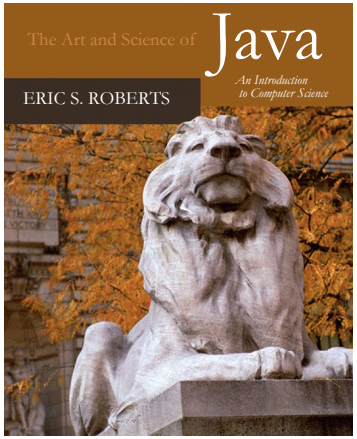
Five-Minute Review

1. What is a *method*? A *static method*?
2. What is the motivation for having methods?
3. What role do methods serve in expressions?
4. What are the mechanics of method calling?
5. What are *local variables*?

Programming – Lecture 6

Objects and Classes (Chapter 6)

- Local/instance/class variables, constants
- Using existing classes: **`RandomGenerator`**
- The implementor's perspective
- Javadoc: The client's perspective
- Defining your own classes



CHAPTER 6

Objects and Classes

To beautify life is to give it an object.

—José Martí, *On Oscar Wilde*, 1888

6.1 Using the `RandomGenerator` class

6.2 The **javadoc** documentation system

6.3 Defining your own classes

6.4 Representing student information

6.5 Rational numbers

6.6 Extending existing classes

Local/Instance/Class Variables

(See also Lec. 03)

```
public class Example {
   int someInstanceVariable;
   static int someClassVariable;
   static final double PI = 3.14;

   public void run() {
      int someLocalVariable;
      ...
   }
}
```

Local Variables

- Declared within method
- One storage location ("box") **per method invocation**
- Stored on *stack*

```
public void run() {
   int someLocalVariable;
   ...
}
```

Instance Variables

- Declared outside method
- One storage location **per object**
- A.k.a. *ivars*, *member variables*, or *fields*
- Stored on *heap*

```
int someInstanceVariable;
```

Class Variables

- Declared outside method, with **`static`** modifier
- Only **one storage location**, for all objects
- Stored in *static data segment*

```
static int someClassVariable;
static final double PI = 3.14;
```

Constants

- Are typically stored in class variables
- **`final`** indicates that these are not modified

```
static final double PI = 3.14;
```

`this`

- **`this`** refers to current object
- May use **`this`** to override *shadowing* of ivars by local vars of same name

```
public class Point {
  public int x = 0, y = 0;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

- **Coding Advice:** re-use var. names in constructors and setters
  (even though examples in book don't always do this ...)

Coding Advice – Getters and Setters

- A *setter* sets the value of an ivar
- Should be named **`setVarName`**

```
public void setX(int x) {
  this.x = x;
}
```

- A *getter* returns the value of an ivar
- Should be named **`getVarName`**, except for boolean ivars, which should be named **`isVarName`**

```
public int getX() {
  return x;
}
```

Coding Advice – Getters and Setters

- To abstract from class implementation, *clients* of a class should access object state only through getters and setters
- *Implementers* of a class can access state directly
- Eclipse can automatically generate generic constructors, getters, setters
- However, should create only those getters/setters that clients really need

Using the `RandomGenerator` Class

- Before you start to write classes of your own, it helps to look more closely at how to use classes that have been developed by others. Chapter 6 illustrates the use of existing classes by introducing a class called **`RandomGenerator`**, which makes it possible to write programs that simulate random processes such as flipping a coin or rolling a die. Programs that involve random processes of this sort are said to be **nondeterministic**.
- Nondeterminstic behavior is essential to many applications. Computer games would cease to be fun if they behaved in exactly the same way each time. Nondeterminism also has important practical uses in simulations, in computer security, and in algorithmic research.

Creating a Random Generator

- The first step in writing a program that uses randomness is to create an instance of the **`RandomGenerator`** class.
- In most cases, you create a new instance of a class by using the **`new`** operator, as you have already seen in the earlier chapters. From that experience, you would expect to create a **`RandomGenerator`** object by writing a declaration like this:

```
RandomGenerator rgen = new RandomGenerator();
```

For reasons that will be discussed in a later slide, using **`new`** is not appropriate for **`RandomGenerator`** because there should be only one random generator in an application. What you want to do instead is to ask the **`RandomGenerator`** class for a common instance that can be shared throughout all classes in your program.

Creating a Random Generator

- The best way to create a **`RandomGenerator`** instance is to call the **`getInstance`** method, which returns a single shared instance of a random generator. The standard form of that declaration looks like this:

```
private RandomGenerator rgen = RandomGenerator.getInstance();
```

- This declaration usually appears outside of any method and is therefore an example of an **instance variable**. The keyword **`private`** indicates that this variable can be used from any method within this class but is not accessible to other classes.
- When you want to obtain a random value, you send a message to the generator in **`rgen`**, which then responds with the result.

Creating a Random Generator

