ADT Implementation in Java

As I said in my introductory lecture, we will be programming in Java in this course. In part, this is because Java is a reasonable language, and also because you already have seen some Java in CSU 213 (CS 2510). 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 values of 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.

Some remarks on the language: Java is a class-based object-oriented language. "Object-oriented" means that (almost) every value in the language is an object. "Class-based" means that objects are created from classes. A class is essentially a template, a description of how objects of the class are created.

It helps to think of a program has having two parts. One part, the part that correspond to the source code, is static. (Static means non-moving, which we take to mean non-executing). It represents what information about the program we have before anything executes. In Java, the only thing we know before a program executes are what classes are defined. Thus, the classes are static. Classes exist, in some sense, even before programs start executing. The other part is the dynamic part, which corresponds to program execution. During execution, instances of the classes, that is, objects, get created, updated, destroyed. Thus, objects are dynamic.

You should probably be aware that not every object-oriented language is class-based. Self, for example, has no concept of class, but still has objects. Java is also statically typed — types are associated with variables, and before a program is run those types are checked, to make sure that values of the right kind are stored in variables, or passed to methods.

Not every object-oriented language is statically typed. Smalltalk, for example, checks types dynamically, like Scheme does. We'll talk more about types in a few lectures.

Object-Oriented Signatures and Specifications

To help us devise implementation for ADTs in object-oriented langauges, let us consider a slightly different way of writing signatures and specifications.

The plan today is to give a quick and dirty implementation of the Map ADT, to illustrate the basic uses of Java, and also to introduce the conventions we will use. Recall the signature we had, augmented with a few new operations that I forgot about last time (`firstXPos` and `firstYPos`, returning the x- and y-coordinates of the first artifact of a map), and modified so that `singleArtifact` takes integers instead of natural numbers.

```
CREATORS    empty          :  -> Map
            singleArtifact :  Artifact int int -> Map
            merge          :  Map Map -> Map

ACCESSORS   isEmpty       :  Map -> boolean
            firstArtifact :  Map -> Artifact
            firstXPos     :  Map -> int
            firstYPos     :  Map -> int
            restArtifacts :  Map -> Map
```

Exercise: *I did not give a specification for* `firstXPos` *and* `firstYPos`*. Write them.*

An object-oriented signature, to a first approximation, consider that accessors (not creators), which must take at least one argument of the type of the ADT[1] take that value on which they act as an *implicit* argument, as opposed to an *explicit* argument that appears in the argument list. Thus, the Map object-oriented signature looks as follows, using a Java-like syntax:

```
CREATORS    Map empty ();
            Map singleArtifact (Artifact, int, int);
            Map merge (Map, Map);

ACCESSORS   boolean isEmpty ();
            Artifact firstArtifact ();
            int firstXPos ();
            int firstYPos ();
            Map restArtifacts ();
```

[1] otherwise, I would claim that such an accessor has no business being part of the ADT.

Let's change how those operations are invoked, because we need to understand where the implicit argument to the accessors is coming from. Creators are invoked as before, e.g., `singleArtifact(a,i,j)`. Accessors, on the other hand, are invoked on an expression yielding a `Map` value, e.g., `v.firstArtifact()`, where `v` is a `Map` value, or `merge(m1,dm).firstArtifact()`.

We can now similarly convert the specification using the above invocation forms:

```
empty().isEmpty() = true
singleArtifact(a,i,j).isEmpty() = false
merge(m1,m2).isEmpty()
   = true     if m1.isEmpty()=true and m2.isEmpty()=true
   = false    otherwise

singleArtifact(a,i,j).firstArtifact() = a
merge(m1,m2).firstArtifact()
   = m2.firstArtifact()      if m1.isEmpty()=true
   = m1.firstArtifact()      otherwise

singleArtifact(a,i,j).firstXPos() = i
merge(m1,m2).firstXPos()
   = m2.firstXPos()      if m1.isEmpty()=true
   = m1.firstXPos()      otherwise

singleArtifact(a,i,j).firstYPos() = j
merge(m1,m2).firstYPos()
   = m2.firstYPos()      if m1.isEmpty()=true
   = m1.firstYPos()      otherwise

singleArtifact(a,i,j).restArtifacts() = empty()
merge(m1,m2).restArtifacts()
   = m2.restArtifacts()               if m1.isEmpty()=true
   = merge(m1.restArtifacts(),m2)     otherwise
```

This presentation of signatures and specifications is easier to implement in an object-oriented language, as we now confirm.

Implementation of Map ADT

Let's implement the above signature, then. The one decision we have to make, at this point, is how to represent maps. We did not have to worry about this when writing the specification, because there, the only thing we cared about is how maps behave, that is, how their operations interact. But now that we have to write code to actually create and

manipulate maps, we need to choose a representation. Later, we will see that there is a very natural representation that falls out of various principles of object-orientation. But for now, we will focus on a naive and somewhat ugly representation, partly for the purpose of contrasting it with something better later on. The representation we choose is to take a map to be a linked list of artifacts (and their position).

With this in mind, we define a class `Map` in a file `Map.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 Map ADT given in lecture
public class Map {

  ...

