# 8 Topics in Subclassing

The idea today is to round off our study of subclassing by examining some of the consequences of having subclassing in the language, as a trade-off for the added ability to reuse client code.

But first, let's take another look at our Specification Design Pattern.

## 8.1 Nested Classes

Last time, we saw a design pattern for deriving an implementation from an ADT specification.

There are two problems with that approach, however, one minor, one major:

(1) Namespace pollution

(2) Extensibility

Let's take care of the minor one now, namespace pollution. We'll return to the extensibility problem later in the course.

Consider the `List` class as derived from the design pattern. It consists of a public abstract class `List`, and two concrete subclasses, `EmptyList` and `ConsList`. Now, because of the way Java works, either all of those classes must be made public, or, in the case where you decide to put all the classes in the same file, only `List` is made public, and the two subclasses are unannotated. This makes the two subclasses *package-public*: they are visible to other classes in the same package, but unavailable to classes outside the package. In other word, they are public for classes within the same package, but private for classes outside the package.

In practice, the only class we really want to be able to create instances of `EmptyList` and `ConsList` is the `List` class. All list object creation should go through the `List` class static methods. In parts, this is because we rarely want to have objects specifically of class `EmptyList` or `ConsList` about, but rather only want to have `List` objects. The static creators in the `List` class ensure this is the case. This is because we don't want client code to rely on those classes being there, because we want to make sure that later on we can come back and replace this implementation of lists if need be, using one that does not rely on those subclasses. Such a replacement, as long as the resulting implementation still satisfies the specification, should be transparent to the clients, which will not be the case if the clients expect either `EmptyList` or `ConsList` to exist.

So how do we prevent `EmptyList` and `ConsList` from being available even from within the package? The answer, which may or may not be obvious, is to simply package up the two subclasses directly within the `List` class, and make them private so that they cannot be accessed from outside the class. Here is the code:

```
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 nested inside List
  private static class EmptyList extends List {
    public EmptyList () { }

    public boolean isEmpty () { return true; }

    public int first () {
      throw new IllegalArgumentException("first() on empty list");
    }

    public List rest () {
      throw new IllegalArgumentException("rest() on empty list");
    }

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


  // Class ConsList nested inside List
  private static class ConsList extends List {
    private int firstElement;
    private List restElements;

    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(); }
  }
}
```

These are examples of *nested classes*. In fact, these are *static* nested classes. A static nested class is just like a class defined in its own file, except that it is defined within another class. (If a nested class $T$ is defined as public within a class $S$, then class $T$ can be accessed from outside $S$ as $S.T$.)

Why do we need the static qualifier? The static qualifier, as usual, indicates that the nested class is associated with the class definition itself. In particular, the nested class cannot access instance variables of the class containing it. (Nor can it invoke instance methods of the class containing it.) This essentially says that the nested class is defined only once, when the containing class is itself defined. Note that this is a "containment" relation. Class $S$ contains class definition $T$, just like it contains static methods and static fields.

(Without the static qualifier, a nested class (say $T$) is associated with instances of the class containing the definition (say $S$). Every instance of $S$ "redefines" the nested class $T$. In particular, the nested class can refer to instance variables of $S$. When they are not static, nested classes are called *inner classes*. Inner classes are very powerful, and are related to closures, which can be used to do higher-order programming. In other words, inner classes give you `lambda`. We will not use inner classes much in this course, but it's good to know that they exist.)

Nesting classes takes care of the namespace pollution problem. As I said, we will return to the extensibility problem later in a few lectures.

## 8.2 Compile-time and Run-time Types

Having a static type system with subclassing brings a whole range of issues to the fore. Not the least of which the need to distinguish between *compile-time types* and *run-time types*.

During program execution (i.e., at run time), an instance of a class `C` is just a chunk of allocated memory that carries a value for all the fields specified in `C`, *along with a mark that says that the object is an instance of class `C`*. This is the *run-time type* of the object. (Also sometimes called a dynamic type.)

We contrast this run-time type with the *compile-time type* of an object, which is the class that an object has according to the annotation we have in our source code. For instance, if we declare `Foo x = new Foo(...)`, then variable `x` and any value it holds has compile-time type `Foo`. (The compile-time type is also called the static type, but that's one more use of the word *static* that you don't need — so we'll stick to compile-time type.)

To show the difference between the two, recall the `distance()` function we saw last time:

```
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));
}
```

The compile-time type of `p` and `q` in the function is `Point` — that's what the source code tells you. And that's independent of how the call to `distance()` is made. As far as the type-checker is concerned, `p` and `q` are `Point`s.