```
private RandomGenerator rgen =
    new RandomGenerator();
```

```
private RandomGenerator rgen =
    RandomGenerator.getInstance();
```

`RandomGenerator` Class

`int nextInt(int low, int high)` Returns a random **`int`** between **`low`** and **`high`**, inclusive.
`int nextInt(int n)` Returns a random **`int`** between 0 and **`n`** − 1.
`double nextDouble(double low, double high)` Returns a random **`double`** d in the range **`low`** $\le d <$ **`high`**.
`double nextDouble()` Returns a random **`double`** d in the range $0 \le d < 1$.
`boolean nextBoolean()` Returns a random **`boolean`** value, which is **`true`** 50 percent of the time.
`boolean nextBoolean(double p)` Returns a random **`boolean`**, which is **`true`** with probability **`p`**, where $0 \le$ **`p`** ≤ 1.
`Color nextColor()` Returns a random color.

Aside: Polymorphism

Definitions vary, but we here distinguish

- Static polymorphism
  - Method overloading
- Dynamic polymorphism
  - Method overriding
- Parametric polymorphism
  - Generics (see later)

https://docs.oracle.com/javase/tutorial/java/IandI/override.html
https://en.wikipedia.org/wiki/Polymorphism_(computer_science)
https://docs.oracle.com/javase/tutorial/java/IandI/override.html
https://www.sitepoint.com/quick-guide-to-polymorphism-in-java/

Static Polymorphism

- Method *overloading*
- Methods of same name but with different parameters
- Aka *static binding*

```
boolean nextBoolean()
boolean nextBoolean(double p)
```

Dynamic Polymorphism

- Method *overriding*
- Subclass implements method of same signature, i.e. same name and with same parameters, as in superclass
- Aka *dynamic binding*
- For static methods: method *hiding*

`toString()`

- is implemented in **`java.lang.Object`**
- may be overridden, e.g. to change how object is printed by **`println`**

Using the Random Methods

- To use the methods from the previous slide in a program, all you need to do is call that method using **`rgen`** as the receiver.
- As an example, you could simulate rolling a die by calling

```
int die = rgen.nextInt(1, 6);
```

- Similarly, you could simulate flipping a coin like this:

```
String flip =
    rgen.nextBoolean() ? "Heads" : "Tails";
```

- Note that the **`nextInt`**, **`nextDouble`**, and **`nextBoolean`** methods all exist in more than one form. Java can tell which version of the method you want by checking the number and types of the arguments. Methods that have the same name but differ in their argument structure are said to be **overloaded**.

Exercises

1. Set the variable **`total`** to the sum of two 6-sided dice.

```
int d1 = rgen.nextInt(1, 6);
int d2 = rgen.nextInt(1, 6);
int total = d1 + d2;
```

Exercises

2. Flip a weighted coin that comes up heads 60% of the time.

```
String flip =
  rgen.nextBoolean(0.6) ? "Heads" : "Tails";
```

Exercises

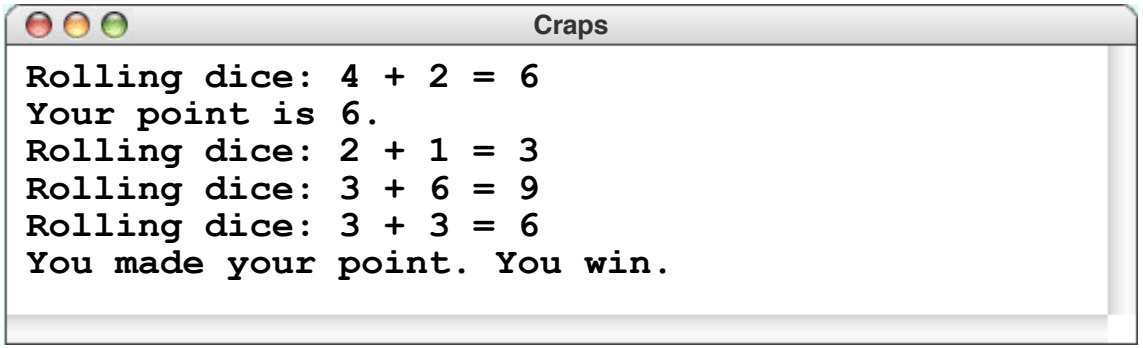
3. Change the fill color of rect to some randomly generated color.

