CS61B Lecture #9: Interfaces and Abstract Cl

Recreation

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

Reminder:

The four projects are individual efforts in this class (no partne
Feel free to discuss projects or pieces of them before doing th
But you must complete each project yourself. That is, feel
discuss projects with each other, but be aware that we expe
work to be substantially different from that of all your classm
this or any other semester). You will find a more detailed acc
our policy in under the "Course Info" tab on the course website

Abstract Methods and Classes

- Instance method can be *abstract:* No body given; must be s in subtypes.
- One good use is in specifying a pure interface to a family of

```
/** A drawable object. */
public abstract class Drawable {
    // "abstract class" = "can't say new Drawable"
    /** Expand THIS by a factor of XSIZE in the X di
     *  and YSIZE in the Y direction. */
    public abstract void scale(double xsize, double

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

- Now a `Drawable` is something that has *at least* the operation 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 a cause it has two methods without any implementation.

Methods on Drawables

```
/** A drawable object. */
public abstract class Drawable {
    /** Expand THIS by a factor of SIZE */
    public abstract void scale(double xsize, do
    /** Draw THIS on the standard output. */
    public abstract void draw();
}
```

- Can't write `new Drawable()`, *BUT*, we can write methods that on `Drawables` in `Drawable` or in other classes:

```
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 "
  stract" by overriding their abstract methods.
- Can define kinds of `Drawable`s that are *concrete*, in that all r
  have implementations and one can use **new** on them:

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 any container whose static type is `Drawable`, ...
- ...and therefore can pass them to any method that expects `D` 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 m
  constructors, and fields of the concrete subtypes.
- But we already have comments for `draw` and `scale` in the class
  and the whole idea of object-oriented programming is that t
  types conform to the supertype both in syntax and beha
  `scale` methods scale their figure), so comments are gener
  helpful on overriding methods. Still, the reader would like
  that a given method ***does*** override something.
- Hence, the `@Override` annotation. We can write:

```
@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 prop
  ridings of the parent's methods, and our style checker wo
  plain about the lack of comments.

Interfaces

- In generic English usage, an *interface* is a "point where inte
  occurs between two systems, processes, subjects, etc." (
  Oxford Dictionary).
- In programming, often use the term to mean a *description*
  generic interaction, specifically, a description of the func
  variables by which two things interact.
- Java uses the term to refer to a slight variant of an abstra
  that (until Java 1.7) contains only abstract methods (and sta
  stants), like this:

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

- Interfaces are automatically abstract: can't say `new Draw`
  can say `new Rectangle(...)`.

Implementing Interfaces

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

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

(We *extend* ordinary classes and *implement* interfaces, he 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 implement Drawable.

Multiple Inheritance

- Can *extend* one class, but *implement* any number of interfa
- 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
  public void put(Obje
}
```

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

- The first argument of copy can be a Source or a Variabl 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, it
```

  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 can use abstract cl
  interfaces and subtyping to get the same effect (with more

Map in Java

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

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

```
IntList map(IntUnaryF
            IntList i
  if (items == null)
    return null;
  else return new Int
      proc.apply(item
      map(proc, items
    );
}
```

- It's the use of this function that's clumsy. First, define c 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
  mous classes:

```
R = map(new IntUnaryFunction() {
            public int apply(int x) { return Math.abs
        }, 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 8

- In Java 8, lambda expressions are even more succinct:

  ```
  R = map((int x) -> Math.abs(x), some list);
  ```
  or even better, when the function already exists:
  ```
  R = map(Math::abs, some list);
  ```

- These figure out you need an anonymous `IntUnaryFunction` a
  ate one.

- You can see examples in `signpost.GUI`:

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

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

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

Inheriting Headers vs. Method Bodies

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

Extending Supertypes, Default Implementatio

- As indicated above, before Java 8, interfaces contained jus constants and abstract methods.
- Java 8 introduced static methods into interfaces and also *methods*, which are essentially instance methods and are use ever a method of a class implementing the interface would ot be abstract.
- Suppose I want to add a new one-parameter `scale` method to crete subclasses of the interface `Drawable`. Normally, tha involve adding an implementation of that method to all c classes.
- We could instead make `Drawable` an abstract class again, bu 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);
  }
}
```

- Useful feature, but, as in other languages with full multipl
itance (like C++ and Python), it can lead to confusing prog
suggest you use them sparingly.