Java Notes

if Statement - Overview

Purpose

The purpose of the if statement is to make decisions, and execute different parts of your program depending on a boolean true/false value. About 99% of the flow decisions are made with if. [The other 1% of the decisions use the switch statement.]

General Forms

The if statement has this form, where condition is true or false.

```java
... // Do these statements before.
if (condition) {
    ... // Do this clause if the condition is true.
}
... // Do these statements after.
```
or

```java
... // Do these statements before.
if (condition) {
    ... // Do this clause if the condition is true
} else {
    ... // Do this clause if the condition is false
}
... // Do these statements after.
```

Example - EvaluateScore.java

This displays one of two messages, depending on an input value.

```java
// Description: Evaluate a test score. Illustrates if statement.
// File    : if/EvalScore.java
// Author: Fred Swartz - 2007-04-09 - Placed in public domain.
import javax.swing.*;

class EvalScore {
    public static void main(String[] args) {//... Input a score.
        String scoreStr = JOptionPane.showInputDialog(null, "Enter your score?");
        int score = Integer.parseInt(scoreStr);

        String comment;   // Message to the user.
        if (score >= 60) {
            comment = "Not so bad";
        } else {
            comment = "This is terrible";
        }

        //... Output the message.
        JOptionPane.showMessageDialog(null, comment);
    }
}
Flowchart representation of if statement

The flow of execution in a program is sometimes represented by a flowchart. Flowcharts are sometimes a good way to outline the flow in a method, especially if it gets complex. They may be useful in user documentation, but most flowcharting is done in quick sketches, which are thrown away once the program is written.

Decisions (if statements) are written in diamonds, and computational processes are written in boxes. There are several styles, and here is one of the most common.

```
score = ...

false
Score > 60
true

comment = "This is terrible"

comment = "Not so bad"

JOptionPane."
```

Alternate ways to write the above if statement

There are lots of ways to write the above if statement. Here are some.

1. Reverse the condition. Which to put first? There are two practices: Either put the normal case first, or the case that makes the boolean condition easiest to read.

   ```
   String comment;   // Message to the user.
   if (score < 60) {
       comment = "This is terrible";
   } else {
       comment = "Not so bad";
   }
   ```

2. Initialize the variable to a default value, and only change it if necessary. This is often used when the condition is only rarely true.

   ```
   String comment = "Not so bad;   // Message to the user.
   if (score < 60) {
       comment = "This is terrible";
   }
   ```

3. BAD: Two ifs. This is almost always a bad way to write an if-else. It's confusing to read, border values can slip through, and both conditions must be evaluated (inefficiency).

   ```
   // BAD BAD BAD BAD BAD BAD BAD BAD BAD
   String comment;   // Message to the user.
   if (score < 60) {
       comment = "This is terrible";
   }
   if (score >= 60) {
       comment = "Not so bad";
   }
   ```
**Brace style**

**Always write braces.** It is good programming style to always write the curly braces, `{}`, altho they are not needed if the clause contains only a single statement. There are two reasons this is good.

- **Reliability.** When code is modified, the indentation is such a strong indicator of structure that the programmer may not notice that the addition of a statement at the "correct" indentation level really isn't included in the scope of the if statement. This is a suprisingly common error.

- **Readability.** It is faster to read code with the braces because the reader doesn't have to keep in mind whether they are dealing with an un-braced single statement or a braced block.

**History.** Braces have been used in most language that have descended from Algol, including C, C++, Java, C# etc because language designers want to make it easy for programmers in earlier languages to make the transition. Braces are an annoying and error prone, and numerous languages, eg, Visual Basic and Python, have tried to choose better notation.

*Copyleft 2007 [Fred Swartz MIT License](http://www.leepoint.net/notes-java/flow/if/10if-general.html)*
Java Notes

'if' Statement - Indentation

Indent to make programs readable

There are several methods to make programs readable. How can you easily make the reader see which statements are inside the true part and false part of an `if` statement.

The best way to show this is to **indent** the statements that are inside. To do this you **move the statements to the right** by a few spaces. People commonly use two, three, or four spaces. Choose one number (e.g., I use 2 or 3), and use it for all programs.

Java doesn't care about your indentation -- it is for humans (including yourself!).

Example 1 - No indentation - BAD BAD BAD

Here is the `paintComponent()` method from a previous page without indentation. This is small, so it's easy to see which statements are in the true and false parts. If the `if` statement is much larger, it will be unreadable without indentation.

