Java Chapter 4 Part 2

* Nesting, Loop Control, File I/O, and Random Numbers
* CIS 255 • Shelby-Hoover Campus

Nested Loops

* When one loop structure is nested inside another, the inner loop completes all of its iterations every time the outer loop goes through one iteration
* Example: if the outer loop has three iterations and the inner loop has five iterations, the code in the inner loop body will execute a total of fifteen times (3 \* 5)
* Example Code:  
    
  public class ProductLoops  
  {  
   public static void main(String[] args)  
   {  
   int product;  
    
   for (int a = 1; a <= 3; a++)  
   {  
   for (int b = 1; b < 6; b++)  
   {  
   product = a \* b;  
   System.out.print(product + " ");  
   }  
   System.out.println(); // start a new line  
   }  
   }  
  }
* Another Example: Clock.java (Code Listing 4-12)
* Another Example: TestAverage2.java (Code Listing 4-13)

Modifying a Loop’s Execution

* Two statements will allow a program to stop a loop in the middle of an iteration
  + break; // stops the loop altogether, proceeds to the next statement after the loop
  + continue; /\* stops the current iteration, returns to either the Boolean expression (while, do-while) or the update statement (for) to see if another iteration is needed \*/
* These statements should be used sparingly; they complicate the flow of the program
  + Having break and continue statements in a loop structure allows for multiple exit points from the loop
  + To maintain program structure, each loop should only have one exit point (evaluating the Boolean expression after completing an iteration of the loop body)

Choosing a Loop Structure

* Most forms of repetition can be accomplished by any of the three loop structures
* Some problems may lend themselves more to one loop structure than to the others
  + while: sentinel-controlled loops, option to skip the loop body altogether
  + do-while: ask-before-iterating loops, forcing at least one iteration
  + for: count-controlled loops

File Input and Output

* Reading data from a file and writing data to a file works similarly to console input and output
* There are three steps involved in file processing:
  + Open the file by declaring a variable of the appropriate class type
  + Perform the reads / writes using methods
  + Close the file to prevent further use of the file in this program
* The syntax in this chapter deals with text files (Unicode); a later chapter in the textbook shows how to process binary files (not human-readable)
* The classes for file output, and most of the classes for file input, are in the package java.io; a single import statement will give your program access to all of the necessary classes:  
    
   import java.io.\*; // a wildcard import statement

Text File Output: PrintWriter

* A PrintWriter object works similarly to the System.out object
* To open a file for output:  
  PrintWriter myOutputFile = new PrintWriter("MyFileName.txt");  
  + The object name may be anything you want (as long as it’s a legal identifier)
  + You may declare as many objects as there are output files in your program (you cannot use a single object for multiple files)
  + If the file should be created in a location other than where your source code file is, include the path to that location in the quotation marks
  + This creates the file if it doesn’t already exist, but it overwrites (destroys) the contents of an existing file with that name (an alternative is described later)

Using a PrintWriter Object

* The methods to write to a PrintWriter object are the same as those for System.out:  
   myOutputFile.println("Hello, file!");  
   myOutputFile.print("Hey, file!");
* Output sent to the file will not be shown on the screen unless you place the same output in corresponding System.out statements or message dialogs
* The file must be closed to make sure that the program writes to the file any data remaining in the buffer:  
   myOutputFile.close();

PrintWriter Examples

* Writing some students’ names with grades to a file:  
  int robsGrade = 103;  
  int priscillasGrade = 77;  
  int jeffsGrade = 91;  
    
  PrintWriter gradeFile = new PrintWriter("Grades.txt");  
    
  gradeFile.println("Rob's grade: " + robsGrade);  
  gradeFile.println("Priscilla's grade: " + priscillasGrade);  
  gradeFile.println("Jeff's grade: " + jeffsGrade);  
    
  gradeFile.close();
* Writing user input to a file: FileWriteDemo.java (Code Listing 4-17)

Dealing with Output Problems

* If a Java program is not allowed to write to the file mentioned for the PrintWriter object, a checked exception will be thrown
  + There must be something in your code to indicate what the program will do with that exception
  + The easiest way to deal with the exception is to add a throws clause to the end of the header of the file-handling method:  
    public static void main(String[] args) throws IOException
  + This instructs Java to perform the normal “Java spit-up” in the console
* If you’d like to add to the existing contents of a file, there’s another class, FileWriter, that can be used with PrintWriter to specify appending rather than overwriting (the output statements are still performed with the PrintWriter object):  
  FileWriter myAppend = new FileWriter("MyFileName.txt", true);  
  PrintWriter myOutputFile = new PrintWriter(myAppend);