```
rect.setFillColor(rgen.nextColor());
```

Simulating the Game of Craps

```
public void run() {
   int total = rollTwoDice();
   if (total == 7 || total == 11) {
      println("That's a natural.  You win.");
   } else if (total == 2 || total == 3 || total == 12) {
      println("That's craps.  You lose.");
   } else {
      int point = total;
      println("Your point is " + point + ".");
      while (true) . . .
   }
}
```

point	total
6	6



skip simulation

Two Perspectives

1. Implementor
   "How does this thing work internally?"

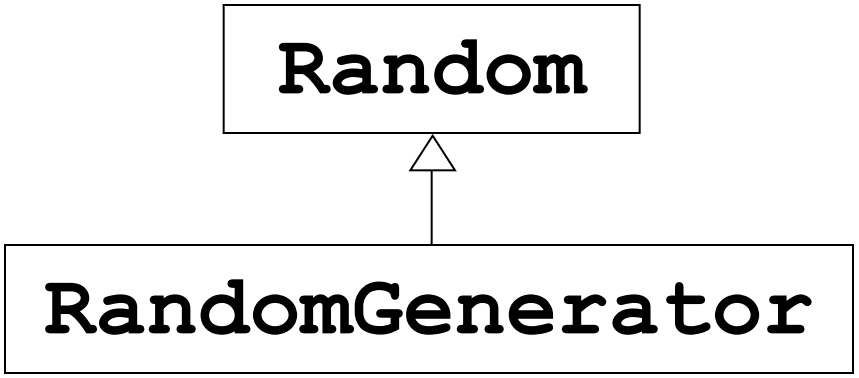
2. Client
   "How do I use this thing?"

Information Hiding ➡ Similar to methods!

Clients and Implementors

- As you work with classes in Java, it is useful to recognize that there are two contrasting perspectives that you can take with respect to a particular class. More often than not, you will find yourself using a class that you didn't actually write, as in the case of the **`RandomGenerator`** class. In such cases, you are acting as a **client** of the class. When you actually write the code, you are then acting as as an **implementor**.
- Clients and implementors look at a class in different ways. Clients need to know what methods are available in a class and how to call them. The details of how each method works are of little concern. The implementor, on the other hand, is primarily interested in precisely those details.
- As was true in the case of a method and its callers, the implementor of a class should try to hide complexity from its clients. The **`RandomGenerator`** class hides a considerable amount of complexity, as you will see on the next few slides.

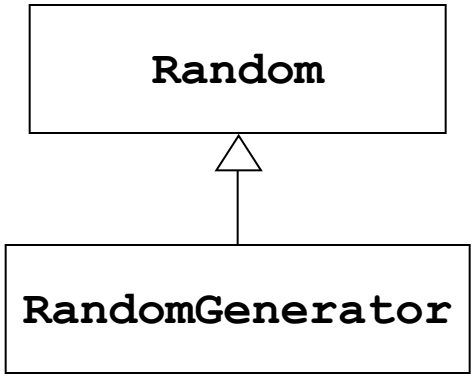
Class Hierarchy



- Clients don't care where methods are implemented
- This design is called a *Layered Abstraction*

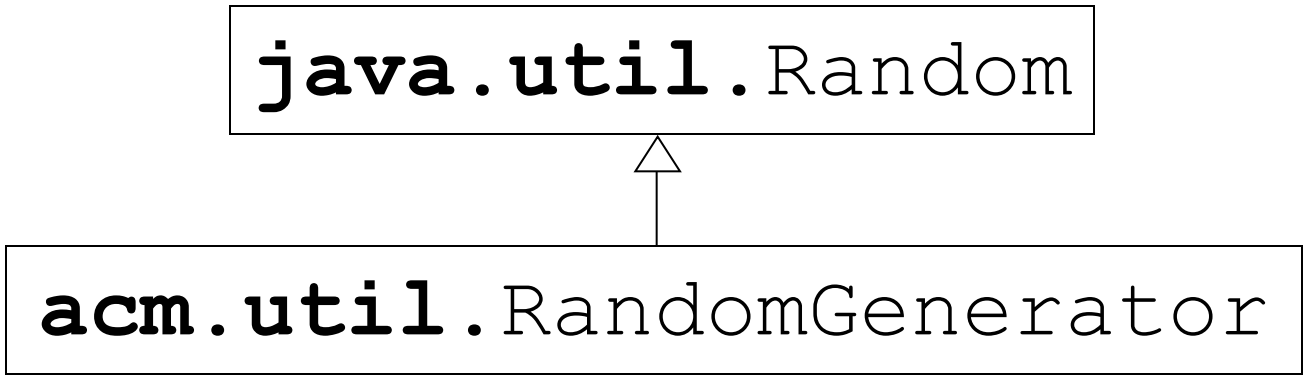
Layered Abstractions

- The **`RandomGenerator`** class is actually implemented as a subclass of a class called **`Random`**, as follows:



- Some of the methods that you call to produce random values are defined in the **`RandomGenerator`** class itself; others are inherited from the **`Random`** class. As a client, you don't need to know which is which.
- Class hierarchies that define methods at different levels are called **layered abstractions**.

Packages




```
import acm.util.RandomGenerator
```

Not:

```
import java.util.Random
```

Java Packages

- Every Java class is part of a **package**, which is a collection of related classes that have been released as a coherent unit.
- The **`RandomGenerator`** class is defined in a package called **`acm.util`**, which is part of the ACM Java Libraries.
- The **`Random`** class is part of the **`java.util`** package, which is a collection of general utility classes.
- Whenever you refer directly to a class, you must import the package in which it lives. For example, any program using the **`RandomGenerator`** class will include the line