```java
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    if (marks < 50)
        g.setColor(Color.red);
    else
        g.setColor(Color.black);
    g.drawString("Score = " + marks, 10, 50);
}
```

Example 2 - No indentation and no line breaks

Even a very short method is almost unreadable when you take out the line breaks and spaces. Here is the same method:

```java
public void paintComponent(Graphics g) {super.paintComponent(g);if (marks<50)
    g.setColor(Color.red);else g.setColor(Color.black);g.drawString("Score = " + marks,10,50);}
```
Java Notes

'else' Not Required

'else' is not required
It is not necessary to have the else part of an if statement. Maybe only 50% of the time there is an else part.

Form
The if statement without an else has this form:
if (condition) {
    do this if the condition is true
}

Example
Here is a paintComponent() method with an if statement without an else clause.

public void paintComponent(Graphics g) {
    super.paintComponent(g); // draw background etc.
    if (marks < 50) {
        g.setColor(Color.red);
    }
    g.drawString("Score = " + marks, 10, 50);
}

When the paintComponent() method begins, the Graphics context g uses Color.black by default. Therefore there is no need to set the color to black.

Copyleft 2003 Fred Swartz MIT License, Last update: 2003-10-31
Braces are your friend

Braces ( ) not required for one statement (but are always good)

If the true or false clause of an if statement has only one statement, you do not need to use braces (also called "curly brackets"). This braceless style is dangerous, and most style guides recommend always using them.

Braceless form

The if statement doesn't need braces if there is only one statement in a part. Here both the true and false parts have only one statement:

```
// Legal, but dangerous.
if (condition)
  Exactly one statement to execute if condition is true
else
  Exactly one statement to execute if condition is false
```

Examples showing what can go wrong

A sample written with braces.

```
//... Good style - Indented with braces.
String comment = "Not so bad.";
if (marks < 60) {
  comment = "This is terrible.";
}
System.out.println(comment);
```

Without braces it's still correct, but not as safe.

```
//... Less good style - Indented but without braces.
String comment = "Not so bad.";
if (marks < 60)
  comment = "This is terrible.";
System.out.println(comment);
```

What can go wrong?

Q1: What does this "legal" version print?

```
//... What does this print?
String comment = "Not so bad.";
if (marks < 60);
  comment = "This is terrible.";
System.out.println(comment);
```

A: it always prints "This is terrible" because of that semicolon after the if clause. The semicolon indicates an empty statement, which satisfies the compiler, but is surely not what you intended. Putting a beginning brace after the if condition prevents programmers from also adding a semicolon and creating this kind of error.

Q2: What's wrong with this?

So your program is working OK without the braces and you decide to add a grade. The compiler is very happy with this, but you won't be. Why?

```
//... What does this print?
String comment = "Not so bad.";
String grade   = "A";
if (marks < 60)
  comment = "This is terrible.";
  grade   = "F";
```
A: Although the comment will be appropriate to the score, the grade will always be "F". Although the second grade assignment is indented, it isn't inside the if because the unbraced clause only includes one statement! This appearance of being included is a major source of programming errors.

Other Java constructions use braces

There are many kinds of Java statements that use braces to group things. You've already seen class and method (eg, main) declarations, which enclose their contents in braces. In addition to ifs, you'll learn about loops (for, while, and do), try...catch, and switch statements which use braces to enclose other statements.

References

- Braceless if considered harmful (cafe.elharo.com/java/braceless-if-considered-harmful/)

Copyleft 2007 Fred Swartz MIT License
Java Notes

'if' Statement - 'else if' style

Series of tests

It is common to make a series of tests on a value, where the else part contains only another if statement. If you use indentation for the else part, it isn't easy to see that these are really a series of tests which are similar. It is better to write them at the same indentation level by writing the if on the same line as the else.

Example -- series of tests - cascading ifs

This code is correctly indented, but ugly and hard to read. It also can go very far to the right if there are many tests.

```java
if (score < 35) {
    g.setColor(Color.MAGENTA);
} else {
    if (score < 50) {
        g.setColor(Color.RED);
    } else {
        if (score < 60) {
            g.setColor(Color.ORANGE);
        } else {
            if (score < 80) {
                g.setColor(Color.YELLOW);
            } else {
                g.setColor(Color.GREEN);
            }
        }
    }
}
```

Example -- using 'else if' style for formatting

Here is the same example, using a style of writing the if immediately after the else. This is a common exception to the indenting rules, because it results in more readable programs. Note that it makes use of the rule that a single statement in one of the Java clauses doesn't need braces.

