Recreation

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

Reminder

"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 and write up 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)."

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

- Can't write new Drawable(), *BUT*, we can write methods that operate 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 "less abstract" by overriding their abstract methods.
- Can define 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 size) { w *= size; h *= size; }
  public void draw() { draw a w x h rectangle }
  private double w,h;
}
```

Any Circle or Rectangle is a Drawable.

```
public class Circle extends Drawable {
  public Circle(double rad) { this.rad = rad; }
  public void scale(double size) { rad *= size; }
  public void draw() { draw a circle with radius rad }
  private double rad;
}
```

Using Concrete Classes

- We *can* create new `Rectangles` and `Circles`.
- 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 Circle(2)
};
drawAll(things);
```

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

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 size);   // 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 { ... }
```

- 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 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

- In 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 x*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` and create one.
- You can see examples in `cube.CubeGUI`:

  ```
  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`,

More Useful (albeit Dangerous) Features of Java 8

- As indicated above, before Java 8, interfaces contained just static constants and abstract methods.
- 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.
- 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.
- Useful feature, but, as in other languages with full multiple inheritance (like C++ and Python), it can lead to confusing programs.
- Concensus is that the new default method feature should be used sparingly.