# 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

# 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 (memory) location **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 variable names (identifiers) 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

# Creating a Random Generator

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

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

# `RandomGenerator` Class

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

Methods of same name, but different signatures (*overloading*)

# 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

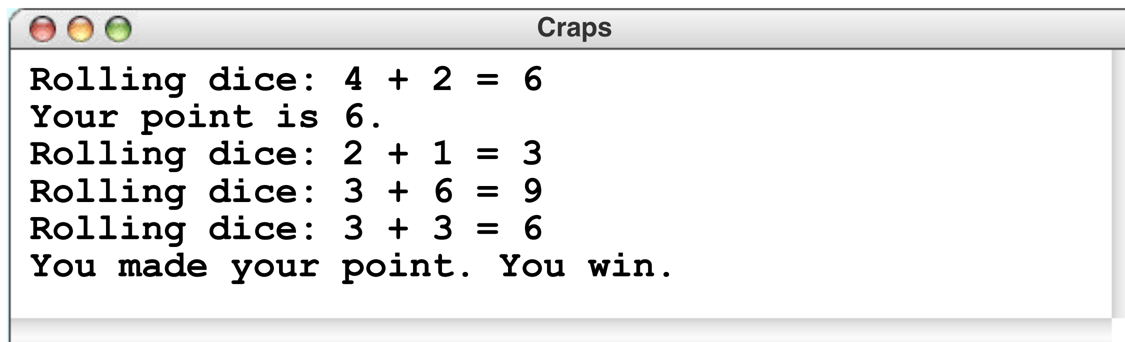
3. Change the fill color of rect to some randomly generated color.

```
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*

# 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`**

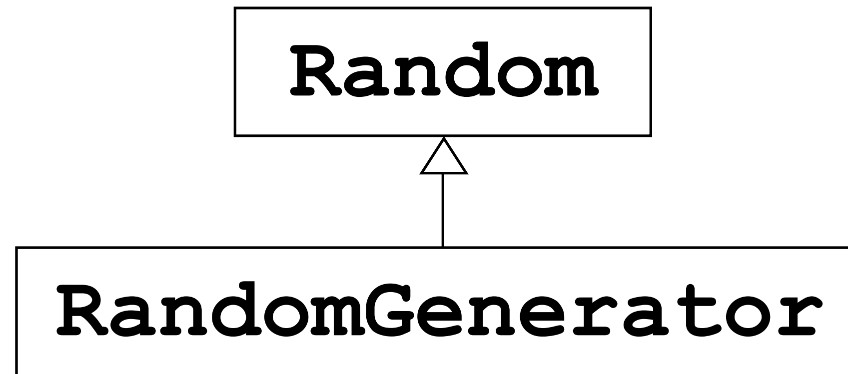
# Two Perspectives

1. Implementor
   “How does this thing work internally?”

2. Client
   “How do I use this thing?”

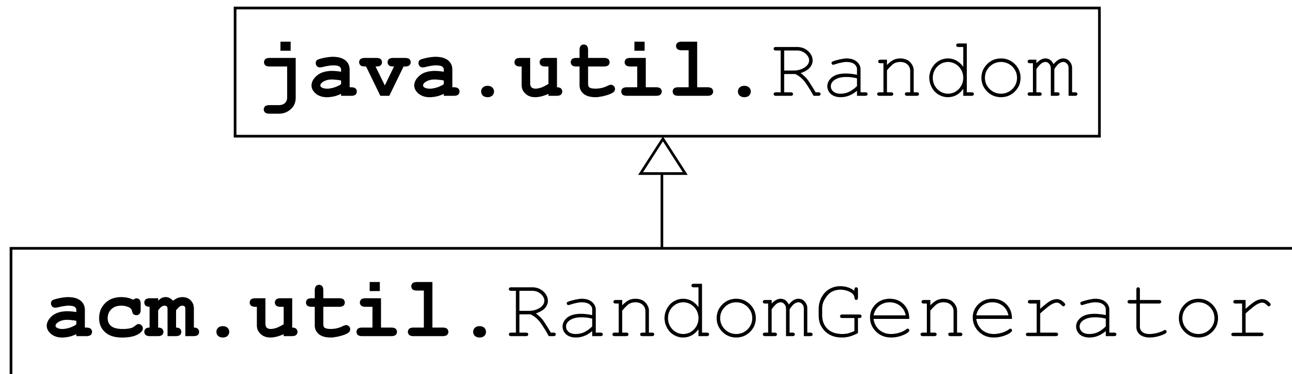
Information Hiding ➡ Similar to methods!

# Class Hierarchy




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

# Packages




**import acm.util.RandomGenerator**

Not:

**import java.util.Random**

# Simulating Randomness

- Computers are not random
  - ➟ Pseudorandom numbers
- Initialized with a *seed value*
- Explicit seed:

```
setSeed(long seed)
```

# 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) …
```

# Two Perspectives

1. Implementor
   “How does this thing work internally?”

2. Client
   “How do I use this thing?”

Information Hiding ➡ Similar to methods!

# 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

# 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;
```

# Example: `Student` Class

Encapsulate these properties:

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

# 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

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

*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**.*



skip code

# 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 + ")";
   }

/* Public constants */

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

/* 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   */

}
```

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

*Classes often export named constants.*

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

# 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*

# 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:

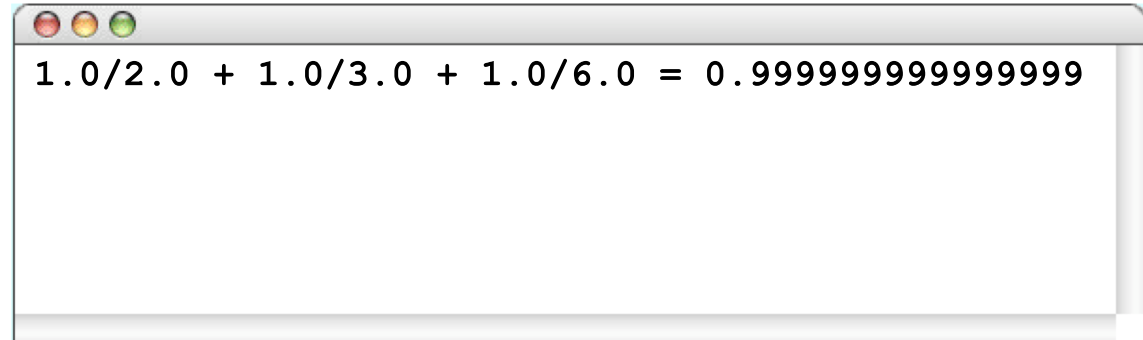
$$\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}$$

**Note:** can view this as *specification* of an ADT

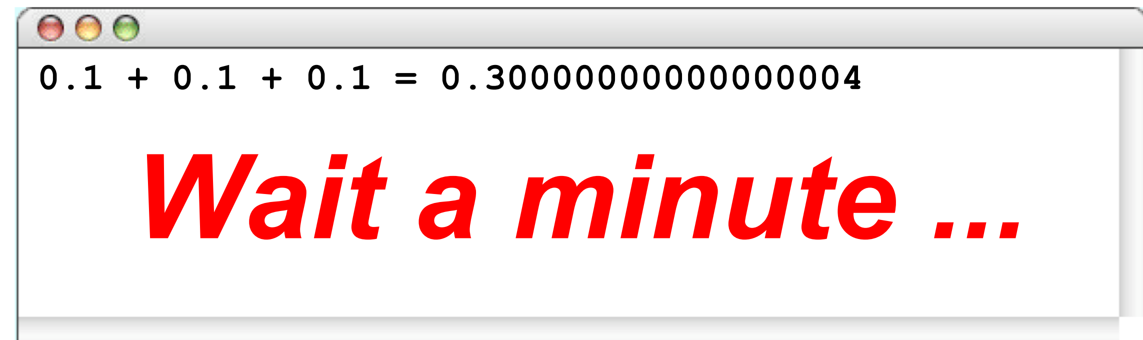
# A Rationale for `Rational`

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

Java:




Even worse:



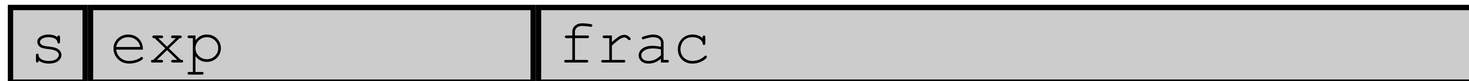
```
public class RationaleForRational extends ConsoleProgram
{
  public void run() {
    double oneHalf = 1.0 / 2.0;
    double oneThird = 1.0 / 3.0;
    double oneSixth = 1.0 / 6.0;
    double oneTenth = 1.0 / 10.0;

    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

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

$0.1999\ 9999\ 9999_{16}$

binary point

0x1999999999999

ASCII Unicode Binärwert ausblenden 8 10 16

0000 0000 0000 0000 . 0001 1001 1001 1001

63 47 32

1001 1001 1001 1001 1001 1001 1001 1001

31 15 0

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 ($0.0\overline{0011}_2$); 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!

# Adding Three Rational Values

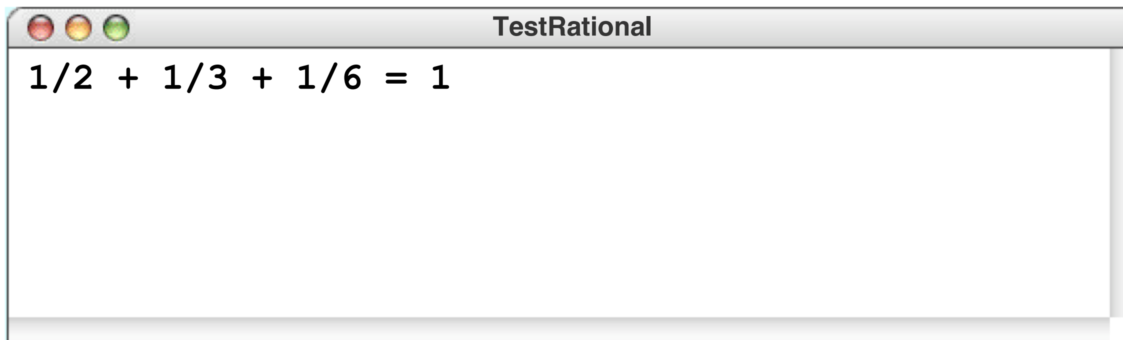
```
public void run() {
  public Rational add(Rational r) {
    public Rational(int x, int y) {
        int g = gcd(Math.abs(x), Math.abs(y));
        num = x / g;
        den = Math.abs(y) / g;
        if (y < 0) num = -num;
    }
```

36

this

| num | 1 |
|---|---|
| den | 1 |

| x | y | g |
|---|---|---|
| 36 | 36 | 36 |




*skip simulation*

# Immutable Classes

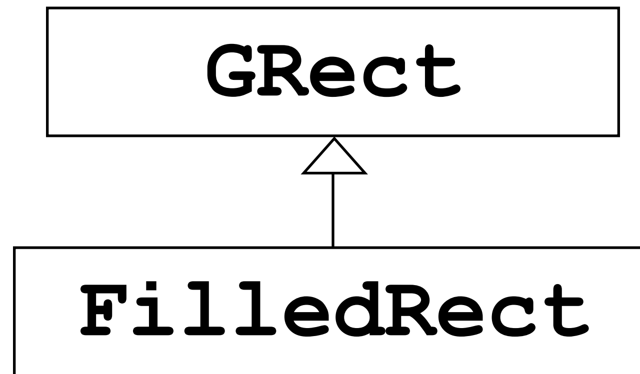
**`Rational`** is *immutable*

- 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()`**

# 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.*

# 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)