```java
if (score < 35) {
    g.setColor(Color.MAGENTA);
} else if (score < 50) {
    g.setColor(Color.RED);
} else if (score < 60) {
    g.setColor(Color.ORANGE);
} else if (score < 80) {
    g.setColor(Color.YELLOW);
} else {
    g.setColor(Color.GREEN);
}
```

Other languages

Some programming languages recognize this common construction with a special elseif keyword. Although it is hardly necessary, this kind of small touch can make a language a little nicer to use. The Java language designers are very conservative about adding keywords to the language, so don't expect it.

Copyright 2006 Fred Swartz MIT License
Java Notes

'if' Statement - if inside if

if inside if

You can put an if statement inside another if statement.

Nearest 'else'

If you use braces, there is no problem with deciding which else goes with which if For example,

```java
if (age < 24) {
    if (height > 200) {
        c = Color.RED;
    }
} else {
    c = Color.BLUE;
}
```

Because the true and false parts are both single statements, you might be tempted to omit braces and write:

```java
if (age < 24)
    if (height > 200)
        c = Color.RED;
else                // DANGER: which 'if' goes with this 'else'
    c = Color.BLUE;
```

But this is WRONG, because 'else' always goes with the nearest 'if' when there are no braces. This code is the same as:

```java
if (age < 24) {
    if (height > 200)
        c = Color.RED;
    else
        c = Color.BLUE;
}
```

Advice: Always use braces on if statements

These kinds of errors are very hard to find. This is another good reason to always use braces.

Watch out for semicolons on your if statements

Why does the following code always say it thinks the user is lying?

```java
String ageStr = JOptionPane.showInputDialog(null, "How old are you?");
int age = Integer.parseInt(ageStr);
if (age > 120 || age < 0);
    System.out.println("I think you're lying about your age!");
```

It's the semicolon! If you put a semicolon directly after the condition in an if statement, Java thinks it's finished with the body of the statement. The indentation of the next line, which is so important to human readers, is ignored by Java.

This is another error that's harder to make if you always follow the condition by an opening brace.

Copyleft 2006 Fred Swartz MIT License
Arrays

Arrays are to data as loops are to control flow.

Overview

The basic solution. Arrays are the basic way to store large numbers of values. Every programming language you're likely to use has arrays as part of the fundamental language, altho with many little variations on how they work.

Strings are like arrays of chars and you should be familiar with the idea of storing multiple values which are accessed by a numeric index.

Other possibilities. Arrays in Java are great for working with a fixed numbers of elements, but Java also has the Collections library of classes that are a better choice when working with a variable number of objects, eg, ArrayList, but it's essential to learn about arrays, so we'll start with them.

Arrays store many values using one name and an index

An array can store many similar values in memory. Each value is accessed by specifying an integer subscript or index in brackets following the array name. "Array" in Java means approximately the same thing as array, matrix, or vector does in math. Unlike math and some other programming languages, in Java you must both declare an array and allocate a fixed amount of memory for it.

Declaring an array

An array variable is like other variables -- you must declare it, which means you must declare the type of elements that are in an array. All elements must be the same type. Write the element type name, then "["], then the name of the array variable. The declaration allocates only enough space for a reference to an array (typically 4 bytes), but doesn't create the actual array object.

```
String[] args;    // args is an array of Strings
int[] scores;    // scores is an array of ints
JButton[] controlButtons; // controlButtons is an array of Buttons
```

No size in declaration. Unlike some languages, never put the size of the array in the declaration because an array declaration specifies only the element type and the variable name. The size is specified when you allocate space for the array.

Names - Plurals or collective nouns are most common for array names

Most programming guidelines suggest using plural names, or nouns denoting a collection of things, for arrays and other collections of multiple values. This is not a rigid rule, and your choice will often be based on linguistic sensitives. For example, you might have an array of words and their frequencies. It would be entirely appropriate to name this wordFrequencyTable because the word "table" suggests many entries. If you had an array of single words, you might call it word or wordList. Naming the array using the singular would probably be very confusing to most readers.

Examples in this text often follow the common convention of using the array variable name "a". This seriously violates the rule of having meaningful names, so please don't adopt this textbook-example style in your code!

Allocate an array object with new

Create an array using new. This example creates an array of 100 int elements, from a[0] to a[99].

