ture #9: Interfaces and Abstract Classes

Recreation

ny polynomial with a leading coefficient of 1 and integral
 rational roots are integers.

ects are individual efforts in this class (no partnerships).
cuss projects or pieces of them before doing the work.
omplete each project yourself. That is, feel free to
s with each other, but be aware that we expect your
tantially different from that of all your classmates (in
r semester). You will find a more detailed account of
ler the "Course Info" tab on the course website.
s your friend!

Abstract Methods and Classes

thod can be *abstract:* No body given; must be supplied

 is in specifying a pure interface to a family of types:

```
ble object. */
ract class Drawable {
tract class" = "can't say new Drawable"
and THIS by a factor of XSIZE in the X direction,
 YSIZE in the Y direction. */
abstract void scale(double xsize, double ysize);

w THIS on the standard output. */
abstract void draw();
```

ble is something that has *at least* the operations scale
it.

a Drawable because it's abstract.

this case, it wouldn't make any sense to create one,
as two methods without any implementation.

Methods on Drawables

```
able object. */
tract class Drawable {
pand THIS by a factor of XSIZE in the X direction,
d YSIZE in the Y direction. */
 abstract void scale(double xsize, double ysize);
aw THIS on the standard output. */
 abstract void draw();
```

ew Drawable(), *BUT*, we can write methods that operate
s

```
awAll(Drawable[] thingsToDraw) {
 (Drawable thing : thingsToDraw)
 thing.draw();
```

 no implementation! How can this work?

Concrete Subclasses

es can extend abstract ones to make them "less abstract"
 their abstract methods.

kinds of Drawables that are *concrete* (non-abstract),
ods having implementations.

thods are implemented, it makes sense to use **new** on
–all the method calls make sense.

Concrete Subclass Examples

```
tangle extends Drawable {
ngle(double w, double h) { this.w = w; this.h = h; }
scale(double xsize, double ysize) {
ze; h *= ysize;

draw() { draw a w x h rectangle }
le w,h;
```

Oval or Rectangle is a Drawable.

```
l extends Drawable {
double xrad, double yrad) {
d = xrad; this.yrad = yrad;

scale(double xsize, double ysize) {
xsize; yrad *= ysize;

draw() { draw an oval with axes xrad and yrad }
le xrad, yrad;
```

Using Concrete Classes

te new Rectangles and Ovals.

classes are subtypes of Drawable, we can put them in
r whose static type is Drawable, ...

fore can pass them to any method that expects Drawable

g

```
e[] things = {
 Rectangle(3, 4), new Oval(2, 2)

(things);
```

 rectangle and a circle with radius 2.

Interfaces

nglish usage, an *interface* is a "point where interaction een two systems, processes, subjects, etc." (*Concise ionary*).

ing, often use the term to mean a *description* of this raction, specifically, a description of the functions or which two things interact.

e term to refer to a slight variant of an abstract class va 1.7) contains only abstract methods (and static constants),

```
ace Drawable {
double xsize, double ysize);   // Automatically public.
;
```

re automatically abstract: can't say `new Drawable();` `Rectangle(...)`.

Aside: Documentation

necker would insist on comments for all the methods, , and fields of the concrete subtypes.

dy have comments for `draw` and `scale` in the class `Drawable`, actice demands that all overridings of these methods at least these comments. Hence, comments are often overriding methods.

der would like to know that a given method *does* override Hence, the `@Override` annotation:

```
de
void scale(double xsize, double ysize) {
d *= xsize; yrad *= ysize;

de
void draw() { draw a circle with radius rad }
```

will check that these method headers are proper overridings t's methods, and our style checker won't complain about omments.

Multiple Inheritance

one class, but *implement* any number of interfaces.

ample:

```
dable {
);

table {
ject x);

implements Readable {
ct 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) { ... }
 }
```

gument of `copy` can be a `Source` or a `Variable`. The e a `Sink` or a `Variable`.

Implementing Interfaces

eat Java interfaces as the public *specifications* of data asses as their *implementations*:

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

ordinary classes and *implement* interfaces, hence the word.)

nterface as for abstract classes:

```
awAll(Drawable[] thingsToDraw) {
 (Drawable thing : thingsToDraw)
 thing.draw();
```

orks for `Rectangles` and any other implementation of

Map in Java

```
n one integer argument */

 IntUnaryFunction {
x);
```

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

of this function that's clumsy. First, define class for e function; then create an instance:

```
mplements IntUnaryFunction {
t apply(int x) { return Math.abs(x); }
```

`Abs(), some list);`

Review: Higher-Order Functions

ou had *higher-order functions* like this:

```
c,      items):
ion      list
 is None:
rn None

rn IntList(proc(items.head), map(proc, items.tail))
```

d write

```
, makeList(-10, 2, -11, 17))
 makeList(10, 2, 11, 17)
bda x: x * x, makeList(1, 2, 3, 4))
 makeList(t(1, 4, 9, 16)
```

t have these directly, but we can use abstract classes s and subtyping to get the same effect (with more writing)

Lambda Expressions

, one can create classes likes `Abs` on the fly with *anonymous*

```
 IntUnaryFunction() {
public int apply(int x) { return Math.abs(x); }
some list);
```

of like declaring

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

ting

```
(new Anonymous(), some list);
```

Lambda in Java

ava, lambda expressions are even more succinct. If a as a type is an interface with a single abstract method *l interface*), Java will figure out the necessary class parameter list and body:

```
t x) -> Math.abs(x), some list);
```

't even need the parameter's type:

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

he body is just a call on a function that already exists:

```
h::abs, some list);
```

out you need an anonymous `IntUnaryFunction` and create

examples of this sort of thing in `flood.GUI`:

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

cond parameter of `ucb.gui2.TopLevel.addMenuButton` *function.*

ava library type `java.util.function.Consumer`, which gument method, like `IntUnaryFunction`.

eriting Headers vs. Method Bodies

lement multiple interfaces, but extend only one class: *rface inheritance*, but *single body inheritance*.

is simple, and pretty easy for language implementors to

ere are cases where it would be nice to be able to "mix tations from a number of sources.

g Supertypes, Default Implementations

above, before Java 8, interfaces contained just static d abstract methods.

duced static methods into interfaces and also *default* ich are essentially instance methods and are used whenever a class implementing the interface would otherwise be

ant to add a new one-parameter `scale` method to all classes of the interface `Drawable`. Normally, that would g an implementation of that method to all concrete

tead make `Drawable` an abstract class again, but in the that can have its own problems.

Default Methods in Interfaces

troduced default methods:

```
rface Drawable {
e(double xsize, double ysize);
();

 by SIZE in the X and Y dimensions. */
oid scale(double size) {
(size, size);
```

ass the implements `Drawab;e` but does not have a definition n one argument, this method will supply a default implementation

re, but, as in other languages with full multiple inheritance Python), it can lead to confusing programs. I suggest n sparingly.

cy Example From the Java Library

```
util.function;

face Consumer<T> {
pt(T t);

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

about the weird stuff in < > for now. We'll get to that at *generic definitions*.

method is another example of Java's version of higher-order

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