7 Code Reuse: Subclassing

A couple of lectures ago, we saw several different kind of errors, and I pointed out that some of those errors can be detected at compile-time, that is, before executing the program, and some can only be detected at run-time.

Java, for instance, tries to catch several kind of errors at compile-time. The Java type system is an algorithm (defined by a set of rules) that takes the source code of a program, and without actually running the code, will try to establish that when the code will execute it will not try to either

- invoke a primitive operation on arguments of the wrong type, or
- invoke a method on an object that does not define that method.

We call a program *safe* when it does not exhibit the above kind of errors. Therefore, the Java type system ensures that if a program type-checks, then it is safe.

Safety is pretty important. It is the first line of defense, generally, to make sure your code is correct. (A safe program is not necessarily correct, of course, since it may now behave as you want it to behave — but an unsafe program is definitely not correct.)

What I want to look at today is a way to reuse code in such a way that we can still preserve the safety of our code. Consider the following example. Take (a simplified form of) the Point ADT from your first homework:

```
public static Point create (int, int);
public int xPos ();
public int yPos ();
public Point move (int, int);
```

with spec:

```
create(x,y).xPos() = x
create(x,y).yPos() = y
create(x,y).move(dx,dy) = create(x+dx,y+dy)
```

This is easy to implement using a `Point` class in Java.

Now, suppose we write a function[1] `distance` to compute the distance between two points. Easy enough:

```
public static double distance (Point p, Point q) {
   return Math.sqrt(Math.pow(p.xPos()-q.xPos(),2.0) +
                    Math.pow(p.yPos()-q.yPos(),2.0));
}
```

Now, clearly, if we call `distance` with two instances of `Point`, then the result is safe, since the function only relies on the fact that the objects passed as arguments have a `xPos()` and a `yPos()` method. And instances of `Point` do have those methods. And indeed, the type checker does not complain if we call `distance` with two instances of `Point`.

Conversely, suppose we call `distance` with two instances of, say, `Drawing`. Then were we to execute the code we'd have a problem because a drawing does not have a method `xPos()`, so that call `p.xPos()` would fail. To prevent this, the type system will complain that calling `distance` with two instances of `Drawing` is a type error, and will keep you from executing the code in the first place.

But notice that we should be able to call `distance` using any objects that implement both `xPos()` and `yPos()`, and the result would still be safe. Let's make this a bit more precise:

Definition: *D is a subclass of C if every instance of D implements all the public methods defined in C, as well as all the public fields defined in C.*

(This is still somewhat imprecise because I should say something about the argument and return types of the methods, but I'll return to that later.)

So, if we give `distance` two instances of a subclass of `Point`, the result should still be safe, at least as the definition of safety we're using is concerned.

If D is a subclass of C, we call C a superclass of D.

Now, note that the only thing that subclassing says is that the subclass should implement all the public methods of its superclass. The implementation of those methods, however, can be utterly different — that's not a problem. They just need to be defined.

Here's an example of a subclass of `Point`. Define an ADT for colored points, the CPoint ADT, as follows:

```
public static CPoint create (int, int, Color);
public int xPos ();
public int yPos ();
public CPoint move (int, int);
public Color color ();
public CPoint newColor (Color);
```

[1]I'm generally going to call static methods *functions*, because they really do play the role of functions, not being associated with an object.

with spec:

```
create(x,y,c).xPos() = x
create(x,y,c).yPos() = y
create(x,y,c).move(dx,dy) = create(x+dx,y+dy,c)
create(x,y,c).color() = c
create(x,y,c).newColor(d) = create(x,y,d)
```

(Assume we have a Color ADT for colors – below, we'll take these to be strings for simplicity.)

Again, it is easy to implement the CPoint ADT in Java, as a class `CPoint`.

Now, every instance of `CPoint` implements a `xPos()` and a `yPos()` method, so we should be able to pass instances of `CPoint` to `distance` and have the resulting call be safe.

And in many languages with subclassing, you can do just that. Not in Java, however. If you try the above in Java, you get a type error when you call `distance` passing in instances of `CPoint`. Why?

7.1 Subclassing in Java

The above definition of subclassing is a form of subclassing known as *structural subclassing*, where subtyping is automatic as long as the right methods are defined. In Java, subclassing is not automatic, you need to point it out explicitly — this is called *nominal subclassing*.