}
```

Let's fill in the body of this class.

There are several rules that we will follow when writing classes in this course. Six, to be precise, to be introduced throughout this example. Here is the first one:

(1) The only methods in the class that we should be able to invoke are those given 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 are specified using the `public` keyword.

We decided to represent maps as lists of artifacts. We have a few ways of doing this. Here is one. We define fields in a map holding the first artifact of the map, its position, and the rest of the map, as well as a flag recording whether the map is empty.

```
private boolean isempty;        // whether the map is empty
private Artifact currArtifact;  // the first artifact of the map
private int currXPos;           // x-position of the first artifact
private int currYPos;           // y-position of the first artifact
private Map rest;               // the rest of the map
```

The second rule explains some of the choices:

(2) All fields are private.

Fields are not part of the signature, so the spirit of rule (1) says that they should be private. This is not a big restrictions, as we can always use methods (if the signature tells us to) to read and update fields. Mostly, this is to make sure that the rest of the code does not depend on there being a particular field in the object, so that we can, for instance, change the representation of an ADT without worrying about breaking code elsewhere in our program.

From CSU 213 at least, you know that every Java class requires a *constructor*, the method invoked when the `new` operator is used to create an instance of the class. The Java constructor for the `Map` class is simple:

```
private Map (boolean e, Artifact a, int x, int y, Map r) {
  isempty = e;
  currArtifact = a;
  currXPos = x;
  currYPos = y;
  rest = r;
}
```

The only thing the constructor does is initialize the fields. That's what a constructor will *always* do:

(3) The class constructor takes are arguments initial values for all the fields and sets them.

More interestingly, note that the constructor is private. This is in keeping with rule (1), because the constructor is not listed in the signature of the ADT, and never will be:

(4) The class constructor is private.

But that's not a big problem, because we have creators, which are used to create new instances. The creators, being defined inside the class, will be able to invoke the constructor.

So let's look at the creators. Creators, the way we will use them, 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*.

```
// Creator for an empty map
public static Map empty () {
  return new Map (true, null, 0, 0, null);
}

// Creator for a map with a single artifact
public static Map singleArtifact (Artifact a, int i, int j) {
  return new Map (false, a, i, j, Map.empty());
}
```

```
// Creator for a merged map
public static Map merge (Map m1, Map m2) {
  if (m1.isEmpty())
    return m2;
  return new Map (false, m1.firstArtifact(),
                  m1.firstXPos(), m1.firstYPos(),
                  Map.merge (m1.restArtifacts(),m2));
}
```

This should all be fairly clear, except perhaps the `static` annotation on the creators. This is because we have painted ourselves in a corner, in a sense. Consider this. We would like to write a creator such as `empty` as a public method that calls the constructor. But how will we able to call the `empty` method? Recall that in general, to invoke a method, we need an instance of the 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. Bummer.

The key is to somehow make sure that the creator methods are available even when *no* instance of the class exist. That's what the `static` annotation does. It says that the method does not require an instance of the class to be invoked. There is no implicit object passed to the method. It acts, essentially, just like a normal function in other language. (Why calling it static? Recall the discussion at the beginning of this lecture, and try to figure it out.)

How do you use static methods? Let's examine more carefully how to call methods:

- A non-static method `m` in class `T` is called with arguments `a` and `b` (say) by writing:

  ```
  obj.m(a,b)
  ```

  where `obj` is some expression denoting an object of class `T`. In the body of method `m`, the keyword `this` denotes the same object as `obj`.

- A static method `m` in class `T` is called with arguments `a` and `b` (say) by writing:

  ```
  T.m(a,b)
  ```

  and the keyword `this` cannot be used in the body of method `m`.[2]

Following this discussion, we add the following rule to our list:

[2]There are 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.

(5) Creators are static methods.

Finally, we can implement the accessors, which in this case simply query the appropriate fields of a map:

```
// Accessor for checking if a map is empty
public boolean isEmpty () {
  return (this.isempty);
}

// Accessor for the first artifact of a map
public Artifact firstArtifact () {
  if (this.isEmpty())
    throw new Error ("applying firstArtifact to an empty map");
  return (this.currArtifact);
}

// Accessor for the x-coordinate of the first artifact of a map
public int firstXPos () {
  if (this.isEmpty())
    throw new Error ("applying firstXPos to an empty map");
  return (this.currXPos);
}

// Accessor for the y-coordinate of the first artifact of a mpa
public int firstYPos () {
  if (this.isEmpty())
    throw new Error ("applying firstYPos to an empty map");
  return (this.currYPos);
}

// Accessor for the rest of the map
public Map restArtifacts () {
  if (this.isEmpty())
    throw new Error ("applying restArtifacts to an empty map");
  return (this.rest);
}
```

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

Fields are never updated. This is important enough that I will make it a rule that we will only break towards the end of the course:

(6) Fields, once initialized by the class constructor, are never updated.

In combination with rule (2) that makes every field private, this makes every instance of the class *immutable*—once created, it cannot be changed. Immutable instances have a host of advantages: the code is easier to reason about, it is easier to replace the code or debug it, etc. As we will see when we look at mutation, understanding what actually happens when a field is updated can get very tricky when a program uses all the features of Java. Because of this, and other reasons that we will return to in the course of the semester, we will restrict our attention to immutable instances.

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 `Map`, or an hypothetical `LargeMap`. Method names and variable names are capitalized but for the first letter, like `singleArtifact`, 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.