```
int[] a;    // Declare a to be an array of ints
a = new int[100]; // Allocate an array of 100 ints
```

These are often combined in one line.

```
int[] a = new int[100]; // Declare and allocate.
```
**Subscripts (indexes/indices)**

Subscripts are enclosed in square brackets \([\cdot]\). \(x_1\) in mathematics is \(x[1]\) in Java, and is pronounced "x-sub-i".

Subscript ranges always start at zero because Java came largely from C++, which had a good reason for using zero (pointer arithmetic on arrays). It isn't the way that humans normally count; you'll just have to live with it.

Java always checks subscript legality to be sure the subscript is \(\geq 0\), and less than the number of elements in the array. If the subscript is outside this range, Java throws ArrayIndexOutOfBoundsException. This is far superior to the behavior of C and C++, which allow out of range references. Consequently, Java programs are far less susceptible to bugs and security flaws than C/C++ programs.

Zero-based indexing is a constant annoyance of Java’s zero-based indexing. The natural human value for hours or days should be used as an index to make the program most readable, but this would mean starting with 1, not zero.

**Length of an array**

Each array has a constant (final) instance variable that has its length. You can find out how many elements an array can hold by writing the array name followed by `.length`. In the previous example, `a.length` would be 100. Remember that this is the number of elements in the array, one more than the maximum subscript.

**Example - Reading into an array**

This shows typical input code for reading into an array from a Scanner. This program is very strange because it doesn’t do anything with the data, but it’s shown here as an early step in the iterative process.

```java
// InputArray.java -- Simple demo of reading into an array.
// Fred Swartz - 26 Aug 2006 - Placed in the public domain.

import java.util.*;

public class InputArray {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int[] scores = new int[100];
        int n = 0;  // Current number of values in scores.

        while (in.hasNextInt()) {
            if (n >= scores.length) {
                System.err.println("Too much data!");
                System.exit(1);
            }
            scores[n] = in.nextInt();
            n++;
        }
        //... Now do something with the scores!
    }
}
```

**Java idiom for looping over an array - for loop**

For loop. The most common use of `.length` is in a for loop test condition. For example, the variable `i` will go over the entire range of subscripts of the array `a`.

```java
int[] a = new int[1000]; // Declare, allocate array of 1000 ints, a[0]...a[999]
```
//... Assign random values to each element.
for (int i = 0; i < a.length(); i++) {
   a[i] = (int)(Math.random() * 100000); // Random number 0-99999
}

//... Add all elements of the array a.
int sum = 0;
for (int i = 0; i < a.length; i++) {
   sum += a[i];
}

Enhanced for loop. If you only need to reference the value of each of the elements, you can use the somewhat simpler enhanced for loop (also known as the foreach loop), which keeps track of the index and assigns successive values to a variable (v in this example). The foreach loop only gets the values, so it couldn't have been used to set the values in the first loop above.

//... Add all elements of the array a.
int sum = 0;
for (int v : a) {
   sum += v;
}

Initial array element values -- zero/null/false

When an array is allocated (with new), all elements are set to an initial value. The initial value is 0 if the element type is numeric (int, float, ...), false for boolean, and null for all object types.

Array Initialization

When you declare an array, you can can also allocate a preinitialized array object in the same statement. In this case, do not give the array size because Java counts the number of values to determine the size. For example,

```java
String[] days = {"Su", "Mo", "Tu", "We", "Th", "Fr", "Sa"};
```

Array diagrams

This code declares and initializes an array.

```java
int[] a = {1, 2, 4, 8, 16};
```

Arrays are often represented with diagrams that represent their memory use. The diagrams below are typical ways to represent the memory used by an array.

Each box represents the amount of memory needed to hold one array element. For ints this is 4 bytes. The array variable, a, is a reference (ie, memory address in the heap) of the block of memory that holds the actual array elements. References are often represented by a black disk with an arrow pointing to the data they reference. Most often array diagrams are written vertically, but sometimes the cells are arranged horizontally, especially for arrays of characters. Of course in the actual physical memory there is no such idea as vertical or horizontal.

