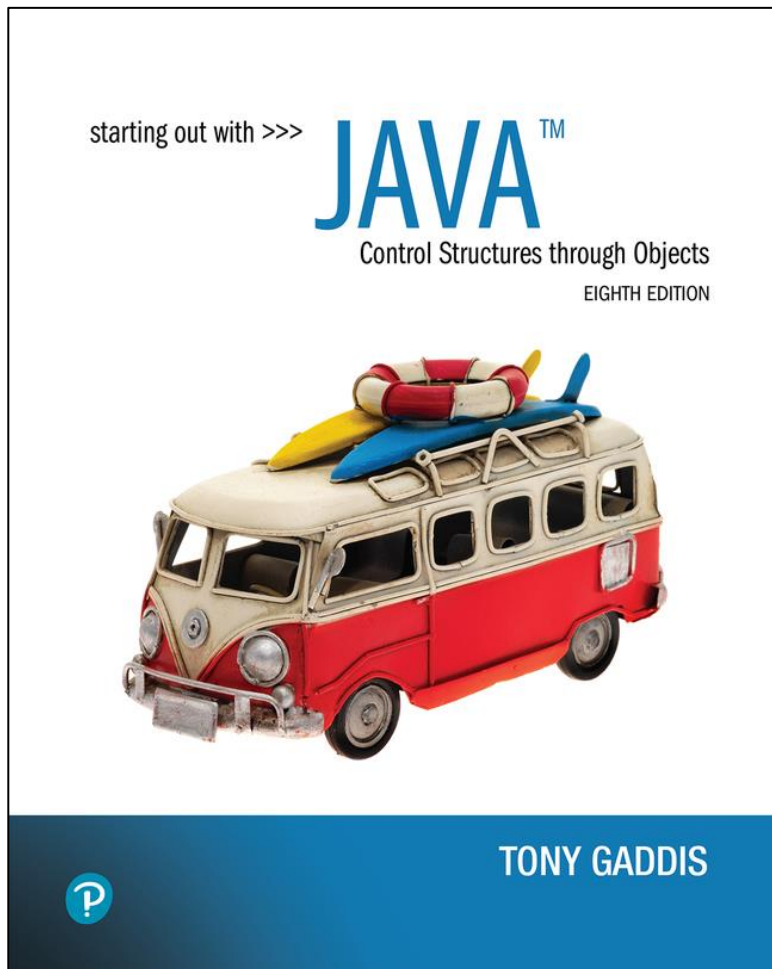


# Starting Out with Java Control Structures Through Objects

Eighth Edition



## Chapter 4

### Loops and Files

# Chapter Topics (1 of 2)

- Chapter 4 discusses the following main topics:
  - The Increment and Decrement Operators
  - The `while` Loop
  - Using the `while` Loop for Input Validation
  - The `do-while` Loop
  - The `for` Loop
  - Running Totals and Sentinel Values

# Chapter Topics (2 of 2)

- Chapter 4 discusses the following main topics:
  - Nested Loops
  - The `break` and `continue` Statements
  - Deciding Which Loop to Use
  - Introduction to File Input and Output
  - Generating Random Numbers with the `Random` class

# The Increment and Decrement Operators

- There are numerous times where a variable must simply be incremented or decremented.

```
number = number + 1;
```

```
number = number - 1;
```

- Java provide shortened ways to increment and decrement a variable's value.
- Using the ++ or -- unary operators, this task can be completed quickly.

```
number++; or ++number;
```

```
number--; or --number;
```

- Example: IncrementDecrement.java

# Differences Between Prefix and Postfix

- When an increment or decrement are the only operations in a statement, there is no difference between prefix and postfix notation.
- When used in an expression:
  - prefix notation indicates that the variable will be incremented or decremented prior to the rest of the equation being evaluated.
  - postfix notation indicates that the variable will be incremented or decremented after the rest of the equation has been evaluated.
- Example: Prefix.java

# The `while` Loop (1 of 2)

- Java provides three different looping structures.
- The `while` loop has the form:

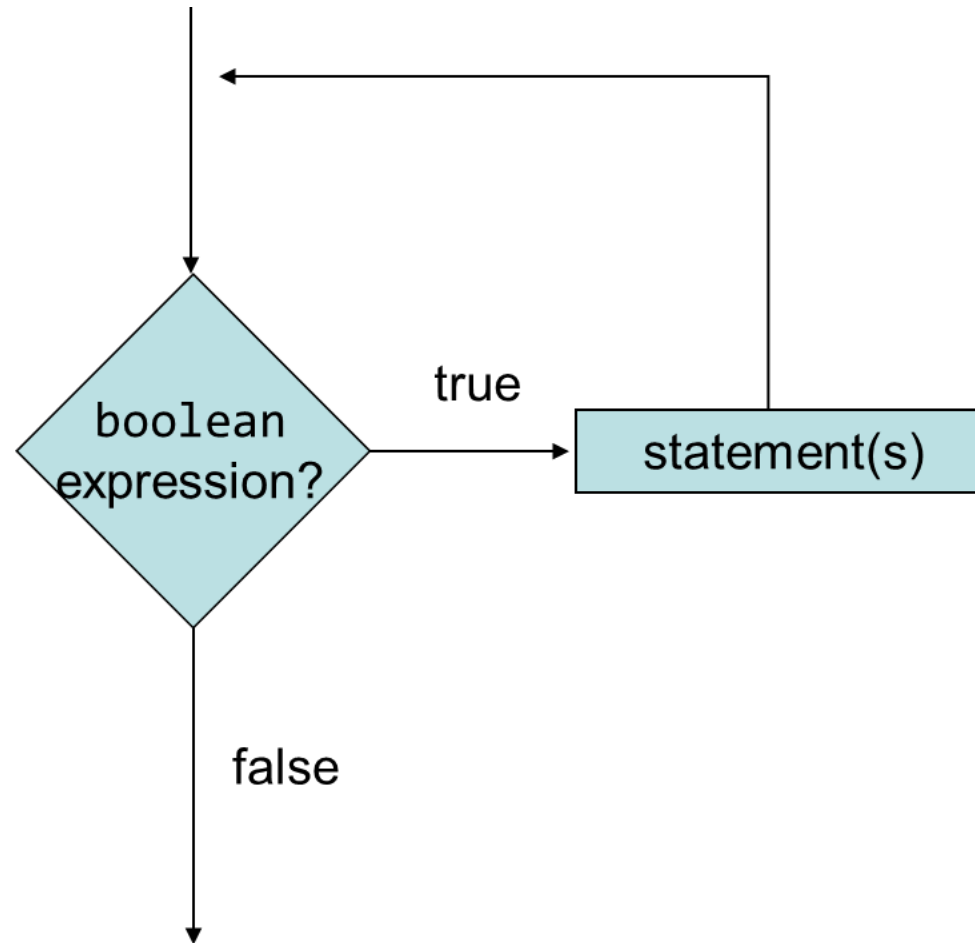
```
while (condition)  
{  
    statements;  
}
```

- While the condition is true, the statements will execute repeatedly.
- The `while` loop is a **pretest** loop, which means that it will test the value of the condition prior to executing the loop.

# The `while` Loop (2 of 2)

- Care must be taken to set the condition to false somewhere in the loop so the loop will end.
- Loops that do not end are called **infinite loops**.
- A `while` loop executes 0 or more times. If the condition is false, the loop will not execute.
- Example: `WhileLoop.java`

# The while Loop Flowchart





# Infinite Loops (1 of 2)

- In order for a `while` loop to end, the condition must become false. The following loop will not end:

```
int x = 20;
while(x > 0)
{
    System.out.println("x is greater than 0");
}
```

- The variable `x` never gets decremented so it will always be greater than 0.
- Adding the `x--` above fixes the problem.

# Infinite Loops (2 of 2)

- This version of the loop decrements `x` during each iteration:

```
int x = 20;
while(x > 0)
{
    System.out.println("x is greater than 0");
    x--;
}
```

# Block Statements in Loops

- Curly braces are required to enclose block statement while loops. (like block `if` statements)

```
while (condition)  
{  
    statement;  
    statement;  
    statement;  
}
```

# The while Loop for Input Validation

- **Input validation** is the process of ensuring that user input is valid.

```
System.out.print("Enter a number in the " +  
                "range of 1 through 100: ");  
number = keyboard.nextInt();  
// Validate the input.  
while (number < 1 || number > 100)  
{  
    System.out.println("That number is invalid.");  
    System.out.print("Enter a number in the " +  
                    "range of 1 through 100: ");  
    number = keyboard.nextInt();  
}
```

