Code Reuse: Subclassing

Suppose that we extended the Point ADT into a Colored Point ADT, with the following interface:

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

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).getColor() = c
create(x,y,c).newColor(d) = create(x,y,d)
```

I claim that this is an extension of the Point ADT simply because every colored point is really a point: everything you can do to a point, you can in fact do to a colored point. This is reflected by the algebraic specification of the two classes, which in some sense agree on the operations that the ADT have in common. **Thought exercise:** can you make formal this relationship? More precisely, how can you get, from the specification of Point and CPoint, that every colored point in fact behaves just like a point? For now, let us rely on our intuition that colored points are points.

What does this mean? It means, among other things, that if we implement the CPoint ADT as a class, we should really make it so that it is a *subclass* of the Point class. This means that an object of class of CPoint can be passed to any method or stored in any variable expecting an object of class Point. The main requirement is that every public method of Point should also be a public method of CPoint. Of course, the implementation of the method can be quite different, but the method should exist, and have the same types. (Similarly, every public field of Point should be a public field of CPoint. But because we will never use public fields in this course —immutability!—we only have to worry about methods.)

Definition: *C is a subclass of D if an instance of C can be used in any context where an instance of D is expected, without causing a "method not found" runtime error.*

Subclassing is expressed in Java using an `extends` annotation on the class. 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: Point can now have subclasses
  protected Point () { }

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

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

  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 getColor () { 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.

This seems like awfully redundant code. And it is. And some of you are screaming, what about inheritance? We'll see inheritance next week. 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 bag of problems with it, that subclassing by itself does not have.

Subclassing enables code reuse. How? **Subclassing lets you reuse client code**. For instance, suppose that you have a method that in its body takes two points p, q, and computes their distance, by using

```
...
double dist = Math.sqrt(Math.pow(p.xPos()-q.xPos(),2.0) +
                        Math.pow(p.yPos()-q.yPos(),2.0));
...
```

Now, this *same piece of code* can be used to work on colored points, because the class `CPoint` is a subclass of `Point`, and because every colored point is guaranteed to have methods `xPos` and `yPos`.

Subclassing for ADTs Implementation

An interesting use of subclassing is to implement some forms of ADTs more cleanly, especially ADTs that have different representations for their values.

Take integer stacks as an example. Consider the following integer stack ADT:

```
public static Stack empty ();
public static Stack push (Stack, int);
public boolean isEmpty ();
public int top ();
public Stack pop ();
public String toString ();

empty().isEmpty() = true
push(s,i).isEmpty() = false
push(s,i).top() = i
push(s,i).pop() = s
empty().toString() = "<bottom of stack>"
push(s,i).toString() = i + " " + s.toString()
```

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

```
public class Stack {
  private Integer topValue;
  private Stack rest;

  private Stack (Integer i, Stack s) {
    topValue = i;
    rest = s;
  }

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

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

  public boolean isEmpty () {
    return (this.topValue == null);
```

```
  }

  public int top () {
    if (this.isEmpty())
      throw new IllegalArgumentException("top() invoked on empty stack");
    return this.topValue;
  }

  public Stack pop () {
    if (this.isEmpty())
      throw new IllegalArgumentException("pop() invoked on empty stack");
    return this.rest;
  }

  public String toString () {
    if (this.isEmpty())
      return "<bottom of stack>";
    return this.top() + " " + this.pop();
  }
}
```

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 `topVal` field is not null, then the `rest` 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 stack is empty or not. It would be much better to instead have two kinds of stacks, an empty stack and a "push stack", and have them implement their respective methods knowing full well what their representation is. This is what we're going to explore now. We will use subclasses to keep track of the kind of stack we have, and the subclasses will implement their methods in full confidence that the stack they are manipulating is of the right kind, without needing to check the representation.

But that means that the `Stack` class, by itself, does not represent anything. The representation is in the subclasses. It does not make sense to create an object of type `Stack` anymore. We will only create an object of class `EmptyStack` or `PushStack`, as they will be named. To enforce this, we are going to make the `Stack` class `abstract`. An abstract class is a class that cannot be instantiated.[1] 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

[1] In contrast, a class that can be instantiated is sometimes called a *concrete* 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 recipe for implementing an immutable ADT that is specified by an algebraic specification using subclasses.

Let T be the name of the ADT.

Assumptions on the structure of the ADT interface:

- 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.

Steps of the recipe:

- Determine which operations of the ADT are basic creators and which are other operations.
- Define an abstract class named T.
- 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.)
- 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.
- For each operation f that is not a basic creator, define an abstract method f.

- 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 recipe for the Stack ADT above gives the following. Can you match the steps with what gets produced?

```
public abstract class Stack {

  public static Stack emptyStack () {
    return new EmptyStack ();
  }

  public static Stack push (Stack s, int i) {
    return new PushStack (s,i);
  }

  public abstract boolean isEmpty ();
  public abstract int top ();
  public abstract Stack pop ();
  public abstract String toString ();

}


class EmptyStack extends Stack {
  public EmptyStack () { }

  public boolean isEmpty () { return true; }

  public int top () {
    throw new IllegalArgumentException ("Invoking top() on empty stack");
  }

  public Stack pop () {
    throw new IllegalArgumentException ("Invoking pop() on empty stack");
  }

  public String toString () {
    return "<bottom of stack>";
```

```
  }
}


class PushStack extends Stack {
  private int topVal;
  private Stack rest;

  public PushStack (Stack s, int v) {
    topVal = v;
    rest = s;
  }

  public boolean isEmpty () { return false; }

  public int top () { return this.topVal; }

  public Stack pop () { return this.rest; }

  public String toString () { return this.top() + " " + this.pop(); }
}
```