Chapter 1

Review of Java Fundamentals

Lecture 3

Jenny Walter

Fall 2008

Using acm.jar files

```
package lect3;

/*
 * File: TestInput.java
 * ------------------------
 * This program creates a new console window for
 * user input.
 */

public class TestInput {

    public static void main(String[] args) {
        new InputConsole();
    }

}
```

All this main method does is create a new InputConsole object--no need to import acm package here.

```
package lect3;

import acm.program.*;

public class InputConsole extends ConsoleProgram{
  public InputConsole() {
    this.start();
    this.println("This simple program reads\n"+
            "an integer, a double, and a String\n"+
            "from the user and displays them on\n"+
            "the screen.");
    int first = this.readInt("\nEnter an integer: ");
    double second = this.readDouble
                                ("\nEnter a double: ");
    String third = this.readLine("\nEnter anything: ");
    this.println("\n\nInteger entered: "+first+
                 " Double entered: "+second+
                 " String entered: "+third);
 }
}
```

InputConsole extends ConsoleProgram, thereby gaining access to all methods in ConsoleProgram

Elegant Code & Indentation

- Rules of indentation:
  - Open curly braces can either be on the same line as the definition that requires the braces or on the following line
  - Within each matched pair of curly braces, the statements should be indented 2-4 spaces and all statements at a given indentation level should be vertically aligned
  - Closing curly braces should be on the line *after* the last statement within the pair of braces

Elegant Code & Documentation

- Rules of documentation:
  - Choose identifiers that reflect the purpose of the entity being named; results in "semi-self-documenting" code (usually more explanation is required)
  - In the header comment (at the top of every file), include:
    - CMPU125, Fall 2008
    - Lab or HW number and date of completion
    - ***Your name***
    - The file name and a brief synopsis of its purpose

Elegant Code & Documentation

- Rules of documentation (cont.):
  - Include header comments before each new method
    - Describe any applicable pre-conditions
    - Describe purpose of parameters and any assumptions you are making about the arguments to those parameters
    - Describe return type and post-conditions
    - NetBeans starts some new comments for you

Readable Code

- Make sure the length of each line is at most 80 characters so the program listing contains no line-wraps
  - Code with line-wraps can be torture to read
  - I hate to be tortured
  - So keep your lines short

  NetBeans has a vertical red line in the editor to show the line length limit in column 80.
- Put a couple of blank lines in your code now and then to separate logical units

The javadoc Documentation System

Java was designed to operate in the web-based environment.

One of the ways 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.

A tutorial on using javadoc can be found at the following link:

http://java.sun.com/j2se/javadoc/writingdoccomments/

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 **javadoc** comments also often contain **@param** and **@result** tags to describe parameters and results, as illustrated on the next slide.

An Example of javadoc Comments

The **javadoc** comment

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

produces the following entry in the "Method Detail" section of the web page.

```
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
```

The javadoc Documentation System

In order to produce .html files like the ones in the Java API, you need to run the javadoc utility, specifying your .java file as input.

Another javadoc tutorial can be found at the link below:

http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/javadoc.html

DrJava

- Free IDE you can download from the web.
  - Advantage over NetBeans is a nicer editor that does automatic text indentation.
  - Disadvantage is that DrJava doesn't do auto completion for things like imported and inherited methods.
  - NetBeans seems to lack a feature for automatically indenting code. One suggestion I have is to do auto-indenting in DrJava
  - Inside DrJava, you can select the whole file (using Ctrl-A or Command-A) and then press tab to indent all lines to the right level. Be sure to save the file after indentation is complete.

Editing, Compiling and such

- In lab or for homeworks, you can work in any IDE you want. Use the emacs editor and compile and run your program in emacs or at the command line if you so choose.
- I would prefer if you could submit just the .java files from the src subdirectory of each project instead of the entire NetBeans project.
- I'll include directions in the next lab, detailing how you can find these files and submit them.

Escape Sequences

- Escape sequences are literals that can be embedded in Strings

`\n`	new line
`\t`	tab
`\"`	double quote
`\\`	backslash
`\'`	single quote

String continuations

- The Java compiler won't allow a String to span more than one line
- This comes up frequently when a call to println doesn't finish a String literal in a single line
- Solution: End the String using a " and add a + (concatenation operator) to continue the String on the next line.

