In the previous chapter, we explored decision structures that allowed our programs to make choices and follow different execution paths. Now we extend our programming capabilities with **loops**, which enable programs to repeat code block execution, and **file I/O**, which allows programs to store and retrieve data persistently. These concepts are essential for creating programs that can process large amounts of data, validate user input, and maintain information between program runs.

What you will learn

In this chapter, you will learn how to:

- Use **increment and decrement operators** (`++`, `--`) in prefix and postfix notation.
- Implement the three primary loop types: **`while`**, **`do-while`**, and **`for` loops**.
- Apply **advanced loop techniques** including running totals, input validation, and nested loops.
- Control loop execution with **`break` and `continue` statements**.
- Generate **random numbers** for simulations and games.
- Write data to files using **`PrintWriter`** and handle file exceptions properly.
- Read data from files using **`Scanner`** and **`File` objects**.
- Understand file paths and use **try-with-resources** for automatic resource management. # 4.1 The Increment and Decrement Operators

Before diving into loops, it is important to understand the shorthand operators used to add or subtract **one** from a variable.

The four ways to write these variable operations are:

Operator	Description
++	Increment variable by 1
–	Decrement variable by 1
+= 1	Add 1 and assign
-= 1	Subtract 1 and assign

For example:

```
int number = 5;
/*
The following statements increment
number by 1:
*/
number = number + 1;
number++;
++number;
```

```
number += 1;

/*
The following statements decrement
number by 1:
*/
number = number - 1;
number--;
--number;
number -= 1;
```

Differences between Prefix and Postfix Notation

The increment (`++`) and decrement (`--`) operators can be used in two modes: prefix (`++number`) or postfix (`number++`). While either can be used when the operator is the sole statement, they operate differently when used within a larger expression.

Postfix notation (`number++`) is the most common form, especially within loops.

- **Prefix Notation (`++number`)**: The variable is incremented or decremented *prior* to the rest of the equation being evaluated. Use this when the updated value is needed immediately.
- **Postfix Notation (`number++`)**: The variable is incremented or decremented *after* the rest of the equation has been evaluated. Use this when the original value is needed first.

Example using Postfix Notation:

```
int x = 5;
int y = x++;
// y is assigned 5 (original value), then x is incremented to 6

// x == 6
// y == 5
```

Example using Prefix Notation:

```
int x = 5;
int y = ++x;
// x is incremented to 6, then y is assigned 6 (updated value)

// x == 6
// y == 6
```

Check Your Understanding (4.1)

1. Explain the primary difference in timing between the prefix and postfix increment operators when used in an expression.
2. What are the four ways to increment a variable `count` by 1?

Practice Problems (4.1)

1. **Operator Evaluation:** What are the final values of `a` and `b` after the following code executes?

```
int a = 10;
int b = 5 + a--;
```

2. **Postfix Application:** Given `int x = 9;`, write a single Java statement using the postfix decrement operator that would set `y` to 9 and then set `x` to 8. # 4.2 Introduction to Loops (while, do-while, for)

Java provides three primary looping structures that enable a program to repeatedly execute a block of code. A loop must contain a mechanism to terminate, otherwise, it becomes an **infinite loop**.

Like `if` statements, all three types of Java loops include a Boolean test condition to determine whether to execute. The difference is that loops execute repeatedly as long as the Boolean expression evaluates to `true`. ## 4.2.1 The while Loop The `while` loop is a **pre-test loop**. This means the Boolean condition is tested *before* the loop body executes. If the condition is initially false, the loop body will never execute.

Use Case: Use the `while` loop when you need to repeat a block of code as long as a condition is true. It is often used for input validation.

Syntax and Example:

```
// While loop syntax -- general formula:
// Pretest loop: checks if condition is true first
while (condition) {
    // statements executed repeatedly
    // Must include code to make the condition false eventually!
}

// Example #1
int count = 1;
while (count <= 5) {
    System.out.println("Iteration: " + count);
    count++; // Statement that moves toward termination:
    // incrementing count will lead to a false condition.
}

// Example #2:
```

```java
String msg = "Learning Java is cool."
boolean isJavaCool = true;
Scanner kbd = new Scanner(System.in);
// Boolean expression that determines whether to execute the loop.
// ONLY executes if boolean expression is initially true.
while (isJavaCool) {
    System.out.println(msg)
    System.out.println("Do you agree? (Y/N)");
    char userAnswer = kbd.nextLine().toLower().charAt(0);
    if (userAnswer == 'n'){
        isJavaCool = false;
        //user can make the the test condition false and exit the loop.
    }
}
```

4.2.2 The do-while Loop

The `do-while` loop is a **post-test loop**. The code block executes **at least once** before the Boolean condition is evaluated.

