` in the ADT applying `m` to `v` and `w` with the same arguments yields results that are equal, then we should take `v.equals(w)` to be `true`.

If you think about it, the converse also should be true: if two values are considered equal, then they should yield the same answers to any accessor. So really equality ought to be determined by the results of invoking accessors on values of the ADT. For drawings, note that the observations we can make on drawings are based on the fact that the lines in a drawing are ordered.

For every ADT 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()`.

What should be the signature of the `equals` operation? Certainly, `equals` should take a value to compare the implicit argument to, and should return a Boolean value. But what should the type of the argument value be? Two choices are usually suggested: either take the type of the argument to be the type of the ADT, or allow any type of argument to be compared with a value of the ADT. For a variety of reasons, not the least of which being that that's how Java asks you to implement equality, we shall take the latter approach. Therefore, for drawings, the signature of the equality operation is:

```
ACCESSOR   boolean equals (Object)
```

We'll come back to the `Object` argument type later in the course, but for the time being, just think of it as a way to indicate that `equals` can take any kind of object as argumnent, not just drawings, or whatever the ADT is. (Clearly, if we compare a value that is not a value of the ADT to a value of the ADT for equality, the result should be false.)

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, at the same positions, in the same order. If we follow the approach I described in Lecture 2, we need to give equations describing how `equals` interacts with every creator.

```
empty().equals(d)
  = true     if d is a drawing and d.isEmpty()=true
  = false    otherwise

oneLine(l).equals(d)
  = true   if d is a drawing and d.isEmpty()=false
              and d.firstLine().equals(l)=true
              and d.restDrawing().isEmpty()=true
  = false  otherwise

merge(d1,d2).equals(d)
  = true   if d is a drawing and d.isEmpty()=true
              and d1.equals(m)=true and d2.isEmpty()=true
```

```
   = true   if d is a drawing and d.isEmpty()=false
               and (d1.isEmpty()=false or d2.isEmpty()=false)
               and d1.firstLine().equals(d.firstLine())=true
               and merge(d1.restDrawing(),d2).equals(d.restDrawing())=true
   = false  otherwise
```

For this specification, I am assuming that the Line ADT has an `equals()` operation that checks whether two lines are equal.

This specification is a bit of a mess, but it works. And it follows the pattern I gave for deriving specifications. It turns out that we can simplify the above specification somewhat, 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 algebraic specifications well. But convince yourself that you can replace the above three equations by the following simpler equation:

```
d1.equals(d2)
   = true    if d2 is a drawing
                and d1.isEmpty()=true and 22.isEmpty()=true
   = true    if d2 is a drawing
                and d1.isEmpty()=false and d2.isEmpty()=false
                and d1.firstLine().equals(d2.firstLine())=true
                and d1.restDrawing().equals(d2.restDrawing())=true
  = false    otherwise
```

This specification is much easier to implement, at least given what we know now. (Later, we will see that the original specification for `equals` can be easier to implement.) In fact, it already *is* an implementation, as we will see below.

Now, in order for `equals()` to truly behave like an equality, it has to satisfy the three main properties 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. Programmers will often unconsciously take as a given that `equals` satisfies the above properties. It is an implicit specification that `equals()` satisfies the three properties above. Because of this, we will require that `equals()` always satisfies these properties. As we shall see later on, it is actually almost impossible to get `equals()` to

satisfy them in Java. In particular, most implementations of equality will fail to satisfy symmetry. That may surprise you, and we'll come back to that point later in the course.

Now, programming languages (Java included) do not enforce any of those properties! It would be cool if they did, but it's a very hard problem. Think about it: you can write absolutely anything in an `equals()` method... so you need to be able to check properties of arbitrary code — you can prove it is impossible to get, say, Eclipse or any compiler to tell you the answer. Stick around for a course on the theory of computation to see why that is.

4.0.1 Implementing Equality in Java

We can easily implement the above `equals` operation in the `Drawing` class from last time:

```
public boolean equals (Object obj) {
  // first check if obj is a Drawing
  if (obj instanceof Drawing) {
    // it is, so let's tell the type system
    Drawing dobj = (Drawing) obj;
    if (this.isEmpty() && dobj.isEmpty())
      return true;
    if (!(this.isEmpty()) && !(dobj.isEmpty()) &&
      this.firstLine().equals(dobj.firstLine()) &&
      this.restDrawing().equals(dobj.restLines()))
      return true;
    return false;
  }
  return false;
}
```

The only slight nastiness here is how to check that an arbitrary object is actually a drawing, as the specification requires. It turns out there is no nice way to do that. We shall use a special operator in the language, `instanceof`, which checks that an object is an instance of a specific class.[1] After the check that `obj` is actually a drawing, we need to tell the type checker, which is actually not smart enough to figure it out by itself. That's what the *cast* on the fifth line does: `Drawing dobj = (Drawing) obj` tells the compiler that argument `obj`, when accessed through the name `dobj`, is an instance of `Drawing`. We'll say more about casts later in the course. The rest of the implementation just follows the single-equation specification for `equals` we gave above.

[1] *This is the only place where I will ever use the* `instanceof` *operator*, and I strongly suggest you follow that advice as well. I won't go as far as saying that `instanceof` is evil, but it's darn close. It is exceedingly difficult to maintain code that uses `instanceof`.

Because equality is so important, Java requires that every class implements an `equals` method. If one is not given, then a default is provided: the `equals` method will just check for object identity `==`, again, generally not what we want.[2]

Because we've changed the default `equals` method of a class by implementing an `equals` method for our ADT, Java forces us to deal with another little detail. Going hand in hand with the `equals` method in Java 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.[3] Thus, two equal objects must have the same hash code. Formally:

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

[2]Java has a certain number of methods (called canonical methods) that it requires every class to implement. If you don't implement explicitly one of those canonical methods, the system assumes a default implenentation. Methods `equals()`, `hashCode()`, and `toString()` are the canonical methods.

[3]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.

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?

Hash codes are extensively used in the Java Collections framework.

For drawings, we can therefore add the following operation to the signature of the ADT:

```
ACCESSOR   int hashCode ()
```

with the following specification, which assumes that the Line ADT includes a `hashCode()` operation as well:

```
empty().hashCode() = 0

oneLine(l).hashCode() = l.hashCode()

merge(d1,d2).hashCode() = d1.hashCode() + d2.hashCode()
```

Exercise: *Provide an implementation for `hashCode()` that implements the above specification.*