String class

- Class `String`
  - Declaration examples:
    - `String title;`
    - `String title = “Walls and Mirrors”;`
  - Assignment example:
    - `Title = “Walls and Mirrors”;`
  - String length example:
    - `title.length();`
  - Referencing a single character
    - `title.charAt(0);`
  - Comparing strings
    - `title.compareTo(string2);`

String class

- Class `String`
  - Other useful methods
    - `substring(int n)`
    - `substring(int i,int n)`
    - `toUpperCase()`
    - `toLowerCase()`
    - `indexOf(int c)`
    - `startsWith(String prefix)`
    - `endsWith(String suffix)`
    - `replace(char newChar, char oldChar)`

String class

- Concatenation example:

```
String monthName = "December";
int day = 31;
int year = 02;
String date = monthName + " " + day +
              ", 20" + year;
```

The + operator works like addition when both operands are numbers and it works like concatenation when one operand is a String

Selection Statements

- The if statement

```
if (expression)
  statement1
```

or

```
if (expression)
  statement1
else
  statement2
```

- Nested if

```
if (expression) {
  statement1
}
else if (expression)
  {
  statement2
}
else {
  statement3
} // end if
```

Selection Statements

- The switch statement

```
switch (integral expression) {
  case 1:
   statement1;
   break;
  case 2, case 3:
   statement2;
  case 4:
   statement3;
   break;
  default:
   statement4;
} //end of switch
```

Iteration Statements

- The while statement

```
while (expression) {
  statement
}
```

- statement is executed as long as expression is true
- statement may not be executed at all

Iteration Statements

- The loop-and-a-half while statement

```
while (true) {
  statement
  if (expression)
     break;
}
```

- break statement
  - Exits the innermost loop if expression is true

Using Continue

- The while statement

```
while (expression1) {
  statementSet1
  if (expression2)
     continue;
  statementSet2
}
```

- continue expression
  - Stops the current iteration of the loop and begins the next iteration at the top of the loop
  - Skips statementSet2 if expression2 is true

Iteration Statements

- The for statement

```
for (initialize; test; update)
  statement
```

- statement is executed as long as test is true
- for statement is equivalent to a while statement

Iteration Statements

- The for statement as loop-and-a-half

```
for (;;) {
  statement
  if (expression)
     break;
}
```

- statement is executed until expression becomes true

Iteration Statements

- The do statement

```
do {
  statement
} while (expression);
```

- statement is executed until expression is false
- do statement loops at least once

static vs non-static class members

- Data fields and method definitions can include the keyword static
- Static members are accessible either through an object of the class type or through the class name
  - I prefer you access static members through the class name (e.g., Math.abs(…), Color.RED, Integer.parseInt(…)) for the sake of readability
- A non-static member cannot be accessed from a static method of the same class unless you first create an object of the class type

Useful Java Classes

- Class StringBuffer
  - Creates mutable strings
  - Provides same functionality as class String
  - More useful methods
    - **public** StringBuffer append(String str)
    - **public** StringBuffer insert(int offset, String str)
    - **public** StringBuffer delete(int start, int end)
    - **public** void setCharAt(int index, char ch)
    - **public** StringBuffer replace(int start, int end, String str)

Useful Java Classes

- Class StringTokenizer
  - Allows a program to break a string into pieces or tokens
  - More useful methods
    - **public** StringTokenizer(String str)
    - **public** StringTokenizer(String str, String delim)
    - **public** StringTokenizer(String str, String delim, boolean returnTokens)
    - **public** String nextToken()
    - **public boolean** hasMoreTokens()

Object class

- The Object class
  - Java supports a single class inheritance hierarchy
    - With class Object as the root
  - Useful methods inherited by all user-defined classes (but need to be overridden to be useful):
    - **public boolean** equals(Object obj)
    - **protected void** finalize()
    - **public int** hashCode()
    - **public String** toString()

The Java Hierarchy

- Each class can extend at most one other class.
- Every class implicitly extends the Object class
  - Result is that every class inherits all the methods defined in the Object class.
  - However, to be useful, these methods must be *overridden*, meaning that an implementation must be provided for that method in each subclass

Overloading vs. Overriding

- *Method overloading* refers to multiple definitions of methods with the same name within the same class. Overloaded methods must have different types or numbers of parameters.

- *Method overriding* refers to subclasses defining methods with the same method definition line (aka signature) as one of their superclasses. To override a superclass method, the method definition line of the subclass must match that of the superclass exactly.

  It is a good idea to override the toString method because it is called automatically when an object is included in a print statement.