Now, the run-time type of `p` and `q` depends on the actual call. Consider the following case:

```
Point p1 = Point.create(0,0);
Point p2 = Point.create(10,10);
double d = distance(p1,p2);
```

Values `p1` and `p2` are created using calls to `Point.create()`, which creates an instance of `Point`. So, the run-time type of `p1` and `p2` is `Point`, for both of them. Then the value of `p1` and `p2` is passed to `distance()`, and when `distance()` executes, `p` in `distance()` refers to `p1` and so has run-time type `Point`, and `q` in `distance()` refers to `p2` and so has run-time type `Point` as well. In this case the compile-time types and the run-time types agree.

What about the following call?

```
CPoint cp1 = CPoint.create(0,0,"red");
CPoint cp2 = CPoint.create(10,10,"blue");
double d = distance(cp1,cp2);
```

Values `cp1` and `cp2` are created using calls to `CPoint.create()`, which creates an instance of `CPoint`. So, the run-time type of `cp1` and `cp2` is `CPoint`, for both of them. Then the value of `cp1` and `cp2` is passed to `distance()`, and when `distance()` executes, `p` in `distance()` refers to `cp1` and so has run-time type `CPoint`, and `q` in `distance()` refers to `cp2` and so has run-time type `CPoint` as well. Here, the run-time type is different from the compile-time type. (The type system, by the way, will essentially ensure that the run-time type is always a subclass of the compile-time type.)

## 8.3 Dynamic Dispatch

So because of subclassing — the fact that we can pass values of a subclass to a method or function that expects values of a particular class without making the program unsafe— there is a distinction between the run-time type of a value and its compile-time type. Why do we care? Because it affects what methods gets called. Consider the `distance()` function above. It makes a call to method `xPos()` of argument `p`. There are two methods `xPos()` we've implemented: one for `Point`, and one for `CPoint`. Now, those methods happen to do the same thing, but there's no reason why they should.[1] So, when you say `p.xPos()`, which method gets invoked? The one in `Point`, or the one in `CPoint`.

Turns out it depends on the language. In some languages, the method called is the one defined in the compile-time type of the value on which the method is invoked. That's the default behavior in C++, for instance. In Java, the method called is the one defined in the run-time type of the value. That way of determining which method gets called — using the run-time type of the value — is called *dynamic dispatch*.

Dynamic dispatch is pretty powerful, because it makes it easy to adjust the behavior of objects by simply giving different definitions for a given method. It is one of the hallmarks of object-oriented programming, and one of the reason it took off when it did. But it's also a software engineering nightmare. Why?

Consider the definition `distance()` above. Someone looking at the code would think that the calls to `xPos()` and `yPos()` are to those method defined in `Point`, but we know that's not true. The methods invoked depend on the run-time type of the values, which can be of any subclass of `Point`. But that means that someone can define a subclass of `Point` that happens to do *anything* in its `xPos()` or `yPos()` method, and the function `distance()` will happily call those methods. Unless one knows exactly what the possible subclasses of `Point` exist and what their methods are doing, it is essentially impossible to predict just what `distance()` does. This gets worse when developing a library and offer something like `distance()`, because then you cannot guarantee anything to users about what the function can do.[2]

[1] If `distance()` for some reason called `toString()`, and if we implemented `toString()` in `Point` and `CPoint`, then the method would be implemented differently: in `Point` it would return a string containing the x and y coordinates of the point, while in `CPoint` it would return a string containing the x and y coordinates of the point as well as the color of the point.

[2] One way around this is to restrict the extent to which `Point` can be subclassed. Languages have ways to do that.

## 8.4 Information Loss and Downcasting

So there is this distinction between the compile-time type of a value and its run-time type. The thing is that the compile-time type of a value is what the type checker uses when type checking (before the program is allowed to execute), and one consequence of subclassing is that sometimes the type system can lose information about the type of values — forcing a compile-time type which is not precise enough for what you want to do, although you know full well that it would execute without a problem.

Consider the following example. First, you need to know something about how type checkers work. When type checking a function or method call, *the type checker only uses the signature of the function or method* — that is, its declared argument type, and its declared return type. There are a couple of reasons for this: one is that it lets you perform separate compilation; another is that it makes type checking much faster (were it too slow people would not use type checkers).

So given that we know this about type checking, what happens at method calls during type checking? We know that we can allow subclassing on argument types — see the `distance()` function above, which allows you to pass in instances of subclasses of `Point`. The type checker lets you do that, because the result is always safe. But what about the return type of the function? It turns out that we cannot allow subclassing of the result of functions or methods. If a function or method returns a `Point`, we must treat the result as a `Point` (or a superclass of `Point`). Doing anything else would allow us to write unsafe programs.

Here's an example. Consider the following two functions:

```
public static Point identity (Point p) {
   return p;
}

public static Point clone (Point p) {
   return Point.create(p.xPos(),p.yPos());
}
```

First note that `identity()` just returns its argument, while `clone()` constructs a new `Point` out of its argument. Second note that the two functions have exactly the same signature: they both take a `Point` argument, and return a `Point` result. Therefore, by the statement I made above regarding how the type checker works, the type checker will treat these two functions exactly the same.

