Eric Roberts
CS 106A

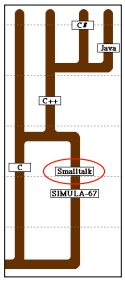
Handout #23
April 20, 2012

Objects and Classes

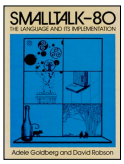
Objects and Classes

Eric Roberts
CS 106A
April 20, 2012

The Smalltalk Language



In the history of programming languages I outlined earlier, the language most responsible for bringing object-oriented ideas into the United States was Smalltalk, which was developed by a team at Xerox PARC that included a computer scientist named Adele Goldberg, whose *Smalltalk-80* book became the major reference work on the language.



A Tale of Two Companies

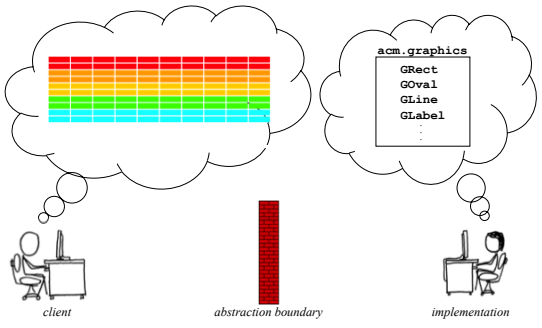
- The emergence of modern computing has a lot to do with the history of two companies in Silicon Valley: Xerox and Apple.
- In 1970, Xerox established the Palo Alto Research Center, which attracted many of the top computer scientists of the day. That lab pioneered many of the fundamental technologies of modern computing such as graphical user interfaces and object-oriented programming long before they became commonplace.
- When those technologies did appear, it was not from Xerox. . . .

Smalltalk and Steve Jobs

Review: Objects and Classes

- An ***object*** is a conceptually integrated data collection that encapsulates ***state*** and ***behavior***.
- A ***class*** is a template that defines the common structure for all objects of that class.
- Each object is created as an ***instance*** of one particular class, but a class can serve as a template for many instances.
- Classes form a hierarchy in which ***subclasses*** can inherit the behavior of their ***superclasses***.
- A ***constructor*** is a specialized method that acts as a factory for making new instances of a particular class. In Java, a constructor always has the same name as the class and is invoked by using the `new` keyword.

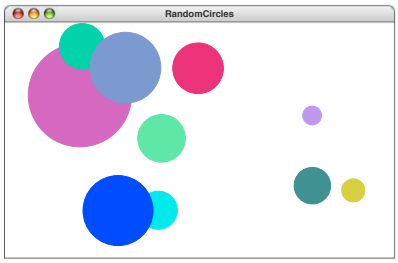
Thinking About Objects



with thanks to Randall Monroe at `xkcd.com`

Exercise: Random Circles

Write a **GraphicsProgram** that draws a set of ten circles with randomly chosen sizes, positions, and colors, subject to the condition that the entire circle must fit inside the window.



Defining Your Own Classes

- The standard form of a class definition in Java looks like this:

```
public class name extends superclass {
   class body
}
```

  Each public class must be stored in a separate file whose name is the name of the class followed by **.java**.

- The **extends** clause on the header line specifies the name of the superclass. If the **extends** clause is missing, the new class becomes a direct subclass of **Object**, which is the root of Java's class hierarchy.
- The body of a class consists of a set of Java definitions that are generically called ***entries***. The most common entries are constructors, methods, instance variables, and constants.

Representing Student Information

- Understanding the structure of a class is easiest in the context of a specific example. The next four slides walk through the definition of a class called **Student**, which is used to keep track of the following information about a student:
  - The name of the student
  - The student's six-digit identification number
  - The number of units the student has earned
  - A flag indicating whether the student has paid all university fees
- Each of these values is stored in an instance variable of the appropriate type.
- In keeping with the modern object-oriented convention used throughout both the book and the ACM Java Libraries, these instance variables are declared as **private**. All access to these values is therefore mediated by methods exported by the **Student** class.

The Student Class

```
/**
 * The Student class keeps track of the following pieces of data
 * about a student: the student's name, ID number, the number of
 * units the student has earned toward graduation, and whether
 * the student is paid up with respect to university bills.
 * All of this information is entirely private to the class.
 * Clients can obtain this information only by using the various
 * methods defined by the class.
 */

public class Student {

/**
 * Creates a new Student object with the specified name and ID.
 * @param name The student's name as a String
 * @param id The student's ID number as an int
 */
   public Student(String name, int id) {
      studentName = name;
      studentID = id;
   }
```

The Student Class

```
/**
 * Gets the name of this student.
 * @return The name of this student
 */
   public String getName() {
      return studentName;
   }

/**
 * Gets the ID number of this student.
 * @return The ID number of this student
 */
   public int getID() {
      return studentID;
   }

/**
 * Sets the number of units earned.
 * @param units The new number of units earned
 */
   public void setUnits(int units) {
      unitsEarned = units;
   }
```

The Student Class

```
/**
 * Gets the number of units earned.
 * @return The number of units this student has earned
 */
   public int getUnits() {
      return unitsEarned;
   }

/**
 * Sets whether the student is paid up.
 * @param flag The value true or false indicating paid-up status
 */
   public void setPaidUp(boolean flag) {
      paidUp = flag;
   }

/**
 * Returns whether the student is paid up.
 * @return Whether the student is paid up
 */
   public boolean isPaidUp() {
      return paidUp;
   }
```

The `Student` Class

```
/**
 * Creates a string identifying this student.
 * @return The string used to display this student
 */
   public String toString() {
      return studentName + " (#" + studentID + ")";
   }

/* Public constants */

/** The number of units required for graduation */
   public static final int UNITS_TO_GRADUATE = 180;

/* Private instance variables */
   private String studentName;    /* The student's name          */
   private int studentID;         /* The student's ID number     */
   private int unitsEarned;       /* The number of units earned  */
   private boolean paidUp;        /* Whether student is paid up  */

}
```