One way to express subclassing in Java is to use an `extends` annotation on the class.[2] Let `Point` be an immutable implementation of the Point ADT. For instance:

```
public class Point {
  private int xpos;
  private int ypos;

  // this is needed − Java specific incantation
  // we'll examine this later.
  // think of this as saying: Point can now have subclasses
  protected Point () { }

  private Point (int x, int y) {
    xpos = x;
    ypos = y;
  }
```

[2]`extends` plays several roles, one of them is to define a subclassing relation between class. We'll see others.

```
  public static Point create (int x, int y) {
    return new Point(x,y);
  }

  public int xPos () { return this.xpos; }

  public int yPos () { return this.ypos; }

  public Point move (int dx, int dy) {
    return create(this.xpos + dx, this.ypos + dy);
  }
}
```

Assume that we have a class `Color` available, the details of which are completely irrelevant. Here is a possible definition of the `CPoint` class:

```
public class CPoint extends Point {
  private int xpos;
  private int ypos;
  private Color color;

  private CPoint (int x, int y, Color c) {
    xpos = x;
    ypos = y;
    color = c;
  }

  public static CPoint create (int x, int y, Color c) {
    return new CPoint (x,y,c);
  }

  public int xPos () { return this.xpos; }

  public int yPos () { return this.ypos; }

  public CPoint move (int dx, int dy) {
    return create(this.xpos + dx, this.ypos + dy, this.color);
  }

  public Color color () { return this.color; }

  public CPoint newColor (Color c) {
    return create(this.xpos, this.ypos, c);
  }
}
```

This works perfectly fine, and Java will work with this quite happily. In particular, the following code compiles:

```
public static void main (String[] argv) {

  CPoint cp1 = CPoint.create(3,4,"Red");
  CPoint cp2 = CPoint.create(5,6,"Blue");

  System.out.println("Distance cp1 <-> cp2 = " + distance(cp1,cp2));
}
```

This illustrates how subclassing enables code reuse:

Subclassing lets you reuse client code.

The `distance` function, which is client code with respect to the `Point` class, this *same piece of code* can be used to work on both points and colored points, because the class `CPoint` is a subclass of `Point`.

Now, this seems like awfully redundant code. And it is. And some of you are screaming, what about inheritance? We'll see inheritance in the coming weeks. Inheritance is something different. **Inheritance is not subclassing.** The two are somewhat related, of course, but it is possible to have subclassing without inheritance. The above code uses subclassing, and does not rely on inheritance. (It is also possible to have inheritance without subclassing, but that's less common.) Inheritance brings its own backpack of problems with it, that subclassing by itself does not have.

7.2 The Specification Design Pattern

An interesting use of subclassing is to derive an implementation from an ADT equipped with an algebraic specification.

Let me use a simple ADT that you are familiar with, lists of integers, with the following signature:

```
public static List empty ();
public static List cons (int, List);
public boolean isEmpty ();
public int first ();
public List rest ();
public String toString ();
```

and specification:

```
empty().isEmpty() = true
cons(i,s).isEmpty() = false

cons(i,s).first() = i

cons(i,s).rest() = s

empty().toString() = "<empty>"
cons(i,s).toString() = i + " " + s.toString()
```

Consider the following direct implementation of the List ADT as a linked list:

```
public class List {
  private Integer firstElement;
  private List restElements;

  private List (Integer i, List s) {
    firstElement = i;
    restElements = s;
  }

  public static List empty () {
    return new List(null,null);
  }

  public static List cons (int i, List s) {
    return new List(new Integer (i),s);
    // class Integer lets you use null as a value
  }

  public boolean isEmpty () {
    return (this.firstElement == null);
  }

  public int first () {
    if (this.isEmpty())
      throw new RuntimeException("empty().first()");
    return this.firstElement;
  }

  public List rest () {
    if (this.isEmpty())
      throw new RuntimeException("empty().rest()");
    return this.restElements;
  }

  public String toString () {
    if (this.isEmpty())
      return "<empty>";
    return this.first() + " " + this.rest();
  }
}
```

This code and its underlying representation is ugly, and I mean that as a technical term. Part of the problem is that there is an implicit invariant: whenever the `firstElement` field is not null, then the `restElements` field better not be null either. (Can you see what can go wrong if we do not respect this invariant?) We must make sure that the implementation always preserves that invariant. There are ways around that, making the invariant more robust, but they're a bit unsatisfying.