- Example: SoccerTeams.java

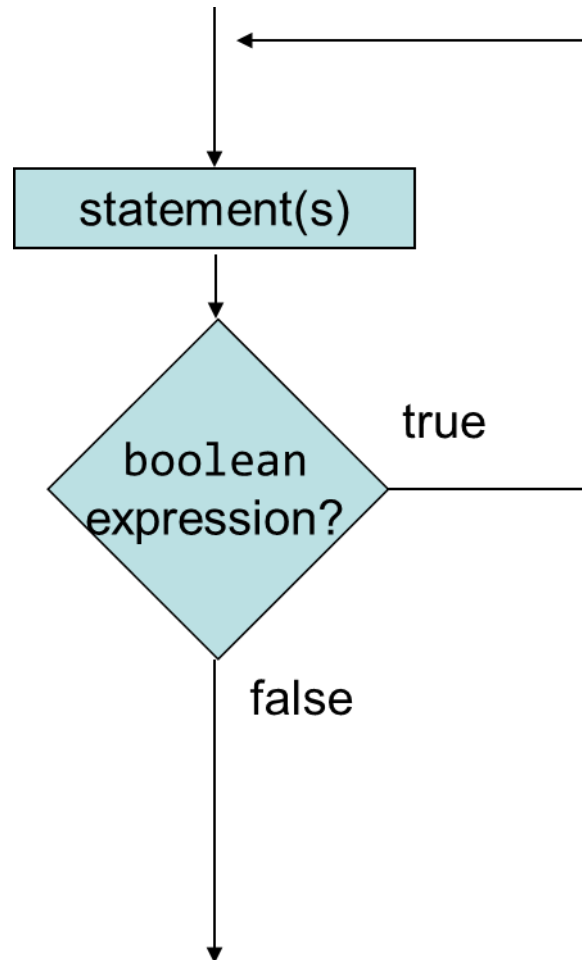
# The do-while Loop

- The `do-while` loop is a **post-test** loop, which means it will execute the loop prior to testing the condition.
- The `do-while` loop (sometimes called a do loop) takes the form:

```
do
{
    statement(s);
} while (condition);
```