Text File Input: File, Scanner

* The Scanner objects examined thus far have been associated with the console (System.in), but Scanner can also be used with input files
* Include the import statements for Scanner and the java.io.\* package
* The method in which the input file is used must have the throws IOException clause at the end of its header
* To use a Scanner object with an input file, a second object of type File is required:  
   File myFile = new File("MyFileName.txt");  
   Scanner myInputFile = new Scanner(myFile);
* Declare as many Scanner objects as needed for files and the console
* Once declared, the object can be used to invoke any of the various methods to obtain input of a specific type (nextInt(), nextDouble(), etc.)
* When finished, invoke the close() method:  
   myInputFile.close();

File / Scanner Examples

* Reading exactly three values from a file:  
    
  double grade1, grade2, grade3;  
    
  File gradeFile = new File("GradeData.txt");  
  Scanner inputFile = new Scanner(gradeFile);  
    
  grade1 = inputFile.nextDouble();  
  grade2 = inputFile.nextDouble();  
  grade3 = inputFile.nextDouble();  
    
  inputFile.close();  
  + The values obtained from the file can be used in the same way that any values of type double are used
  + If a value of a type other than what is required by the Scanner method is encountered, an exception will be thrown
* Reading the first line of text from a file: ReadFirstLine.java (Code Listing 4-18)

Reading an Entire File

* If the number of items to be read from the file is not known ahead of time, the Scanner method hasNext() can be used to continue until no unread items remain in the file
* The call to the method is used as the Boolean expression controlling the loop (the method returns true or false)
* Example: FileReadDemo.java (Code Listing 4-19)

Checking for File Existence

* To prevent an exception caused by the input file not being found where expected, use the method exists() on the File object before declaring the Scanner object
* If the exists() method returns false (use the ! operator), display an error message and exit:  
    
  File myFile = new File("SomeFile.txt");  
  if (!myFile.exists())  
  {  
   System.out.println("File not found.");  
   System.exit(0);  
  }  
  Scanner myFileScanner = new Scanner(myFile);
* Example: FileSum2.java (Code Listing 4-21)
* This can also be used to prevent overwriting an existing output file (if exists() returns true, don’t try to use that file as the argument to a PrintWriter object)

Random Numbers

* Many programs (games, scientific programs, statistical programs, programs involving encryption) require the use of random numbers
* Java has a class, Random (from java.util), that provides several methods for obtaining random numbers of different types (similar to the Scanner class methods)
* Declare a Random object first:  
    
  Random myRandom = new Random();
* To obtain a random number in a specific range, you may need to specify an argument and other arithmetic:  
    
  int value = myRandom.nextInt(99) + 1;  
  + The call to nextInt(99) returns a random number between 0 and 99
  + Adding 1 to this value shifts the range up to be between 1 and 100
* Example: MathTutor.java (Code Listing 4-23)

Random Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| nextDouble() | Returns a double containing a random number between 0.0 and 1.0 |
| nextFloat() | Returns a float containing a random number between 0.0 and 1.0 |
| nextInt() | Returns an int containing a random number in the range of int  (over 2 billion, positive or negative) |
| nextInt(limit) | Returns an int containing a random number between 0 and limit  (an int) |
| nextLong() | Returns a long containing a random number in the range of long (over 9 quintillion, positive or negative) |

Table 4-1 shows some of the methods from the class Random (the Java APIs at <http://java.sun.com/> will list other methods to obtain random numbers)

Reminders for Loops

* The use of prefix or postfix mode with increment and decrement operators only impacts embedded operations
* In Java, Boolean expressions are usually written in parentheses
* Only the do-while loop has a semicolon after the parentheses containing the Boolean expression
* Multi-statement loop bodies must be enclosed in curly brackets
* Separate the three sections of the for loop header with semicolons, not commas

More Reminders for Loops

* The body of a while or do-while loop should contain a statement that updates the loop control variable
* If you place an update statement in the header of a for loop, don’t repeat that statement in the loop body
* Accumulator (sum) variables must start at 0
* The sentinel value used to end a loop should not be processed as valid data
* Sentinel-controlled loops should have one input statement before the header and one input statement before the body’s end

Reminders for File I/O and Random Numbers

* File input and file output require an extra import statement and a throws clause
* You cannot use a single PrintWriter or Scanner object for multiple files at the same time
* By default PrintWriter overwrites existing files; use FileWriter as an intermediate object to append to the file instead
* Make sure an input file exists before you try reading from it
* Use the Scanner methods that match the types of values expected from the input file
* To read the entire contents of a file, place the input statement in a loop controlled by the method hasNext()
* Different Random class methods return random numbers of different types in different ranges (in the range of the data type, between 0 and 1, or between 0 and the argument)