Another problem with the implementation is that there are all kinds of checks in the code that test whether the list is empty or not. It would be much better to instead have two kinds of lists, an empty list and a "cons list", and have them implement their respective methods knowing full well what their representation is. This is what we're going to explore now.

This is a design pattern that I will call the *Specification Design Pattern*. A design pattern is just a "recipe" for how to write code to achieve a certain objective, here, to implement an ADT.

The idea is to use define a class `List`, and to use subclasses of `List` to keep track of the kind of list we have — these subclasses will implement their methods in full confidence that the list they are manipulating is of the right kind, without needing to check the representation.

One consequence of all the action happening in the subclasses is that the `List` class, by itself, does not represent anything. The representation is in the subclasses. It does not make sense to create an object of class `List` anymore. We will only create an object of class `EmptyList` or `ConsList`, as they will be named. To enforce this, we are going to make the `List` class `abstract`. An abstract class is a class that cannot be instantiated.[3] Therefore, it does not have a Java constructor. The only use of an abstract class is to serve as a superclass for other classes. An abstract class need not implement all its methods. It can have abstract methods that simply promise that subclasses will implement those methods. (Java will enforce this, by making that all concrete subclasses of the abstract class do provide an implementation for the methods.)

Here is a formal description of the Specification Design Pattern for implementing an im-

[3]In contrast, a class that *can* be instantiated is sometimes called a *concrete* class.

mutable ADT that is specified using an algebraic specification. Let T be the name of the ADT.

We make the following assumptions on the structure of the ADT defintion:

- Assume that the signature is given in OO-style: except for the basic creators, each operation of the ADT takes an implicit argument which is an object of type T.
- Except for the basic creators, each operation is specified by one or more equations. If an operation is specified by more than one equation, then the left hand sides of the equations differ according to which basic creator was used to create an argument of type T.
- The equations have certain other technical properties that allow them to be used as rewriting rules.
- We are to implement the ADT in Java.
- The creators of the ADT are to be implemented as static methods of a class named T.
- Other operations of the ADT are to be implemented as methods of a class named T.

The design pattern proper is the following:

(1) Determine which operations of the ADT are basic creators and which are other operations.

(2) Define an abstract class named T.

(3) For each basic creator c, define a concrete subclass of T whose instance variables correspond to the arguments that are passed to c. For each such subclass, define a Java constructor that takes the same arguments as c and stores them into the instance variables.

(So far we have defined the representation of T. Now we have to define the operations.)

(4) For each creator of the ADT, define a static method within the abstract class that creates and returns a new instance of the subclass that corresponds to c.

(5) For each operation f that is not a basic creator, define an abstract method f.

(6) For each operation f that is not a basic creator, and for each concrete subclass C of T, define f as a dynamic method within C that takes the arguments that were declared for the abstract method f and returns the value specified by the algebraic specification for the case in which Java's special variable `this` will be an instance of C. If the algebraic specification does not specify this case, then the code for f should throw a `RuntimeException` such as an `IllegalArgumentException`.

Following this design patter for the List ADT above gives the following. Can you match the steps with what gets produced?

```
public abstract class List {

  public static List empty () {
    return new EmptyList ();
  }

  public static List cons (int i, List s) {
    return new ConsList (i,s);
  }

  public abstract boolean isEmpty ();
  public abstract int first ();
  public abstract List rest ();
  public abstract String toString ();

}



class EmptyList extends List {

  // this is the only time where we'll have public constructors
  public EmptyList () { }

  public boolean isEmpty () { return true; }

  public int first () {
    throw new RuntimeException("empty().first()");
  }

  public List rest () {
    throw new RuntimeException("empty().rest()");
  }

  public String toString () {
    return "<empty>";
  }
}
```

```
class ConsList extends List {
  private int firstElement;
  private List restElements;

  // this is the only time where we'll have public constructors
  public ConsList (int i, List s) {
    firstElement = i;
    restElements = s;
  }

  public boolean isEmpty () { return false; }

  public int first () { return this.firstElement; }

  public List rest () { return this.restElements; }

  public String toString () { return this.first() + "  " + this.rest(); }
}
```