```
import acm.util.*;
```

When you use the **`RandomGenerator`** class, you do not need to import the **`java.util`** package (unless you use it for some other purpose). The fact that **`RandomGenerator`** is built on top of **`Random`** is part of the complexity hidden from clients.

Simulating Randomness

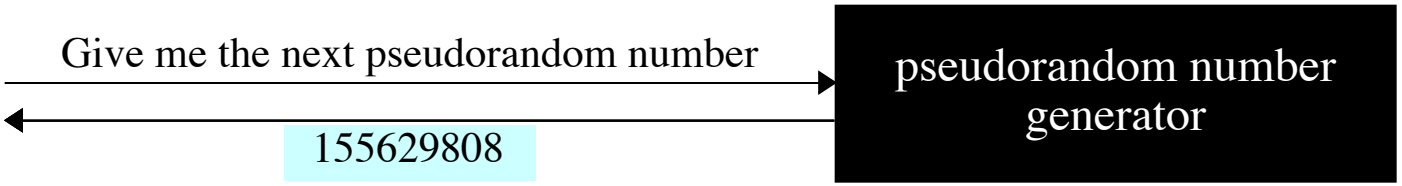
- Computers are not random
  - ➟ Pseudorandom numbers
- Initialized with a *seed value*
- Explicit seed:
  `setSeed(long seed)`

Simulating Randomness

- Nondeterministic behavior turns out to be difficult to achieve on a computer. A computer executes its instructions in a precise, predictable way. If you give a computer program the same inputs, it will generate the same outputs every time, which is not what you want in a nondeterministic program.
- Given that true nondeterminism is so difficult to achieve in a computer, classes such as **`RandomGenerator`** must instead *simulate* randomness by carrying out a deterministic process that satisfies the following criteria:
  1. The values generated by that process should be difficult for human observers to predict.
  2. Those values should appear to be random, in the sense that they should pass statistical tests for randomness.
- Because the process is not truly random, the values generated by **`RandomGenerator`** are said to be **pseudorandom**.

Pseudorandom Numbers

- The **`RandomGenerator`** class uses a mathematical process to generate a series of integers that, for all intents and purposes, appear to be random. The code that implements this process is called a **pseudorandom number generator**.
- The best way to visualize a pseudorandom number generator is to think of it as a black box that generates a sequence of values, even though the details of how it does so are hidden:



- To obtain a new pseudorandom number, you send a message to the generator asking for the next number in its sequence.
- The generator then responds by returning that value.
- Repeating these steps generates a new value each time.

The Random Number Seed

- The pseudorandom number generator used by the **`Random`** and **`RandomGenerator`** classes generates seemingly random values by applying a function to the previous result. The starting point for this sequence of values is called the **seed**.
- As part of the process of starting a program, Java initializes the seed for its pseudorandom number generator to a value based on the system clock, which changes very quickly on a human time scale. Programs run just a few milliseconds apart will therefore get a different sequence of random values.
- Computers, however, run much faster than the internal clock can register. If you create two **`RandomGenerator`** instances in a single program, it is likely that both will be initialized with the same seed and therefore generate the same sequence of values. This fact explains why it is important to create only one **`RandomGenerator`** instance in an application.

Debugging and Random Behavior

- Even though unpredictable behavior is essential for programs like computer games, such unpredictability often makes debugging extremely difficult. Because the program runs in a different way each time, there is no way to ensure that a bug that turns up the first time you run a program will happen again the second time around.
- To get around this problem, it is often useful to have your programs run deterministically during the debugging phase. To do so, you can use the **`setSeed`** method like this:

```
rgen.setSeed(1);
```

This call sets the random number seed so that the internal random number sequence will always begin at the same point. The value 1 is arbitrary. Changing this value will change the sequence, but the sequence will still be the same on each run.

Two Perspectives

1. Implementor
   "How does this thing work internally?"

2. Client
   "How do I use this thing?"

Information Hiding ➡ Similar to methods!

The javadoc Documentation System

- Unlike earlier languages that appeared before the invention of the World-Wide Web, Java was designed to operate in the web-based environment. From Chapter 1, you know that Java programs run on the web as applets, but the extent of Java's integration with the web does not end there.
- One of the most important ways in which Java works together with the web is in the design of its documentation system, which is called **javadoc**. The **javadoc** application reads Java source files and generates documentation for each class.
- The next few slides show increasingly detailed views of the **javadoc** documentation for the **`RandomGenerator`** class.
- You can see the complete documentation for the ACM Java Libraries by clicking on the following link:

`http://jtf.acm.org/javadoc/student/`

Sample javadoc Pages

Overview Package Student Complete Tree Index Help

PREV CLASS NEXT CLASS
SUMMARY: FIELD | CONSTR | METHOD

FRAMES NO FRAMES
DETAIL: FIELD | CONSTR | METHOD

acm.util

Class RandomGenerator

```
java.lang.Object
  |
  +--java.util.Random
        |
        +--acm.util.RandomGenerator
```

`public class RandomGenerator extends Random`

This class implements a simple random number generator that allows clients to generate pseudorandom integers, doubles, booleans, and colors. To use it, the first step is to declare an instance variable to hold the random generator as follows:

```
private RandomGenerator rgen = RandomGenerator.getInstance();
```

By default, the **`RandomGenerator`** object is initialized to begin at an unpredictable point in a pseudorandom sequence. During debugging, it is often useful to set the internal seed for the random generator explicitly so that it always returns the same sequence. To do so, you need to invoke the **`setSeed`** method.

The **`RandomGenerator`** object returned by **`getInstance`** is shared across all classes in an application. Using this shared instance of the generator is preferable to allocating new instances of **`RandomGenerator`**. If you create several random generators in succession, they will typically generate the same sequence of values.

Sample javadoc Pages

Constructor Summary

`RandomGenerator()` Creates a new random generator.

Method Summary

`RandomGenerator`	**`getInstance()`** Returns a **`RandomGenerator`** instance that can be shared among several classes.
`boolean`	**`nextBoolean(double p)`** Returns a random **`boolean`** value with the specified probability.
`Color`	**`nextColor()`** Returns a random opaque color whose components are chosen uniformly in the 0-255 range.
`double`	**`nextDouble(double low, double high)`** Returns the next random real number in the specified range.
`int`	**`nextInt(int low, int high)`** Returns the next random integer in the specified range.

Inherited Method Summary

`boolean`	**`nextBoolean()`** Returns a random **`boolean`** that is **`true`** 50 percent of the time.
`double`	**`nextDouble()`** Returns a random **`double`** d in the range $0 \le d < 1$.
`int`	**`nextInt(int n)`** Returns a random **`int`** k in the range $0 \le k <$ **`n`**.
`void`	**`setSeed(long seed)`** Sets a new starting point for the random number generator.

Sample javadoc Pages

Constructor Detail

`public RandomGenerator()`

Creates a new random generator. Most clients will not use the constructor directly but will instead call `getInstance` to obtain a `RandomGenerator` object that is shared by all classes in the application.

Usage: `RandomGenerator rgen = new RandomGenerator();`

Method Detail

`public RandomGenerator()`

Returns a `RandomGenerator` instance that can be shared among several classes.

Usage: `RandomGenerator rgen = RandomGenerator.getInstance();`
Returns: A shared `RandomGenerator` object

`public boolean nextBoolean(double p)`

Returns a random `boolean` value with the specified probability. You can use this method to simulate an event that occurs with a particular probability. For example, you could simulate the result of tossing a coin like this:

```
String coinFlip = rgen.nextBoolean(0.5) ? "HEADS" : "TAILS";
```

Usage: `if (rgen.nextBoolean(p)) ...`
Parameter: `p` A value between 0 (impossible) and 1 (certain) indicating the probability
Returns: The value `true` with probability `p`

Writing javadoc Comments

- The **javadoc** system is designed to create the documentary web pages automatically from the Java source code. To make this work with your own programs, you need to add specially formatted comments to your code.
- A **javadoc** comment begins with the characters **`/**`** and extends up to the closing **`*/`** just as a regular comment does. Although the compiler ignores these comments, the **javadoc** application reads through them to find the information it needs to create the documentation.
- Although **javadoc** comments may consist of simple text, they may also contain formatting information written in HTML, the hypertext markup language used to create web pages. The **javadoc** comments also often contain **`@param`** and **`@result`** tags to describe parameters and results, as illustrated on the next slide.

Writing Javadoc Comments

```
/**
 * Returns the next random integer between 0 and
 * {@code n}-1, inclusive.
 *
 * @param n The number of integers in the range
 * @return A random integer between 0 and {@code n}-1
 */
public int nextInt(int n)
```

`public int nextInt(int n)`

Returns the next random integer between 0 and `n`-1, inclusive.

Parameter: `n` The number of integers in the range
Returns: A random integer between 0 and `n`-1

Note: Eclipse can automatically generate Javadoc templates ("Generate Element Comment").
These must of course still be filled with content!

Aside: What is **`null`**?

- Variables with primitive type have to have a value before being used.
  `char, byte, short, int, long, float, double, boolean`

- Variables with object type don't.

```
Wubbel myWubbel = new Wubbel();
Wubbel noWubbel = null;
if (noWubbel != null) …
```

Defining Classes

`public class` *name* **`[extends`** *superclass* **`]`** **`{`**

class body

`}`

Class body has following types of *entries*:

- Class var's, constants
- Instance variables
  Object state
- Constructors
- Methods

Defining Your Own Classes

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

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

- 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 collection of Java definitions that are generically called **entries**. The most common entries are constructors, methods, instance variables, and named constants.

Access Control/Visibility for Entries

```
public int nextInt();
```

access modifier

`public`	Visible to everyone. ("*exported*")
`private`	Visible in same class only.
`protected`	Visible in same package and subclasses and subclasses thereof, etc.
(no keyword)	Visible in same package only, not in subclasses. ("*package-private*")

Coding advice: make visibilities as restrictive as possible, preferably **`private`**

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
(default)	Y	Y	N	N
`private`	Y	N	N	N

https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

```
public int publicIvar;
protected int protectedIvar;
int packagePrivateIvar;
private int privateIvar;
```

Controlling Access to Entries

- Each entry in a Java class is marked with one of the following keywords to control which classes have access to that entry:

`public`	All classes in the program have access to any public entry. The public entries in a class are said to be **exported** by that class.
`private`	Access to entries declared as private is limited to the class itself, making that entry completely invisible outside the class.
`protected`	Protected entries are restricted to the class that defines them, along with any of its subclasses or any classes in the same package.
(no keyword)	If the access keyword is missing, the entry is visible only to classes in the same package. Such entries are called **package-private**.

- The text uses only **`public`** and **`private`**. All entries are marked as **`private`** unless there is a compelling reason to export them.

Example: **`Student`** Class

Encapsulate these properties:

- ID
- Name
- Credit points
- Paid tuition fee?

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 credits the student has earned (which may include a decimal fraction to account for half- and quarter-credit courses)
  - 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

This comment describes the class as a whole.

```
/**
 * The Student class keeps track of the following pieces of data
 * about a student: the student's name, ID number, the number of
 * credits 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 {
```

The class header defines `Student` *as a direct subclass of* `Object`.