```
1
2
4
8
16
```
Array variables are references to a block of elements

When you declare an array variable, Java reserves only enough memory for a reference (Java's name for an address or pointer) to an array object. References typically require only 4 bytes. When an array object is created with `new`, a reference is returned, and that reference can then be assigned to a variable. When you assign one array variable to another, only the reference is copied. For example,

```java
int[] a = new int[] {100, 99, 98}; // "a" references the array object.
int[] b; // "b" doesn't reference anything.

b = a;      // Now "b" refers to the SAME array as "a"
b[1] = 0;   // Also changes a[1] because a and b refer to the same array.
```

Example (AutoSales) - Translating between subscript and data ranges

Indexes in other ranges. If the index data is in a range far from zero, eg auto sales by year, translate the index by subtracting the minimum value. For example,

```java
import java.util.*;

public class AutoInfo {
   static final int FIRST_YEAR = 1900;
   static final int LAST_YEAR  = 2020;

   public static void main(String[] args) {
      int[] autoSales = new int[LAST_YEAR - FIRST_YEAR + 1];
      Scanner in = new Scanner(System.in);

      //... Read data as pairs of year and sales values.
      while (in.hasNextInt()) {
         int year  = in.nextInt();
         int sales = in.nextInt();
         autoSales[year - FIRST_YEAR] = sales;
      }

      //... Here's where code to process this data belongs.

      for (int year = FIRST_YEAR; year <= LAST_YEAR; year++) {
         int sales = autoSales[year - FIRST_YEAR];
         if (sales > 0) {
            System.out.println(year + " " + sales);
         }
      }
   }
}
```

Alternate loop to display values. Another way to write the above output is to loop over index values, then translate these to the appropriate data range.

```java
for (int i = 0; i < autoSales.length; i++) {
   int sales = autoSales[i];
   int year  = FIRST_YEAR + i;
   ...
}
```
if (sales > 0) {
    System.out.println(year + " " + sales);
}

Common array problems

Some common array programming mistakes are:

- Runtime error: Forgetting that array subscripts start with zero.
- Compile-time error: Writing `a.length()` instead of `a.length`. The `length()` method is used with Strings, not arrays.
- Compile-time error: Declaring an array with a size. Eg, `int[100] a;` instead of `int[] a = new int[100];`.

Copyleft 2007 Fred Swartz MIT License
Java Notes

Loops - Introduction

The purpose of loop statements is to repeat Java statements many times. There are several kinds of loop statements in Java.

**while statement - Test at beginning**

The `while` statement is used to repeat a block of statements while some condition is true. The condition must become false somewhere in the loop, otherwise it will never terminate.

```java
//... While loop to build table of squares.
String result = "";  // StringBuilder would be more efficient.
int i = 1;
while (i <= 20) {
    result = result + i + " squared is " + (i * i) + "\n";
    i++;
}  
JOptionPane.showMessageDialog(null, "Tables of squares\n" + result);
```

The following example has an assignment inside the condition. Note that "=" is assignment, not comparison ("=="). This is a common coding idiom when reading input.

```java
//... Add a series of numbers.
JOptionPane.showMessageDialog(null, "Enter ints. Cancel to end");
String valStr;
int sum = 0;
while ((valStr = JOptionPane.showInputDialog(null, "Number?")) != null) {
    sum += Integer.parseInt(valStr.trim());
}
JOptionPane.showMessageDialog(null, "Sum is " + sum);
```

**for statement - Combines three parts**

Many loops consist of three operations surrounding the body: (1) initialization of a variable, (2) testing a condition, and (3) updating a value before the next iteration. The `for` loop groups these three common parts together into one statement, making it more readable and less error-prone than the equivalent `while` loop. For repeating code a known number of times, the `for` loop is the right choice.

```java
//... For loop to build table of squares.
String result = "";  // StringBuilder would be more efficient.
for (int i = 1; i <= 20; i++) {
    result += i + " squared is " + (i * i) + "\n";
}  
JOptionPane.showMessageDialog(null, "Tables of squares\n" + result);
```

**do..while statement - Test at end**

When you want to test at the end to see whether something should be repeated, the `do..while` statement is the natural choice.

```java
String ans;
do {
    . . .
    ans = JOptionPane.showInputDialog(null, "Do it again (Y/N)?");
}while (ans.equalsIgnoreCase("Y"));
```

**"foreach" statement - Java 5 data structure iterator**

...
Java 5 introduced what is sometimes called a "for each" statement that accesses each successive element of an array, List, or Set without the bookkeeping associated with iterators or indexing.