- Example: TestAverage1.java

# The do-while Loop Flowchart



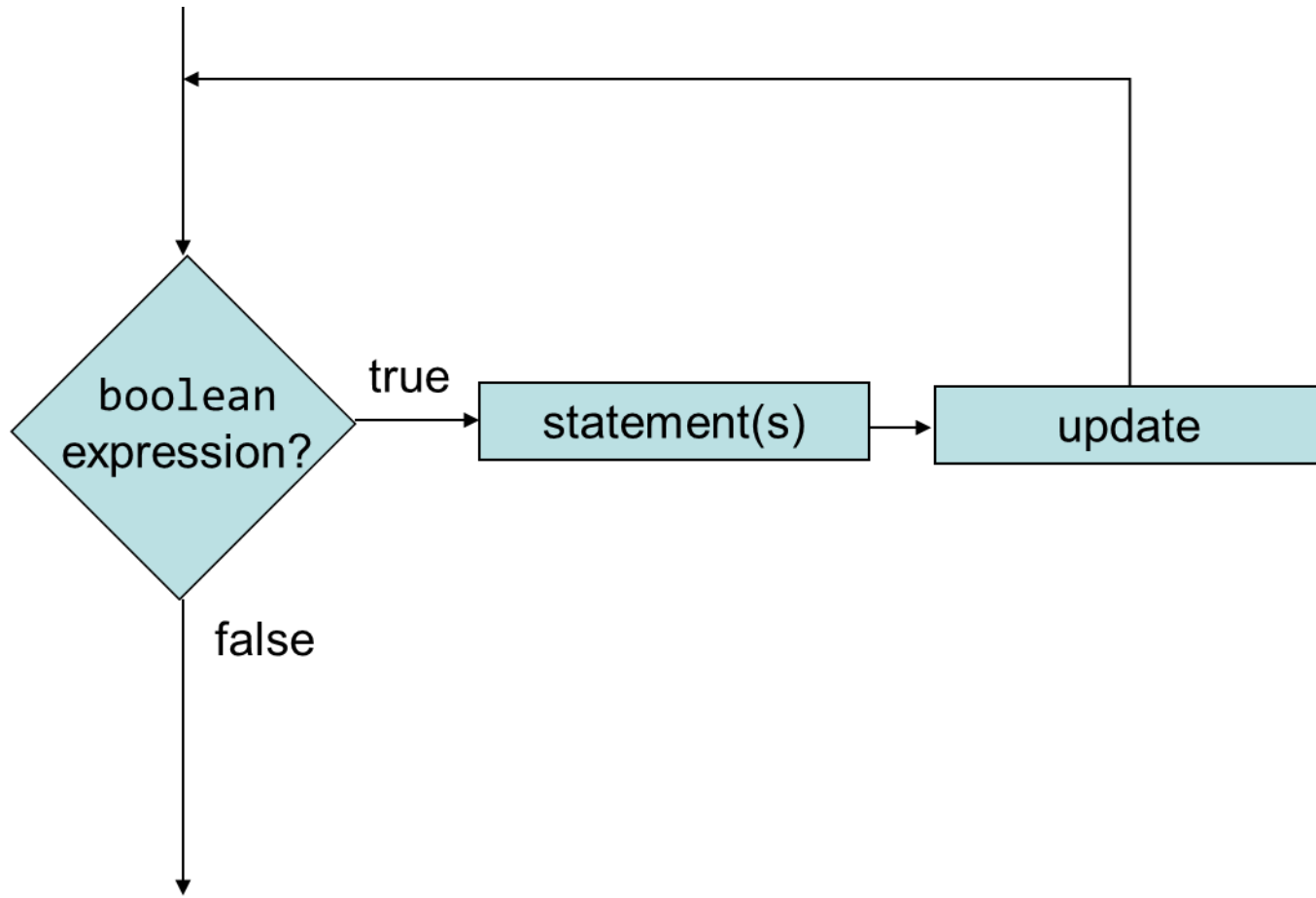
# The `for` Loop

- The `for` loop is a pre-test loop.
- The `for` loop allows the programmer to initialize a control variable, test a condition, and modify the control variable all in one line of code.
- The `for` loop takes the form:

```
for (initialization; test; update)  
{  
    statement(s);  
}
```

- See example: `Squares.java`

# The for Loop Flowchart





# The Sections of the `for` Loop

- The **initialization section** of the `for` loop allows the loop to initialize its own control variable.
- The **test section** of the `for` statement acts in the same manner as the condition section of a `while` loop.
- The **update section** of the `for` loop is the last thing to execute at the end of each loop.
- Example: `UserSquares.java`

# The `for` Loop Initialization

- The initialization section of a `for` loop is optional; however, it is usually provided.
- Typically, `for` loops initialize a counter variable that will be tested by the test section of the loop and updated by the update section.
- The initialization section can initialize multiple variables.
- Variables declared in this section have scope only for the `for` loop.

# The Update Expression

- The update expression is usually used to increment or decrement the counter variable(s) declared in the initialization section of the for loop.
- The update section of the loop executes last in the loop.
- The update section may update multiple variables.
- Each variable updated is executed as if it were on a line by itself.

# Modifying the Control Variable

- You should avoid updating the control variable of a `for` loop within the body of the loop.
- The update section should be used to update the control variable.
- Updating the control variable in the `for` loop body leads to hard to maintain code and difficult debugging.

# Multiple Initializations and Updates

- The `for` loop may initialize and update multiple variables.

```
for (int i = 5, int j = 0; i < 10 || j < 20; i++, j+=2)
{
    statement(s);
}
```

- Note that the only parts of a `for` loop that are mandatory are the semicolons.