```
/**
 * 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;
}
```

This comment describes the constructor.

The constructor sets the instance variables.

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;
   }
```

*These methods retrieve the value of an instance variable and are called **getters**. Because the student name and ID number are fixed, there are no corresponding setters.*

*This method changes the value of an instance variable and is called a **setter**.*

```
/**
 * Sets the number of credits earned.
 * @param credits The new number of credits earned
 */
   public void setCredits(double credits) {
      creditsEarned = credits;
   }
```

The `Student` Class

```
/**
 * Gets the number of credits earned.
 * @return The number of credits this student has earned
 */
   public double getCredits() {
      return creditsEarned;
   }

/**
 * 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;
   }
```

Names for getter methods usually begin with the prefix **`get`**. *The only exception is for getter methods that return a* **`boolean`**, *in which case the name typically begins with* **`is`**.

The `Student` Class

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

The `toString` *method tells Java how to display a value of this class. All of your classes should override* `toString`.

```
/* Public constants */

/** The number of credits required for graduation */
   public static final double CREDITS_TO_GRADUATE = 32.0;
```

Classes often export named constants.

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

}
```

These declarations define the instance variables that maintain the internal state of the class. All instance variables used in the text are private.

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-credits field to 97 by writing

```
topStudent.setCredits(97);
```

or get Harry's full name by calling

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

A Class Design Strategy

1. Which instance variables do I need?
2. Which of them can be changed?
3. Which constructors make sense?
4. Which methods do I need?

Example: **`Employee`** class

Download this presentation to see the next few slides, not shown in class

Exercise: Design an `Employee` Class

- Create a definition for a class called **`Employee`**, which keeps track of the following information:
  - The name of the employee
  - A number indicating the order in which this employee was hired
  - A flag indicating whether the employee is still active
  - The salary (a number that may contain a decimal fraction)
- The name and employee number should be assigned as part of the constructor call, and it should not be possible to change them subsequently. By default, new employees should be marked as active. The salary field need not be initialized.
- The class should export appropriately named getters for all four fields and setters for the last two.

The **Employee** Class

```
/**
 * The Employee class keeps track of the following pieces of
 * data about an employee: the name, employee number, whether
 * the employee is active, and the annual salary.
 */

public class Employee {

/**
 * Creates a new Employee object with the specified name and
 * employee number.
 * @param name The employee's name as a String
 * @param id The employee number as an int
 */
   public Employee(String name, int id) {
      employeeName = name;
      employeeNumber = id;
      active = true;
   }
```

The **Employee** Class

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

/**
 * Gets the employee number of this employee.
 * @return The employee number of this employee
 */
   public int getEmployeeNumber() {
      return employeeNumber;
   }

/**
 * Sets whether the employee is active.
 * @param flag The value true or false indicating active status
 */
   public void setActive(boolean flag) {
      active = flag;
   }
```

The **Employee** Class

```
/**
 * Returns whether the employee is active.
 * @return Whether the employee is active
 */
   public boolean isActive() {
      return active;
   }

/**
 * Sets the employee's salary.
 * @param salary The new salary
 */
   public void setSalary(double salary) {
      annualSalary = salary;
   }

/**
 * Gets the annual salary for this employee.
 * @return The annual salary for this employee works
 */
   public double getSalary() {
      return annualSalary;
   }
```

The **Employee** Class

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

/* Private instance variables */
   private String employeeName; /* The employee's name             */
   private int employeeNumber;  /* The employee number             */
   private boolean active;      /* Whether the employee is active */
   private double annualSalary; /* The annual salary               */

}
```

Exercises

```
Employee founder = new
    Employee("Ebenezer Scrooge", 1);
Employee partner = new
    Employee("Jacob Marley", 2);
Employee clerk = new
    Employee("Bob Cratchit", 3);
```

1. Mark Jacob Marley as inactive.

```
partner.setActive(false);
```

2. Double Bob Cratchit's salary.

```
clerk.setSalary(2 * clerk.getSalary());
```

Exercise: Using the **`Employee`** Class

- Now that you have defined **`Employee`**, write declarations for three variables that contain the names of the following three employees: Ebenezer Scrooge (employee #1), Jacob Marley (employee #2), and Bob Cratchit (employee #3).

```
Employee founder = new Employee("Ebenezer Scrooge", 1);
Employee partner = new Employee("Jacob Marley", 2);
Employee clerk = new Employee("Bob Cratchit", 3);
```

- Using these variables, write a Java statement that marks the **`Employee`** instance for Jacob Marley as inactive.

```
partner.setActive(false);
```

- Write a Java statement that doubles Bob Cratchit's salary.

```
clerk.setSalary(2 * clerk.getSalary());
```

Example: `Rational` Class

Encapsulate these properties:

- Numerator
- Denominator

Provides these 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:

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

Note: can view this as specification of an ADT

A Rationale for `Rational`

Math: $\frac{1}{2} + \frac{1}{3} + \frac{1}{6} = 1$

Java:

```
1.0/2.0 + 1.0/3.0 + 1.0/6.0 = 0.999999999999999
```

Even worse:

```
0.1 + 0.1 + 0.1 = 0.30000000000000004
```

Wait a minute ...

