Doing it in Java

CS 5010 Program Design Paradigms
"Bootcamp"

Lesson 5.4

Lesson Outline

- In this lesson, we'll see how the binary tree example from Lesson 5.1 might be done in Java.
- This is representative of other object-oriented languages.
- This lesson is enrichment for those of you who already know some Java.
- It is not intended to teach you Java or OOP; that will come later in the course.

Key Points for Lesson 5.4

- objects are like structs
- classes are like structure definitions, but with methods (functions) as well as fields
- To invoke a method of some object, send a message to the object.
- the interface of an object is the set of messages to which it responds
- interfaces correspond to data definitions
- Racket code and Java code are similar, but grouped differently

Classes

- A class is like a **define-struct**.
- It specifies the names of the fields of its objects.
- It also contains some *methods*. Each method has a name and a definition.
- To create an object of class **C**, we say

```
new C()
```

You say more than this, but this is good enough right now.

A typical class definition

```
class C1 {
        int x;
        int y;
        int r;

        public C1(int x_init, int y_init, int r_init) {
          x = x_init ; y = y_init; r = r_init; }

        public int foo () { return x + y; }
        public int bar (int n) { return r + n; }
        ...
     }
```

Every object of class **C1** has three fields, named **x**, **y**, and **r**.

Some annoying boilerplate for constructing objects of this class. This is the code that is executed when you call **new**. You don't need to worry about this right now.

The class definition also defines two methods, named **foo** and **bar**, that are applicable to any object of this class.

How do you compute with an object?

- To invoke a method of an object, we *invoke a method of the object.*
- For example, to invoke the **area** method of an object **obj1**, we write

```
obj1.area()
```

if area was a method that took an argument, like bar on the preceding slide, we'd put the argument here

- If **obj1** is an object of class **C**, this invokes the **area** method in class **C**.
- We sometimes say that we send **obj1** an **area** message.

Example

- If obj1 was an object of class C1 with
  - x = 10, y = 20, r = 14
- and obj2 was a object of class C1 with
  - x = 15, y = 35, r = 5
- then we would have
  - obj1.bar(8) = 22
  - obj2.bar(8) = 13

Every object knows its own class

```
class C2 {
    int a;
    int b;
    int c;

    // constructor (annoying boilerplate)
    public C2(int a_init, int b_init, int c_init) {
      a = a_init; b = b_init; c = c_init; }

    public int foo () { return a + b; }
    public int bar (int n) { return c * n; }

}
```

Here's a different class, with different field names and with the same names but different definitions than those in C1.

Every object knows its own class

- If we define obj3 by new C2(15,35,5), and we send a message to obj3, then the methods defined in class C2 will be invoked.
- So:
  - obj2.bar(8) = 5 + 8 = 14
  - obj3.bar(8) = 5 * 8 = 40

Interfaces are data types

- In Java, we characterize values by their behavior, not by their structure.
- The set of messages to which an object responds (along with their contracts) is called its *interface*.
- So the contract for an object-oriented method of function should be expressed in terms of *interfaces*.
- So interfaces play the role of data types in the OOP setting.

Example in Racket

```
;; Imagine we had a data definition in
;; Racket:

;; A GreenThing is represented as one of
;; -- (make-C1 x y r)
;; -- (make-C2 a b c)
;; with the following fields:
;;  x,y,r,a,b,c : Int
```

In Java, we do it differently

- In Java, we characterize objects by their behavior, not by their structure.
- So in Java we would write

A Java interface

```
// a GreenThing is an object of any class that implements
// the GreenThing interface.

// any class that implements GreenThing must provide
// methods named foo and bar
// with the specified contracts

interface GreenThing {

    int foo ();
    int bar (int n);

}
```

Classes C1 and C2 both implement GreenThing

```
class C1 implements GreenThing {
    int x;
    int y;
    int r;

    // constructor omitted

    public int foo () {
      return x + y; }
    public int bar (int n) {
      return r + n; }

}
```

```
class C2 implements GreenThing {
    int a;
    int b;
    int c;

    // constructor omitted

    public int foo () {
      return a + b; }
    public int bar (int n) {
      return c * n; }

}
```

In Java, you must explicitly declare the interface that a class is supposed to implement. Then the compiler checks that you've done it correctly.

So what?

- Now we can write a method that will take any GreenThing, whether it's a C1 or a C2:

```
static int apply_bar (GreenThing o) {
    return o.bar(8);
  }
```

static is Java's way of writing functions that are not associated with any object. If you don't know about this, don't worry about it for now– we are just trying to communicate ideas, not details.

Tests

```
C1 obj1 = new C1(10, 20, 14);
C1 obj2 = new C1(15, 35, 5)
C2 obj3 = new C2(15, 35, 5);

assert obj1.bar(8) == 22;
assert obj2.bar(8) == 13;
assert obj3.bar(8) == 40;

// now let's run the same three tests,
// but using the apply_bar method

assert apply_bar(obj1) == 22;
assert apply_bar(obj2) == 13;
assert apply_bar(obj3) == 40;
```

apply_bar will work on any **GreenThing**, whether it was constructed as a **C1** or as a **C2** (or any other class that implements **GreenThing**).

Now let's do binary trees

```
;; A Binary Tree is represented as a BinTree, which is either:
;; (make-leaf datum)
;; (make-node lson rson)

;; INTERPRETATON:
;; datum       : Real        some real data
;; lson, rson : BinTree     the left and right sons of this node

;; IMPLEMENTATION:
(define-struct leaf (datum))
(define-struct node (lson rson))

;; CONSTRUCTOR TEMPLATES:
;; -- (make-leaf Number)
;; -- (make-node Tree Tree)
```

Remember the Racket version from Lesson 5.1

The BinTree interface

```
// a BinTree is an object of any class that
// implements BinTree.

interface BinTree {

    int leaf_sum (); // returns the sum of the
                     // values in the leaves
    int leaf_max (); // returns the largest value
                     // in a leaf of the tree
    int leaf_min (); // returns the smallest value
                     // in a leaf of the tree

}
```

The Leaf class

```
class Leaf implements BinTree {
    int datum;   // some integer data

    Leaf (int n) {datum = n;}

    public int leaf_sum () {return datum;}
    public int leaf_max () {return datum;}
    public int leaf_min () {return datum;}

}
```

The Node class

```
class Node implements BinTree {
    BinTree lson, rson;   // the left and right sons

    Node (BinTree l, BinTree r) {lson = l ; rson = r;}

    public int leaf_sum () {
        return (lson.leaf_sum() + rson.leaf_sum());
    }

    public int leaf_max () {
        return (max (lson.leaf_max(), rson.leaf_max()));
    }

    public int leaf_min () {
        return (min (lson.leaf_min(), rson.leaf_min()));
    }
}
```

recur on the sons, and then take their sum, just like in the Racket code

and similarly....

Organization of the code

The Racket and Java versions have basically the same 6 snippets of code, but they are grouped differently

organization in Racket

	Leaf	Node
leaf_sum		
leaf_max		
leaf_min		

organization in Java

	Leaf	Node
leaf_sum		
leaf_max		
leaf_min		

Key Points for Lesson 5.4

- objects are like structs
- classes are like structure definitions, but with methods (functions) as well as fields
- To invoke a method of some object, send a message to the object.
- the interface of an object is the set of messages to which it responds
- interfaces correspond to data definitions
- Racket code and Java code are similar, but grouped differently

Next Steps

- Study the files 05-4-1-classes.java, 05-4-2-interfaces.java, and 05-4-3-javatrees.java in the Examples folder.
- If you have questions about this lesson, ask them on the Discussion Board
- Go on to the next lesson