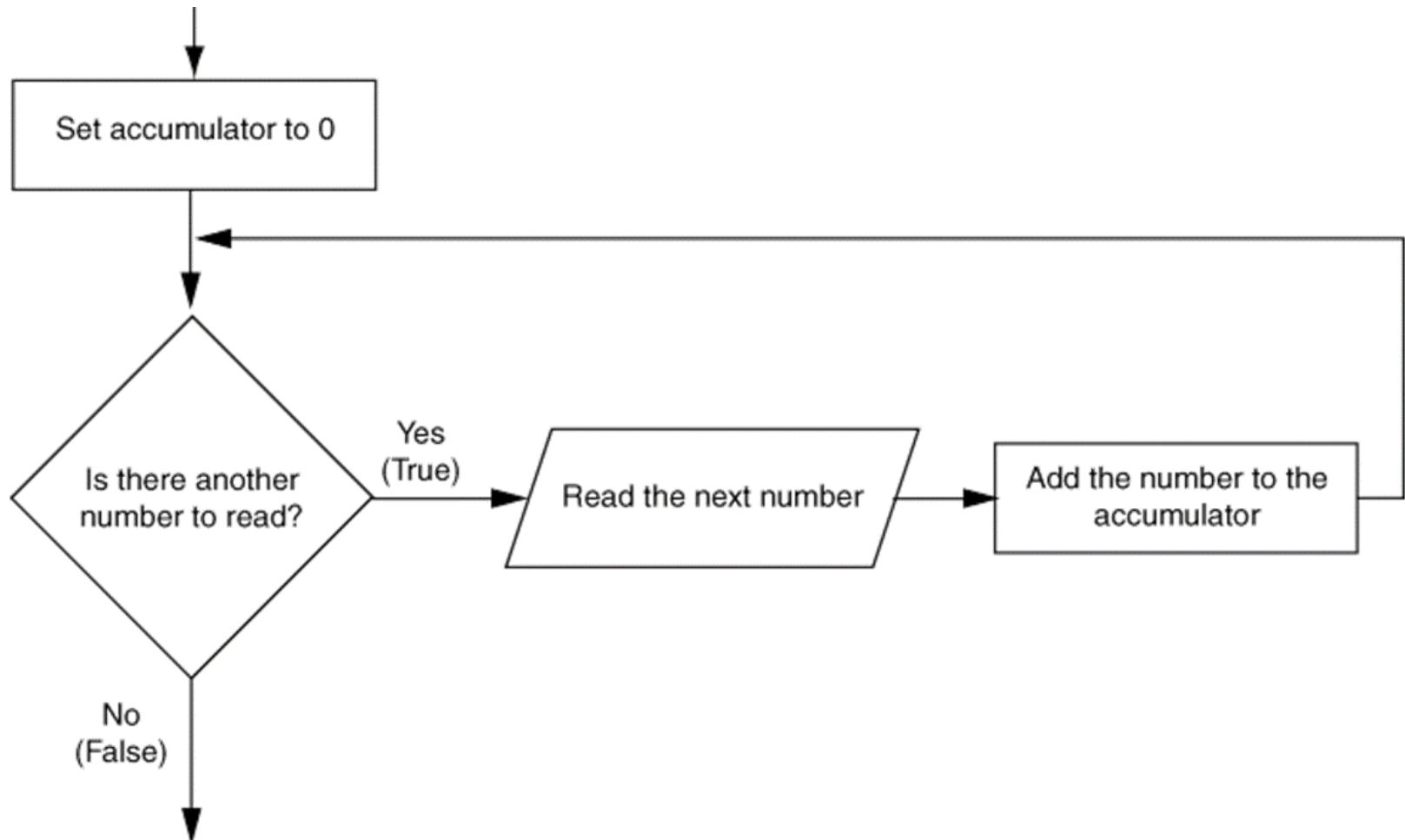
```
for (;;)
{
    statement(s);
} // infinite loop
```

- If left out, the test section defaults to true.

# Running Totals

- Loops allow the program to keep running totals while evaluating data.
- Imagine needing to keep a running total of user input.
- Example: TotalSales.java

# Logic for Calculating a Running Total



# Sentinel Values

- Sometimes the end point of input data is not known.
- A **sentinel value** can be used to notify the program to stop acquiring input.
- If it is a user input, the user could be prompted to input data that is not normally in the input data range (i.e.  $-1$  where normal input would be positive.)
- Programs that get file input typically use the end-of-file marker to stop acquiring input data.
- Example: SoccerPoints.java



# Nested Loops

- Like `if` statements, loops can be nested.
- If a loop is nested, the inner loop will execute all of its iterations for each time the outer loop executes once.

```
for (int i = 0; i < 10; i++)  
    for (int j = 0; j < 10; j++)  
        loop statements;
```

- The loop statements in this example will execute 100 times.
- Example: `Clock.java`

# The `break` Statement

- The `break` statement can be used to abnormally terminate a loop.
- The use of the `break` statement in loops bypasses the normal mechanisms and makes the code hard to read and maintain.
- It is considered bad form to use the `break` statement in this manner.

# The `continue` Statement

- The `continue` statement will cause the currently executing iteration of a loop to terminate and the next iteration will begin.
- The `continue` statement will cause the evaluation of the condition in `while` and `for` loops.
- Like the `break` statement, the `continue` statement should be avoided because it makes the code hard to read and debug.