```
public class RationaleForRational extends ConsoleProgram
{
  public void run() {
    double oneHalf = 1.0 / 2.0;
    double oneThird = 1.0 / 3.0;
    double oneSixth = 1.0 / 6.0;
    double oneTenth = 1.0 / 10.0;

    double threeThirds = oneThird + oneThird + oneThird;
    println("threeThirds = " + threeThirds);
          // Output: "threeThirds = 1.0"
    double sixSixths = oneHalf + oneThird + oneSixth;
    println("sixSixths = " + sixSixths);
          // Output: "sixSixths = 0.9999999999999999"
    double threeTenths = oneTenth + oneTenth + oneTenth;
    println("threeTenths = " + threeTenths);
          // Output: "threeTenths = 0.30000000000000004"
  }
}
```

IEEE 754 Floating Point

Numerical Form: $-1^{s}\ M\ \ 2^{E}$

- Sign bit *s*
- Significand *M* normally fractional value in [1.0,2.0)
- Exponent *E* weighs value by power of two

Encoding

- **`s`** is sign bit
- **`exp`** field encodes *E*
- **`frac`** field encodes *M*

For much more detail, see https://en.wikipedia.org/wiki/IEEE_754

$1.0000\ 0000\ 0000_{16} / A_{16} =$
$0.1999\ 9999\ 9999_{16}$

Computers (usually) cannot represent repeating decimals (such as $0.\overline{3}_{10}$)

Computers (usually) cannot represent repeating binaries either (such as $0.\overline{1}_{2}$)

Some non-repeating decimals (such as 0.1_{10}) correspond to repeating binaries ($0.\overline{00011}_{2}$);

thus computers cannot (easily) represent 0.1!

How about the converse? (Exercise)

Coding Advice – Floating Point

```
double x = 0, max = 5, step = 0.1;
do {
    x = x + step;
    println("Applied " + x + " x-ray units.");
} while (x != max);
```

WARNING: this would never terminate!

Use instead: `while (x <= max)`

In general, avoid (in-)equality checks with floating point, use <= or >= instead!

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 numerator and denominator of any **`Rational`** are always reduced to lowest terms. Moreover, since these values never change once a new **`Rational`** is created, this property will remain in force.
  - The **`add`**, **`subtract`**, **`multiply`**, and **`divide`** methods are written so that one of the operands is the receiver (signified by the keyword **`this`**) and the other is passed as an argument. Thus to add **`r1`** and **`r2`** you would write:

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

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);
   }
```

These constructors are overloaded so that there is more than one way to create a **`Rational`** *value. These two versions invoke the primary constructor by using the keyword* **`this`**.

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;
   }
```

The primary constructor creates a new `Rational` *from the numerator and denominator. The call to* `gcd` *ensures that the fraction is reduced to lowest terms.*

The `add` *method creates a new* `Rational` *object using the addition rule. The two rational values appear in* `this` *and* `r`.

```
/**
 * 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);
   }
```

These methods (along with **`divide`** *on the next page) work just like the* **`add`** *method but use a different formula. Note that these methods do have access to the components of* **`r`**.

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;
      }
   }
```

This method converts the `Rational` *number to its string form. If the denominator is 1, the number is displayed as an integer.*

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;
  }
```

Euclid's `gcd` *method is declared to be* `private` *because it is part of the implementation of this class and is never used outside of it.*

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

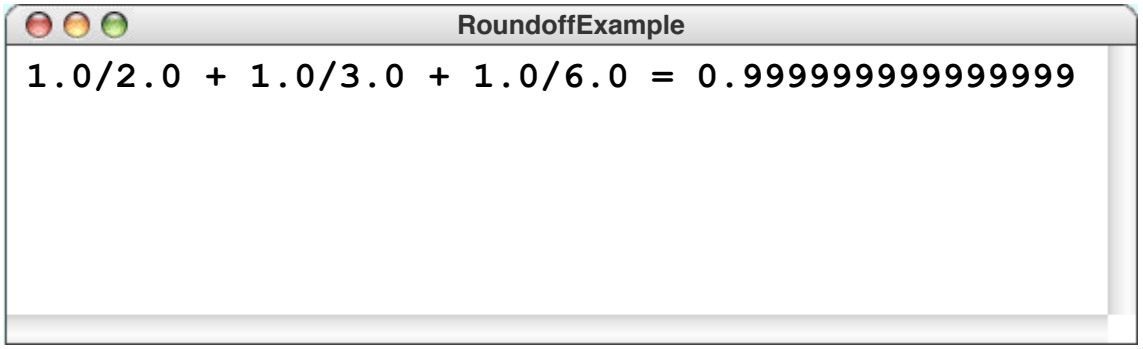
As always, the instance variables are private to this class.

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:



- The simulation treats the **`Rational`** values as abstract objects. Chapter 7 reprises the example showing the memory structure.

Adding Three Rational Values