```java
//... Variable declarations.
JTextArea nameTextArea = new JTextArea(10, 20);
String[] names = {"Michael Maus", "Mini Maus"};

//... Display array of names in a JTextArea.
for (String s : names) {
    nameTextArea.append(s);
    nameTextArea.append("\n");
}
```

**Similar to the 'if' statement**

There are three general ideas that you will see in many parts of Java.

- **Braces** `{}` to enclose multiple statements in the body.
- **Indentation** to show the extent of the body clearly.
- **Boolean** (true/false) conditions to control whether the body is executed.

**Scope of loop indicated with braces {}**

If the body of a loop has more than one statement, you must put the statements inside braces. If there is only one statement, it is not necessary to use braces `{}`. However, many programmers think it is a **good idea to always use braces** to indicate the scope of statements. Always using braces allows the reader to relax and not worry about the special single statement case.

**Indentation.** All statements inside a loop should be indented one level (eg, 4 spaces), the same as an `if` statement.

_Copyright 2005 [Fred Swartz](http://www.fredswartz.com) MIT License_
Java Notes

'for' Loop

Purpose

The for statement is similar to the while statement, but it is often easier to use if you are counting or indexing because it combines three elements of many loops: initialization, testing, and incrementing.

General Form

The for and equivalent while statements have these forms.

```java
for (init-stmt; condition; next-stmt) {
    body
}
```

```java
init-stmt;
while (condition) {
    body
    next-stmt;
}
```

There are three clauses in the for statement.

1. The init-stmt statement is done before the loop is started, usually to initialize an iteration variable.
2. The condition expression is tested before each time the loop is done. The loop isn't executed if the boolean expression is false (the same as the while loop).
3. The next-stmt statement is done after the body is executed. It typically increments an iteration variable.

Example - Printing a table of squares

Here is a loop written as both a while loop and a for loop. First using while:

```java
int number = 1;
while (number <= 12) {
    System.out.println(number + " squared is " + (number * number));
    number++;
}
```

And here is the same loop using for.

```java
for (int number = 1; number <= 12; number++) {
    System.out.println(number + " squared is " + (number * number));
}
```

Example - Counting doubled characters

This code will look at each character in a string, sentence, and count the number of times any character occurs doubled.

```java
String sentence = "...;
int doubleCount = 0;       // Number of doubled characters.

// Start at second char (index 1).
for (int pos = 1; pos < sentence.length(); pos++) {
    // Compare each character to the previous character.
    if (sentence.charAt(pos) == sentence.charAt(pos-1)) {
        doubleCount++;
    }
}
```

Summary
The `for` loop is shorter, and combining the initialization, test, and increment in one statement makes it easier to read and verify that it's doing what you expect. The `for` loop is better when you are counting something. If you are doing something an indefinite number of times, `while` loop may be the better choice.

Copyleft 2005 Fred Swartz MIT License
Java Summary: Math.random() and java.util.Random

Math.random() method
A standard way to generate random numbers is to use the Math.random() method, which returns a double value in the range 0.0 up to, but not including 1.0. You can use multiplication to expand the range, addition to shift the range, and casting to convert it to an integer. The advantage of this method over the Random class (see below) is that there is no need to create an object. The disadvantage is that you often have to do more work to translate the return value into a range you want.

double Math.random()

Returns a number x in the range, 0.0 <= x < 1.0.

java.util.Random Class
The java.util.Random class provides more flexible ways to generate uniformly distributed random numbers, providing easy generation of types other than double, as well as providing a Gaussian distribution.

Random Constructors
- Random r = new Random(); Uses time in milliseconds as the seed.
- Random r = new Random(long seed); Uses the provided seed for testing purposes.

Random Methods
All methods return a uniform distribution of values, except nextGaussian(). Assume r is a Random object.

int i = r.nextInt(int n) Returns random int >= 0 and < n.
int i = r.nextInt() Returns random int (full range).
long l = r.nextLong() Returns random long (full range).
float f = r.nextFloat() Returns random float >=0.0 and < 1.0.
double d = r.nextDouble() Returns random double >=0.0 and < 1.0.
boolean b = r.nextBoolean() Returns random boolean (true or false).
double d = r.nextGaussian() Returns random double with mean 0.0 and standard deviation 1.0.

Copyright 2005 Fred Swartz