CS61B Lecture #9: Interfaces and Abstract Classes

Recreation

Show that for any polynomial with a leading coefficient of 1 and integral coefficients, all rational roots are integers.

Reminders:

1. The four projects are individual efforts in this class (no partnerships). Feel free to discuss projects or pieces of them before doing the work. But you must complete each project yourself. That is, feel free to discuss projects with each other, but be aware that we expect your work to be substantially different from that of all your classmates (in this or any other semester). You will find a more detailed account of our policy in under the "Course Info" tab on the course website.
2. `git status` is your friend!
3. Please use `git-bug` for submitting problems to us. Piazza and email are really not great for bug reports.

Abstract Methods and Classes

- Instance method can be *abstract:* No body given; must be supplied in subtypes.
- One good use is in 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 something that has *at least* the operations `scale` and `draw` on it.
- Can't create a `Drawable` because it's abstract.
- In fact, in this case, it wouldn't make any sense to create one, because it has two methods without any implementation.

Methods on Drawables

```
/** A drawable object. */
public abstract class 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();
}
```

- Can't write new Drawable(), *BUT*, we can write methods that operate on Drawables

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

- But draw has no implementation! How can this work?

Concrete Subclasses

- Regular classes can extend abstract ones to make them "less abstract" by overriding their abstract methods.
- Can define kinds of `Drawables` that are *concrete* (non-abstract), with all methods having implementations.
- Since all methods are implemented, it makes sense to use **new** on these types—all the method calls make sense.

Concrete Subclass Examples

```
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 x h rectangle }
    private double w,h;
}
```

Any Oval or Rectangle is a Drawable.

```
public class Oval extends Drawable {
    public Oval(double xrad, double yrad) {
        this.xrad = xrad; this.yrad = yrad;
    }
    public void scale(double xsize, double ysize) {
        xrad *= xsize; yrad *= ysize;
    }
    public void draw() { draw an oval with axes xrad and yrad }
    private double xrad, yrad;
}
```

Using Concrete Classes

- We *can* create new `Rectangles` and `Ovals`.
- Since these classes are subtypes of `Drawable`, we can put them in any container whose static type is `Drawable`, ...
- ...and therefore can pass them to any method that expects `Drawable` parameters:
- Thus, writing

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

draws a 3×4 rectangle and a circle with radius 2.

Aside: Documentation

- Our style checker would insist on comments for all the methods, constructors, and fields of the concrete subtypes.
- But we already have comments for `draw` and `scale` in the class `Drawable`, and good practice demands that all overridings of these methods should obey at least these comments. Hence, comments are often unhelpful on overriding methods.
- Still, the reader would like to know that a given method ***does*** override something. Hence, the `@Override` annotation:

```
@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 our style checker won't complain about the lack of comments.

Interfaces

- In generic English usage, an *interface* is a "point where interaction occurs between two systems, processes, subjects, etc." (*Concise Oxford Dictionary*).
- In programming, often use the term 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()`; can say `new Rectangle(...)`.

Implementing Interfaces

- Idea is to treat Java interfaces as the public specifications of data types, and classes as their implementations:

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

  (We *extend* ordinary classes and *implement* interfaces, hence the change in keyword.)

- Can use the interface as for abstract classes:

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

- Again, this works for Rectangles and any other implementation of Drawable.

Multiple Inheritance

- Can *extend* one class, but *implement* any number of interfaces.
- Contrived Example:

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

interface Writable {
  void put(Object x);
}

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

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

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

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

- The first argument of `copy` can be a `Source` or a `Variable`. The second can be a `Sink` or a `Variable`.

Review: Higher-Order Functions

- In Python, you had *higher-order functions* like this:

```
def map(proc,       items):
    #  function       list
    if items is None:
        return None
    else:
        return IntList(proc(items.head), map(proc, items.tail))
```

and you could write

```
map(abs, makeList(-10, 2, -11, 17))
  ====> makeList(10, 2, 11, 17)
map(lambda x: x * x, makeList(1, 2, 3, 4))
  ====> makeList(t(1, 4, 9, 16)
```

- Java does not have these directly, but we can use abstract classes or interfaces and subtyping to get the same effect (with more writing)

Map in Java

```
/** Function with one integer argument */

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

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

- It's the use of this function that's clumsy. First, define class for absolute value function; then create an instance:

```
class Abs implements IntUnaryFunction {
  public int apply(int x) { return Math.abs(x); }
}
-----------------------------------------
R = map(new Abs(), some list);
```

Lambda Expressions

- Since Java 7, one can create classes likes `Abs` on the fly with *anonymous classes*:

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

- This is sort of like declaring

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

  and then writing

```
R = map(new Anonymous(), some list);
```

Lambda in Java

- In modern Java, lambda expressions are even more succinct. If a parameter has a type is an interface with a single abstract method (a *functional interface*), Java will figure out the necessary class from just a parameter list and body:

```
R = map((int x) -> Math.abs(x), some list);
```

We don't even need the parameter's type:

```
R = map((x) -> Math.abs(x), some list);
```

and if the body is just a call on a function that already exists:

```
R = map(Math::abs, some list);
```

- These figure out you need an anonymous `IntUnaryFunction` and create one.
- You can see examples of this sort of thing in `flood.GUI`:

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

Here, the second parameter of `ucb.gui2.TopLevel.addMenuButton` is a *call-back function.*

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

Inheriting Headers vs. Method Bodies

- 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 implementors to implement.
- However, there are cases where it would be nice to be able to "mix in" implementations from a number of sources.

Extending Supertypes, Default Implementations

- As indicated above, 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.
- Suppose I want to add a new one-parameter `scale` method to all concrete subclasses of the interface `Drawable`. Normally, that would involve adding an implementation of that method to all concrete classes.
- We could instead make `Drawable` an abstract class again, but in the general case that can have its own problems.

Default Methods in Interfaces

- So 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);
  }
}
```

- Now in any class the implements `Drawab;e` but does not have a definition of `scale` with one argument, this method will supply a default implementation for it.
- Useful feature, but, as in other languages with full multiple inheritance (like C++ and Python), it can lead to confusing programs. I suggest you use them sparingly.

Fancy Example From the Java Library

```
package java.util.function;

public interface Consumer<T> {
    void accept(T t);

    default Consumer<T> andThen(Consumer<? super T> after) {
        return (x) -> { accept(x); after.accept(x); }
    }
}
```

- Don't worry about the weird stuff in < > for now. We'll get to that when we look at *generic definitions*.
- The `andThen` method is another example of Java's version of higher-order function:

```
Consumer<String> print1 = (x) -> System.out.print(x);
print1.accept("Hello");   // Prints "Hello"
Consumer<String> print2 = print1.andThen((y) -> System.out.print(y));
print2.accept("Hello");   // Prints "HelloHello"
```