# Deciding Which Loops to Use

- The `while` loop:
  - Pretest loop
  - Use it where you do not want the statements to execute if the condition is false in the beginning.
- The `do-while` loop:
  - Post-test loop
  - Use it where you want the statements to execute at least one time.
- The `for` loop:
  - Pretest loop
  - Use it where there is some type of counting variable that can be evaluated.


# File Input and Output

- Reentering data all the time could get tedious for the user.
- The data can be saved to a file.
  - Files can be **input files** or **output files**.
- Files:
  - Files have to be opened.
  - Data is then written to the file.
  - The file must be closed prior to program termination.
- In general, there are two types of files:
  - binary
  - text

# Writing Text to a File

- To open a file for text output you create an instance of the `PrintWriter` class.

```
PrintWriter outputFile = new PrintWriter("StudentData.txt");
```



**Pass the name of the file that you wish to open as an argument to the `PrintWriter` constructor.**

**Warning: if the file already exists, it will be erased and replaced with a new file.**

# The `PrintWriter` Class (1 of 3)

- The `PrintWriter` class allows you to write data to a file using the `print` and `println` methods, as you have been using to display data on the screen.
- Just as with the `System.out` object, the `println` method of the `PrintWriter` class will place a newline character after the written data.
- The `print` method writes data without writing the newline character.

# The PrintWriter Class (2 of 3)

**Open the file.**

```
PrintWriter outputFile = new PrintWriter("Names.txt");  
outputFile.println("Chris");  
outputFile.println("Kathryn");  
outputFile.println("Jean");  
outputFile.close();
```

**Close the file.**

**Write data to the  
file.**



# The `PrintWriter` Class (3 of 3)

- To use the `PrintWriter` class, put the following import statement at the top of the source file:

```
import java.io.*;
```

- See example: `FileWriteDemo.java`

# Exceptions (1 of 2)

- When something unexpected happens in a Java program, an **exception** is thrown.
- The method that is executing when the exception is thrown must either handle the exception or pass it up the line.
- Handling the exception will be discussed later.
- To pass it up the line, the method needs a `throws` clause in the method header.

# Exceptions (2 of 2)

- To insert a `throws` clause in a method header, simply add the word **throws** and the name of the expected exception.
- `PrintWriter` objects can throw an `IOException`, so we write the `throws` clause like this:

```
public static void main(String[] args) throws IOException
```

# Appending Text to a File (1 of 2)

- To avoid erasing a file that already exists, create a `FileWriter` object in this manner:

```
FileWriter fw = new FileWriter("names.txt", true);
```

- Then, create a `PrintWriter` object in this manner:

```
PrintWriter fw = new PrintWriter(fw);
```

# Appending Text to a File (2 of 2)

- The two statements can be combined into a single statement:

```
PrintWriter fw = new PrintWriter(  
    new FileWriter("names.txt", true));
```

# Specifying a File Location (1 of 2)

- On a Windows computer, paths contain backslash (\) characters.
- Remember, if the backslash is used in a string literal, it is the escape character so you must use two of them:

```
PrintWriter outFile =  
    new PrintWriter("D:\\PriceList.txt");
```

# Specifying a File Location (2 of 2)

- This is only necessary if the backslash is in a string literal.
- If the backslash is in a `String` object then it will be handled properly.
- Fortunately, Java allows Unix style filenames using the forward slash (/) to separate directories:

```
PrintWriter outFile = new  
    PrintWriter("/home/rharrison/names.txt");
```

# Reading Data From a File (1 of 4)

- You use the `File` class and the `Scanner` class to read data from a file:

**Pass the name of the file as an argument to the `File` class constructor.**

```
File myFile = new File("Customers.txt");  
Scanner inputFile = new Scanner(myFile);
```

**Pass the `File` object as an argument to the `Scanner` class constructor.**



# Reading Data From a File (2 of 4)

- You can declare the `File` object and the `Scanner` object in one statement:

```
Scanner inputFile = new Scanner(  
    new File("Customers.txt"));
```

# Reading Data From a File (3 of 4)

```
Scanner keyboard = new Scanner(System.in);  
System.out.print("Enter the filename: ");  
String filename = keyboard.nextLine();  
Scanner inputFile = new Scanner(new File(filename));
```