Now, note that `identity()` returns its argument untouched. Meaning that if we construct a colored point and pass it to `identity`, the result will have run-time type `CPoint`. In contrast, `clone()` construct an actual `Point` from whatever its argument is, meaning that if we pass `clone()` a colored point, the result will have run-time type `Point`.

Here's another function that uses the code above;

```
public static Color extractColor1 (CPoint cp) {
   ??? newcp = identity(cp);
   return newcp.color();
}
```

What is the compile-time type of `newcp`? I claim it must be `Point`, the type stated in the signature of `identity()`. And that *despite the fact that we know full well that the run-time type of the result will always be a `CPoint` because the argument `cp` will be a `CPoint`.* Why?

Because that's the best the type checker can do. It must treat `identity()` and `clone()` — they have the same signature. So the type checker will type check the code above in the same way as it type checks the following one:

```
public static Color extractColor2 (CPoint cp) {
   ??? newcp = clone(cp);
   return newcp.color();
}
```

In this case, `newcp` has run-time type `Point`, because `clone()` creates an actual bona fide `Point`. And if we allow `???` to be `CPoint`, then the next line would try to call `color()` on a value with run-time type `Point`, which would fail with a method-not-found error. So the type checker has to force you to say that the result is a `Point`. Because `extractColor1()` and `extractColor2()` differ only in the function called, `identity()` or `clone()`, and because the type checker treats those two functions the same because they have the same type, the type checker treats the two functions `extractColor1()` and `extractColor2()` the same, and as we saw, the only conservative thing it can do, to avoid the unsafe example of `extractColor2()`, is to force you to write `Point` for `???`. Again, even though we know full well that in `extractColor1()`, writing a `CPoint` there would be safe. We know that, but the type checker doesn't.

So the best we can write is

```
public static Color extractColor1 (CPoint cp) {
   Point newcp = identity(cp);
   return newcp.color();
}
```

which of course fails to type check because `newcp` has compile-time type `Point`, and when type checking the call `newcp.color()`, the type checker complains that a value with compile-time type `Point` is not guaranteed to have a `color()` method, so the call fails to type check.

This is an example where we've lost compile-time type information when calling `identity` — we know the argument has compile-time type `CPoint`, but the best we can say is that the result has compile-time type `Point`, even though, as I said, in `extractColor1()`, we know full well that the result always have run-time type `CPoint`. Can we recover this information?

One way to regain this information is to try to *cast* the object to the run-time type we expect it to have. A cast is an expression

```
(C) e
```

where `C` is a class name, and `e` is an expression that yields an object.[3] As far as the type checker is concerned, the result of a cast is a value with compile-time type `C`. A cast is an assertion you, the programmer, are making, and the type checker takes you at face value.[4] A cast executes as follows: first evaluate `e` down to an object, then check if the run-time type of the result is a subclass of `C` — if not, the cast throws a `ClassCastException`; if it is, then the program continues. The idea is that we can adjust the compile-time type of an expression (presumably to recover compile-time type information that was lost), but the trade-off is that there is some checking that needs to occur during execution, as opposed to doing all the checking during type checking, before the code executes.

Because downcasts can throw an exception when they fail, you should wrap downcasts with a try/catch combination to ensure that your whole program does not abort. Alternatively, you can also check whether the downcast will succeed by checking that the `obj` you want to downcast to class `C` can be viewed as an instance of `C`, by evaluating `obj instanceof C`, which is true if the run-time type of `obj` is a subclass of `C`.

Using casts, we can correct `extractColor1()` so that the type checker accepts it:

```
public static Color extractColor1 (CPoint cp) {
   Point newcp_temp = identity(cp);
   CPoint newcp = (CPoint) newcp_temp;
   return newcp.color();
}
```

or, equivalently,

```
public static Color extractColor1 (CPoint cp) {
   CPoint newcp = (CPoint) identity(cp);
   return newcp.color();
}
```

Now the type checker is happy with the code — `identity()` returns a value with compile-time type `Point`, but we cast it to `CPoint`, and the type checker trusts you and lets you treat the result as a `CPoint`. During execution, after calling `identity(cp)` and getting a value back, the cast checks that the run-time type of the value is a `CPoint`, which we know it

[3]We sometimes call a cast a *downcast* when casting expression `e` to a class `C` that is a subclass of the compile-time type of `e`.

[4]There is some sanity checking going on. In Java, the casting operator checks that the class `C` you are trying to cast expression `e` to is either a subclass or a superclass of the compile-time type of `e`.

will be, and the program happily executes and extracts the color form the provided colored point.

Now, to see if you understand what's going on, tell me whether the following code type checks, and if it does what happens during execution? (This is a perfect exam question, by the way...)

```
public static Color extractColor3 (CPoint cp) {
   CPoint newcp = (CPoint) clone(cp);
   return newcp.color();
}
```