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` and `!(1==2)`, where `!` is negation. Similarly, `true==true`, but `!(true==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,

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

But:

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

Although `obj1` and `obj2` "look the same", they are different objects. Each invocation of `Map.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 maps are equal if they contain the same sequence of artifacts. 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 maps are based on the fact that the artifacts in a map 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()`.

The signature of the equality operation on maps will be:

```
ACCESSOR   boolean equals (Map);
```

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 maps? As we said above, we want to consider two maps equal when they have the same artifacts in them, at the same positions, in the same order. If we follow the recipe outlined a couple of lectures ago, we need to say how `equals` interacts with all the creators.

```
  empty().equals(m) = m.isEmpty()

  singleArtifact(a,i,j).equals(m)
    = true   if m.firstArtifact().equals(a)=true
                and m.firstXPos()=i
                and m.firstYPos()=j
                and m.restArtifacts().isEmpty()=true
    = false  otherwise

  merge(m1,m2).equals(m)
    = true   if m1.isEmpty()=true and m2.equals(m)=true
    = true   if m1.isEmpty()=false
                and m1.firstArtifact().equals(m.firstArtifact())=true
                and m1.firstXPos()=m.firstXPos()
                and m1.firstYPos()=m.firstYPos()
                and merge(m1.restArtifacts(),m2).equals(m.restArtifacts())=
                                                                       true
    = false  otherwise
```

For this specification, I am assuming that the `Artifact` ADT has an `equals()` operation that checks when two artifacts are equal.

This specification is a bit of a mess, but it works. And it follows the "recipe" we 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:

```
m1.equals(m2)
   = true    if m1.isEmpty()=true and m2.isEmpty()=true
   = true    if m1.isEmpty()=false and m2.isEmpty()=false
                and m1.firstArtifact().equals(m2.firstArtifact())=true
                and m1.firstXPos()=m2.firstXPos()
```

```
                 and m1.firstYPos()=m2.firstYPos()
                 and m1.restArtifacts().equals(m2.restArtifacts())=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 not actually trivial to get `equals()` to satisfies them in Java. In particular, the implementation most directly derived from the specification of `equals` I gave above for maps will usually fail to satisfy symmetry. Again, we'll come back to that point later.

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.

Implementing Equality in Java

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

```
public boolean equals (Map m) {
  if (this.isEmpty() && m.isEmpty())
    return true;
  if (!(this.isEmpty()) && !(m.isEmpty()) &&
      this.firstArtifact().equals(m.firstArtifact()) &&
      this.firstXPos() == m.firstXPos() &&
      this.firstYPos() == m.firstYPos() &&
      this.restArtifacts().equals(m.restArtifacts()))
    return true;
  return false;
}
```

Because we are working in Java, though, we have to deal with a slight nastiness of the language. In particular, 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.[1]

No big deal. We have an `equals` method in the `Map` class now. The problem is that Java requires the `equals` method to have the following signature:

```
  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 maps (or whatever the class is about).

So Java requires us to have an `equals` method with the above signature, while we have one with a different signature. Let's think for a second what we should do if we compare a map against something that is not a map for equality. A reasonable answer is to just say that the comparison yields false, the two cannot be equal. This gives us a way out. Here are the implementation of the two equals methods:

```
public boolean equals (Map m) {
  if (this.isEmpty() && m.isEmpty())
    return true;
  if (!(this.isEmpty()) && !(m.isEmpty()) &&
      this.firstArtifact().equals(m.firstArtifact()) &&
      this.firstXPos() == m.firstXPos() &&
      this.firstYPos() == m.firstYPos() &&
      this.restArtifacts().equals(m.restArtifacts()))
    return true;
  return false;
}

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

[1]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.

Jave allows us to have multiple methods with the same name in a class, as long as their argument types are different. Operationally, when you invoke the method `equals`, Java uses the type of the argument to decide which method to invoke—it uses the one with the most specific type that can deal with the supplied argument. (More details when we talk about subtyping.) So, if we give a map to `equals`, the first `equals` is used, and the result depends on whether the two maps are equals; and if we give a non-map to `equals`, the second `equals` is used, and the result is false, as expected. Beautiful. Note that we need to add `boolean equals (Object)` to the signature of the ADT (and give it a specification), because it is a method that is exported by the class.

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.[2] 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()`.

[2]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.

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.