CS61B Lecture 9

Monday, February 10, 2020

Abstract Classes

Instance methods can be **abstract**. No body is given, but must be supplied in its subtype. One good use of this is specifying a pure interface to a family of types:

```
/** A drawable object. */
public abstract class Drawable {
    // "abstract class" = "can't say new Drawable"

    /** Expand THIS by a factor of XSIZE in the X
     *  direction, and YSIZE in the Y direction. */
     public abstract void scale(double xsize, double ysize);

     /** Draw THIS on the standard output. */
     public abstract void draw();
}
```

Now a `Drawable` is a class that has *at least* the operatoons `scale` and `draw` on it. You can't create an instance of it, because it's abstract, and in this case, it would make no sense anyway, because it just has two methods that lack an implementation.

Methods on Drawable

So what is the point of this anyway? We can't call `new Drawable()`, but we can write methods that operate on `Drawable`s in `Drawable` in other classes:

```
void drawAll(Drawable[] thingsToDraw) {
    for (Drawable thing : thingsToDraw) {
        thing.draw();
    }
}
```

How can this work? Well, regular classes can extend abstract classes to make them less abstract, overriding their abstract methods. We can define different kinds of `Drawable`s that are concrete, in that all methods have implementations, and one can use `new` on them.

```
public class Rectangle extends Drawable {
    public Rectangle(double w, double h) {this.w = w; this.h = h; }
    public void scale(double xsize, double ysize) {
        w *= xsize; h *= ysize;
    }
    public void draw() { // draw a W*H rectangle }
    private double xrad, yrad;
}

public class Oval extends Drawable {
    public Oval(double xrad, double yrad) {
```

```
        this.xrad = xrad; this.yrad = yrad;
    }
    public void scale(double xsize, douuble ysize) {
        xrad *= xsize; yrad *= ysize;
    }
    public void draw() { // draw an oval with axes XRAD and YRAD }
    private double xrad, yrad;
}
```

Unlike with `Drawable`, we can create new `Rectangle`s and `Oval`s. Since these classes are subtypes of `Drawable`, we can put them in any container whose static type is `Drawable`, and therefore we can pass them to any method that expects `Drawable` parameters:

```
Drawable[] things = {
    new Rectangle(3,4), new Oval(2,2);
}
drawAll(things);
```

This code draws a 3*4 rectangle, and a circle with a radius of 2.

> **Aside: 61B specifics**
>
> The course staff have provided a style checker, which generally insists of comments for all methods, constructors and fields of the concrete subtypes.
>
> However, we already have comments for `draw` and `scale` in the class `Drawable`, and the whole idea of object-oriented programming is that the subtypes conform to the supertype both in syntax and behavior (all `scale` methods scale their figure), so comments are generally not helpful on overriding methods. Still, the reader would like to know that a given method does override something.
>
> Hence, when you ride an overriding method, we give it the `@Override` notation:
>
> ```
> @Override
> public void scale(double xsize, double ysize) {
>     xrad *= xsize; yrad *= ysize;
> }
>
> @Override
> public void draw() { // draw a circle with radius rad }
> ```
>
> The compiler will check that these method headers are proper overridings of the parent's methods, and the style checker won't complain about the lack of comments.

Interfaces

In programming, we often use the term "interface" to mean a description of this generic interaction, specifically, a description of the functions or variables by which two things interact. Java uses the term to refer to a slight variant of an abstract class that (until Java 1.7) contains only abstract methods (and static constants), like this:

```
public interface Drawable {
    void scale(double xsize, double ysize); //Automatically public.
    void draw();
}
```

Interfaces are automatically abstract: can't say new `Drawable()`, but you can say `new Rectangle(...)`.

There are some difference between interfaces and abstract classes. You don't write `extend`, but instead `implement`.

```
public class Rectangle implements Drawable { ... }
```

We can use the interface as for abstract classes:

```
void drawAll (Drawable[] thingsToDraw) {
    for (Drawable thing : thingsToDraw) {
        thing.draw();
    }
}
```

Again, this works for any other implementation of `Drawable`.

Multiple Inheritance

You can only extend one class, but you can implement any number of interfaces:

```
interface Readable {
    Object get();
}

interface Writable {
    void put(Object x);
}

class Source implements Readable {
    public Object get() { ... }
}

class Sink implements Writable {
    public void put(Object x) { ... }
}

class Variable implements Readable, Writable {
    public Object get() { ... }
    public void put(Object x) { ... }
}

void copy(Readable r, Writable w) {
    w.put(r.get());
}
```

The first argument of the `copy` method can be a `Source` or a `Variable`, while the second can be a `Sink` or a `Variable`.

Higher-Order Functions

In Python, we had higher order functions like this:

```
def map(proc, items):
    """Apply PROC to every item in ITEMS."""
    if items is None:
        return None
    else:
        return ...

map(lambda x: x*x, makeList(1,2,3))
```

Java does not have higher-order functions in this sense: you can write a method that directly takes in or returns another method. We can, however, use abstract classes or interfaces and subtyping to get the same effect.

Map

In Java, the map function is rather strange and clumsy to use:

```
public interface IntUnaryFunction {
    int apply(int x);
}

class Abs implements IntUnaryFunction {
    public int apply(int x) {
        return Math.abs(x);
    }
}

IntList map(IntUnaryFunction proc, IntList items) {
    if (items == null) {
        return null
    } else {
        return new IntList(proc.apply(items.head),map(proc,items.tail));
    }
}
```

Java developers made it easier to do this by allowing anonymous classes, starting with Java 7:

```
R = map(new IntUnaryFunction { public int apply(int x) {return Math.abs(x);} },
IntList items);

/** Equivalent to: */
class Anonymous implements IntUnaryFunction {
    public int apply(int x) { return Math.abs(x); } }
R = map(new Anonymous(), some list)
```

And in Java 8, then were made even more succinct:

```
R = map((int x) -> Math.abs(x), IntList items);

/** or, if such a function already exists */
R = map(Math::abs, IntList items);
```

Java now figures out you need an anonymous `IntUnaryFunction` and creates one for you. Such examples can be seen in Signpost's GUI.

```
addMenuButton("Game->New", this::newGame);
```

It has the Java library type `java.util.function.Consumer`, which has a one-argument method, like `IntUnaryFunction`,

More on Inheritance

One can implement multiple interfaces, but extend only one class: **multiple interface inheritance**, but **single body inheritance**. This scheme is simple, and pretty easy for language implementers to implement. However, there are cases where it would be nice to be able to "mix in" implementations from a number of sources.

Before Java 8, interfaces contained just static constants and abstract methods. Java 8 introduced static methods into interfaces and also default methods, which are essentially instance methods and are used whenever a method of a class implementing the interface would otherwise be abstract.

For example, I want to add a new, one parameter `scale` method to all concrete subclasses to the interface `Drawable` (our previous method took 2 parameters).

One way to do this is to make `Drawable` abstract again, but that comes with the same restrictions as before.

Thus, Java 8 introduced default methods:

```
public interface Drawable {
    void scale
    (double xsize, double ysize);
    void draw();
    /** Scale by SIZE in the X and Y dimensions. */
    default void scale(double size) {
        scale(size, size);
    }
}
```

It's a useful feature, but as in other languages with full multiple inheritance, it can lead to confusing programs.