- The lines above:
  - Creates an instance of the `Scanner` class to read from the keyboard
  - Prompt the user for a filename
  - Get the filename from the user
  - Create an instance of the `File` class to represent the file
  - Create an instance of the `Scanner` class that reads from the file

# Reading Data From a File (4 of 4)

- Once an instance of Scanner is created, data can be read using the same methods that you have used to read keyboard input (`nextLine`, `nextInt`, `nextDouble`, etc).

```
// Open the file.  
Scanner inputFile = new Scanner(new File("Names.txt"));  
  
// Read a line from the file.  
String str = inputFile.nextLine();  
  
// Close the file.  
inputFile.close();
```

# Exceptions

- The `Scanner` class can throw an `IOException` when a `File` object is passed to its constructor.
- So, we put a `throws IOException` clause in the header of the method that instantiates the `Scanner` class.
- See Example: `ReadFirstLine.java`

# Detecting the End of a File (1 of 2)

- The `Scanner` class's `hasNext()` method will return `true` if another item can be read from the file.

```
// Open the file.
Scanner inputFile = new Scanner(new File(filename));

// Read until the end of the file.
while (inputFile.hasNext())
{
    String str = inputFile.nextLine();
    System.out.println(str);
}

inputFile.close(); // close the file when done.
```

# Detecting the End of a File (2 of 2)

- See example: `FileReadDemo.java`

# Managing Resources With the `try-with-resources` Statement (1 of 4)

- The `try-with-resources` statement can be used to open a file and automatically close the file.
- General format:

```
try (Declaration statement for file object)  
{  
    // Statements that work with the file...  
}
```

# Managing Resources With the `try-with-resources` Statement (2 of 4)

- Example:

```
try (PrintWriter outputFile = new PrintWriter("myfile.txt"))
{
    // Statements that write to the file...
}
```

- This example opens a file named `myfile.txt`
- Code inside the braces can use the `PrintWriter` object to write data to the file
- When the code inside the braces has finished, the file is automatically closed



# Managing Resources With the `try-with-resources` Statement (3 of 4)

- Example:

```
try (Scanner inputFile = new Scanner(new File("myfile.txt")))
{
    // Statements that read from the file...
}
```

- This example opens a file named `myfile.txt`
- Code inside the braces can use the `Scanner` object to read data from the file
- When the code inside the braces has finished, the file is automatically closed

# Managing Resources With the `try-with-resources` Statement (4 of 4)

- Opening two files with a `try-with-resources` statement:

```
try (Scanner inputFile = new Scanner(new File("File1.txt"));  
    PrintWriter outputFile = new PrintWriter("File2.txt"))  
{  
    // Statements that work with both files..  
}
```

- This example opens a file named `File1.txt` for reading and a file named `File2.txt` for writing
- Code inside the braces can use the `Scanner` object to read data from `File1.txt` and the `PrintWriter` object to write data to `File2.txt`
- When the code inside the braces has finished, both files are automatically closed

# Generating Random Numbers with the Random Class

- Some applications, such as games and simulations, require the use of randomly generated numbers.
- The Java API has a class, `Random`, for this purpose. To use the `Random` class, use the following `import` statement and create an instance of the class.

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

```
Random randomNumbers = new Random();
```

# Some Methods of the Random Class

Method	Description
<code>nextDouble()</code>	Returns the next random number as a <code>double</code> . The number will be within the range of 0.0 and 1.0.
<code>nextFloat()</code>	Returns the next random number as a <code>float</code> . The number will be within the range of 0.0 and 1.0.
<code>nextInt()</code>	Returns the next random number as an <code>int</code> . The number will be within the range of an <code>int</code> , which is $-2,147,483,648$ to $+2,147,483,648$ .
<code>nextInt(int n)</code>	This method accepts an integer argument, <code>n</code> . It returns a random number as an <code>int</code> . The number will be within the range of 0 to <code>n</code> .

See example: RollDice.java

# Copyright



**This work is protected by United States copyright laws and is provided solely for the use of instructors in teaching their courses and assessing student learning. Dissemination or sale of any part of this work (including on the World Wide Web) will destroy the integrity of the work and is not permitted. The work and materials from it should never be made available to students except by instructors using the accompanying text in their classes. All recipients of this work are expected to abide by these restrictions and to honor the intended pedagogical purposes and the needs of other instructors who rely on these materials.**