```
public void run() {
```

```
public Rational add(Rational r) {
```

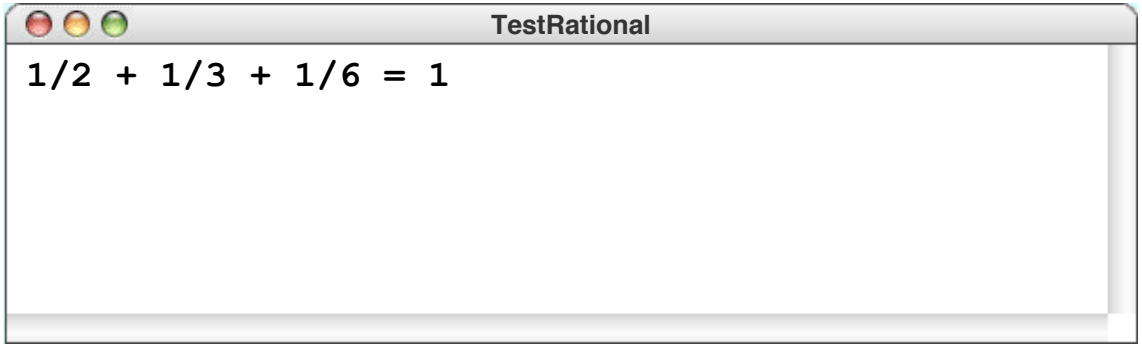
36

```
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;
}
```

this

num	1
den	1

x	y	g
36	36	36



skip simulation

Immutable Classes

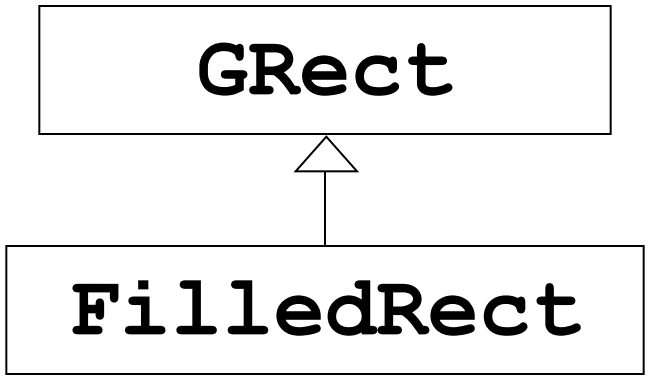
`Rational` is *immutable*

- No method can change internal state
- No setters
- Instance variables are private

Another immutable class: **`String`**

Not immutable classes are *mutable*

Extending Classes



`FilledRect` is filled by default

User can supply a fill color to the constructor

Constructors Calling ...

`super(...)` invokes constructor of superclass

`this(...)` invokes constructor of this class

If none of these calls are made, constructors implicitly call **`super()`**

Default constructor:

- is provided automatically if no other constructor is provided
- does nothing, except call **`super()`**

Extending Existing Classes

- The examples shown in the earlier slides have all extended the built-in **`Object`** class. More often than not, however, you will find that you want to extend an existing class to create a new class that inherits most of its behavior from its superclass but makes some small extensions or changes.
- Suppose, for example, that you wanted to define a new class called **`FilledRect`** that is similar to the **`GRect`** class, except that it is filled rather than outlined by default. Moreover, as a convenience, you would like to be able to specify an optional color for the rectangle as part of the constructor. Calling

```
add(new FilledRect(10, 10, 100, 75, Color.RED));
```

  for example, should create a 100x75 rectangle solidly filled in red and then add it to the canvas at the point (10, 10).
- The code for the **`FilledRect`** class appears on the next slide.

FilledRect

```
/**
 * This class is a GObject subclass that is almost identical
 * to GRect except that it starts out filled instead of outlined.
 */
public class FilledRect extends GRect {

/** Creates a new FilledRect with the specified bounds. */
   public FilledRect(double x, double y,
                     double width, double height) {
      super(x, y, width, height);
      setFilled(true);
   }

/** Creates a new FilledRect with the specified bounds and color. */
   public FilledRect(double x, double y,
                     double width, double height, Color color) {
      this(x, y, width, height);
      setColor(color);
   }

}
```

This syntax calls the superclass constructor.

This syntax calls another constructor in this class.

Rules for Inherited Constructors

- Whenever you create an object of an extended class, Java must call some constructor for the superclass object to ensure that its structure is correctly initialized.
- If the superclass does not define any explicit constructors, Java automatically provides a **default constructor** with an empty body.
- Java therefore invokes the superclass constructor in one of the following ways:
  - Classes that begin with an explicit call to **`this`** invoke one of the other constructors for this class, delegating responsibility to that constructor for making sure that the superclass constructor gets called.
  - Classes that begin with a call to **`super`** invoke the constructor in the superclass that matches the argument list provided.
  - Classes that begin with no call to either **`super`** or **`this`** invoke the default superclass constructor with no arguments.

Rules for Inherited Methods

- When one class extends another, the subclass is allowed to **override** method definitions in its superclass. Whenever you invoke that method on an instance of the extended class, Java chooses the new version of the method provided by that class and not the original version provided by the superclass.
- The decision about which version of a method to use is always made on the basis of what the object in fact *is* and not on what it happens to be declared as at that point in the code.
- If you need to invoke the original version of a method, you can do so by using the keyword **`super`** as a receiver. For example, if you needed to call the original version of an **`init`** method as specified by the superclass, you could call

```
super.init();
```

Summary

- Two perspectives on classes
  - Implementor
  - Client
- Javadoc produces documentation
- Classes consist of entries
- Classes can be mutable or immutable
- Entries can be **`public`**, **`private`**, **`protected`**, and package-private
- Constructors of extended classes always call a superclass constructor (explicitly or implicitly)