Use Case: Use a `do-while` loop when you must ensure the loop body is executed at least once. This is useful for displaying a menu or prompting a user for input before validating it.

Syntax and Example:

```java
// Do While loop syntax -- general formula:
do {
    // Post-test loop: statements executed at least once
} while (condition);

// Example #1:
int count = 10; // Even though count > 5, it runs once.
do {
    System.out.println("Run at least once.");
    count++;
} while (count < 5);
```

A real world example could be when you need to prompt the user for input and ensure that the input meets certain criteria before proceeding. The prompt should be shown at least once.

```java
// Example #2:
int number;
do {
    System.out.print("Enter a positive number: ");
    number = scanner.nextInt();
} while (number <= 0);
```

Another common use for `do while` loops is when you want to display a menu to the user and perform actions based on the user's choice. The menu should be displayed at least once.

```
// Example #3:
int choice;
Scanner kbd = new Scanner(System.in);
do {
    System.out.println("1. Option 1");
    System.out.println("2. Option 2");
    System.out.println("3. Exit");
    System.out.print("Enter your choice: ");
    choice = kbd.nextInt();

    switch (choice) {
        case 1 -> System.out.println("You selected option 1.\n");
        case 2 -> System.out.println("You selected option 2.\n");
        case 3 -> System.out.println("Exiting...\n");
        default -> System.out.println("Invalid choice. Try again.\n");
    }
} while (choice != 3);
kbd.close();
```

4.2.3 The for Loop

The `for` loop is a **pre-test loop**; like the `while` loop, it will not execute if the initial condition is false.

For loops are ideal for **count-controlled loops** where the exact number of iterations is known beforehand. They allow the programmer to initialize a control variable, test a condition, and modify the control variable all in one line of code. The variable that controls the number of iterations is the **loop control variable**.

Syntax and Example:

A `for` loop consists of three sections in its header:

1. **Initialization**: Runs once at the start, often initializing the control variable.
2. **Test**: A Boolean expression checked before each iteration (pre-test).
3. **Update**: The last thing to execute at the end of each loop, typically modifying the control variable (e.g., incrementing/decrementing).

```
for (initialization; test; update) {
    // statements executed repeatedly
}

// Example #1: Looping 10 times
```

```java
for (int i = 0; i < 10; i++) {
    System.out.println("This is iteration number: " + i);
}

// Example #2: Printing number and number squared for 1-20 using printf
System.out.printf("%7s - %7s %n", "number", "n squared");
for (int i=1; i <= 20; i++){
    System.out.printf("%7d - %-7d %n", i, i*i);
}
```

Output of example #2:

```
 number - n squared
      1 - 1
      2 - 4
      3 - 9
      4 - 16
      5 - 25
      6 - 36
      7 - 49
      8 - 64
      9 - 81
     10 - 100
     11 - 121
     12 - 144
     13 - 169
     14 - 196
     15 - 225
     16 - 256
     17 - 289
     18 - 324
     19 - 361
     20 - 400
```

Check Your Understanding (4.2)

1. Which two Java loops are categorized as pre-test loops, and what does this mean for execution?
2. When is a `do-while` loop the preferred choice over a `while` loop?
3. Explain the function of a **sentinel value** and how it relates to loop termination.

Practice Problems (4.2)

1. **Loop Selection:** You need to write a program that repeatedly asks the user for a test score between 0 and 100 and will not accept any input

outside that range. Which loop structure is best suited for this task? Explain why.

2. **For Loop Execution:** How many times will the following `for` loop execute? `java for (int count = 1; count <= 12; count += 1) { // loop body }` # 4.3 Loop Usage and Techniques

4.3.1 Running Totals and Accumulators

A **running total** is a sum of numbers accumulated through multiple iterations of a loop. The variable used to keep this running total is called an **accumulator**. The accumulator variable must be initialized to zero before the loop begins.

Example:

```
int runningTotal = 0; // The accumulator, initialized to 0
Scanner kbd = new Scanner(System.in);
System.out.print("Type a number: ");

while (kbd.hasNextInt()) {
    System.out.print("Type a number (or non-number to quit): ");
    int nextNumber = kbd.nextInt();
    runningTotal += nextNumber; // Using combined assignment operator
}

System.out.println("Total sum: " + runningTotal);
kbd.close();
```

4.3.2 Input Validation

The `while` and `do while` loops are highly effective for ensuring user input meets specific criteria. This technique is known as a **user controlled loop**.