Using the `Student` Class

- Once you have defined the `Student` class, you can then use its constructor to create instances of that class. For example, you could use the following code to create two `Student` objects:

```
Student chosenOne = new Student("Harry Potter", 123456);
Student topStudent = new Student("Hermione Granger", 314159);
```

- You can then use the standard receiver syntax to call methods on these objects. For example, you could set Hermione's number-of-units field to 263 by writing

```
topStudent.setUnits(263);
```

or get Harry's full name by calling

```
chosenOne.getName();
```

Rational Numbers

- As a more elaborate example of class definition, section 6.4 defines a class called `Rational` that represents ***rational numbers***, which are simply the quotient of two integers.
- Rational numbers can be useful in cases in which you need exact calculation with fractions. Even if you use a `double`, the floating-point number 0.1 is represented internally as an approximation. The rational number 1 / 10 is exact.
- Rational numbers support the standard arithmetic operations:

Addition: $\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$

Multiplication: $\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$

Subtraction: $\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$

Division: $\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}$

Implementing the `Rational` Class

- The next five slides show the code for the `Rational` class along with some brief annotations.
- As you read through the code, the following features are worth special attention:
  - *The constructors for the class are overloaded.* Calling the constructor with no argument creates a `Rational` initialized to 0, calling it with one argument creates a `Rational` equal to that integer, and calling it with two arguments creates a fraction.
  - *The constructor makes sure that the number is reduced to lowest terms.* Moreover, since these values never change once a new `Rational` is created, this property will remain in force.
  - *Operations are specified using the receiver syntax.* When you apply an operator to two `Rational` values, one of the operands is the receiver and the other is passed as an argument, as in

```
r1.add(r2)
```

The `this` Keyword

- The implementation of the `Rational` class makes use of the Java keyword `this`, which always refers to the current object. The code, however, uses it in two distinct ways:
  1. In the first two versions of the constructor, the keyword `this` is used to invoke one of the other constructors for this class.
  2. In the methods that implement the arithmetic operators, `this` is used to emphasize that a named field comes from this object rather than from one of the other values of that class.
- The keyword `this` is also often used to distinguish between an instance variable and a local variable with the same name. If you include `this` before the variable name, you indicate to the compiler that you mean the instance variable.

The `Rational` Class

```
/**
 * The Rational class is used to represent rational numbers, which
 * are defined to be the quotient of two integers.
 */
public class Rational {

/** Creates a new Rational initialized to zero. */
   public Rational() {
      this(0);
   }

/**
 * Creates a new Rational from the integer argument.
 * @param n The initial value
 */
   public Rational(int n) {
      this(n, 1);
   }
```

The `Rational` Class

```
/**
 * Creates a new Rational with the value x / y.
 * @param x The numerator of the rational number
 * @param y The denominator of the rational number
 */
   public Rational(int x, int y) {
      int g = gcd(Math.abs(x), Math.abs(y));
      num = x / g;
      den = Math.abs(y) / g;
      if (y < 0) num = -num;
   }

/**
 * Adds the rational number r to this one and returns the sum.
 * @param r The rational number to be added
 * @return The sum of the current number and r
 */
   public Rational add(Rational r) {
      return new Rational(this.num * r.den + r.num * this.den,
                          this.den * r.den);
   }
```

The `Rational` Class

```
/**
 * Subtracts the rational number r from this one.
 * @param r The rational number to be subtracted
 * @return The result of subtracting r from the current number
 */
   public Rational subtract(Rational r) {
      return new Rational(this.num * r.den - r.num * this.den,
                          this.den * r.den);
   }

/**
 * Multiplies this number by the rational number r.
 * @param r The rational number used as a multiplier
 * @return The result of multiplying the current number by r
 */
   public Rational multiply(Rational r) {
      return new Rational(this.num * r.num, this.den * r.den);
   }
```

The `Rational` Class

```
/**
 * Divides this number by the rational number r.
 * @param r The rational number used as a divisor
 * @return The result of dividing the current number by r
 */
   public Rational divide(Rational r) {
      return new Rational(this.num * r.den, this.den * r.num);
   }

/**
 * Creates a string representation of this rational number.
 * @return The string representation of this rational number
 */
   public String toString() {
      if (den == 1) {
         return "" + num;
      } else {
         return num + "/" + den;
      }
   }
```

The `Rational` Class

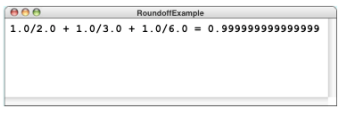
```
/**
 * Calculates the greatest common divisor using Euclid's algorithm.
 * @param x First integer
 * @param y Second integer
 * @return The greatest common divisor of x and y
 */
   private int gcd(int x, int y) {
      int r = x % y;
      while (r != 0) {
         x = y;
         y = r;
         r = x % y;
      }
      return y;
   }

/* Private instance variables */
   private int num;    /* The numerator of this Rational   */
   private int den;    /* The denominator of this Rational */
}
```

Simulating Rational Calculation

- The next slide works through all the steps in the calculation of a simple program that adds three rational numbers.

$$\frac{1}{2} + \frac{1}{3} + \frac{1}{6}$$

- With rational arithmetic, the computation is exact. If you write this same program using variables of type **`double`**, the result looks like this:



Adding Three Rational Values

```
public void run() {
   Rational a = new Rational(1, 2);
   Rational b = new Rational(1, 3);
   Rational c = new Rational(1, 6);
   Rational sum = a.add(b).add(c);
   println(a + " + " + b + " + " + c + " = " + sum);
}
```