Example of Input Validation:

```
Scanner keyboard = new Scanner(System.in);
int score;

// Loop keeps running as long as the score is outside the valid range (0-100)
do {
    System.out.print("Enter test score (0-100): ");
    score = keyboard.nextInt();
} while (score < 0 || score > 100);
```

4.3.3 Initializing Multiple Variables in a for Loop

It is possible to initialize and update multiple variables within the single line of a `for` loop header, though this is not commonly seen. This technique is useful for managing multiple variables within one loop for efficiency and synchronization.

Example of Multi-Variable Loop:

```
// i increments starting from 0, j decrements starting from 10
for (int i = 0, j = 10; i < j; i++, j--) {
    System.out.println("i: " + i + ", j: " + j);
}
```

4.3.4 Nested Loops

A **nested loop** occurs when one loop is placed inside the body of another loop. Nested loops are essential when operations require multiple levels of iteration, such as working with multi-dimensional arrays, generating combinations, or printing patterns.

Example of Nested Loops (Printing a Pattern):

```
// Example #1:
// Outer loop controls rows
for (int row = 1; row <= 5; row++) {
    // Inner loop controls columns (prints stars)
    for (int col = 1; col <= row; col++) {
        System.out.print("*");
    }
    System.out.println(); // Move to the next row
}

// Output:

*
**
***
****
*****
```

Example of Nested Loops (Clock):

In this nested loop, the outer loop increments the hours on the clock, while the inner loop increments the minutes.

```
// Example #2: Clock loop
for (int hours = 1; hours <= 12; hours++){
    for (int minutes = 0; minutes <= 59; minutes++){
        System.out.printf("%02d:%02d\n", hours, minutes);
    }
}
```

The outer loop increments only after the inner loop condition becomes false (The inner condition testing for whether `minutes <= 59`).

4.3.5 The break and continue Statements

The `break` and `continue` statements offer ways to alter the normal execution flow of a loop. * **`break`**: This statement immediately terminates the loop (or `switch` statement) it is contained within, causing the program to proceed to the statement immediately following the loop. * **`continue`**: This statement causes the program to skip the remainder of the current loop iteration and immediately start the next iteration.

Examples:

```
// Example #1: Use break to escape a for loop
for (int i=0; i<10; i++){
    if (i == 5){
        break;
    }
    System.out.println(i);
}

// Output:
0
1
2
3
4


// Example #2: Use continue to skip the current loop iteration
for (int i=0; i<10; i++){
    if (i == 5){
        continue;
    }
    System.out.println(i);
}

// Output:
0
1
2
3
4
6
7
8
9
```

Check Your Understanding (4.3)

1. In the context of running totals, what is an accumulator, and what value should it be initialized to?
2. Give an example scenario where using a nested `for` loop would be necessary.

Practice Problems (4.3)

1. **Loop Tracing:** Trace the following code and determine the final value of `x`.

```
int x = 10;
while (x < 50) {
    x += 10;
}
```

2. **Code Interpretation:** Describe what the following nested loop code would produce:

```
for (int i = 1; i <= 2; i++) {
    for (int j = 1; j <= 3; j++) {
        System.out.print(i + "." + j + " ");
    }
}
```

4.4 Generating Random Numbers

Many applications, such as games or simulations, require the generation of randomly generated numbers. Java (via the Java API) provides the `Random` class for this purpose.

To use the `Random` class, you must include the import statement and create an instance of the class:

```
import java.util.Random;

Random randomNumbers = new Random(); // Creating an instance
```

Using the nextInt() and nextDouble() methods

The `nextInt()` method can be used to generate random integers. To generate a number within a specific range, bounds are specified.

To generate a random integer in a range starting from 0 up to (but not including) a specified bound:

```
// Generates random int from 0 up to (but not including) 10
int randomNumber = randomNumbers.nextInt(10);
```

To generate a random number within an arbitrary range (e.g., 1 through 6, simulating a game die):

```
// Formula:
// For 1 through 6: nextInt(6) + 1
int dieValue = randomNumbers.nextInt(6) + 1; // Range 1 to 6

// For 1 through 10: nextInt(10) + 1
```

There are several methods for generating random number types using an instance of Random:

```
// Returns the next random number as a double.
// The number will be within the range of 0.0 and 1.0.
randomNumbers.nextDouble();

// Returns the next random number as a float.
// The number will be within the range of 0.0 and 1.0.
randomNumbers.nextFloat();

// Returns the next random number as an int.
// The number will be within the range of an int, which is -2,147,483,648 to +2,147,483,648
randomNumbers.nextInt();

// This method accepts an integer argument, n.
// It returns a random number as an int.
// The number will be within the range of 0 to n-1.
randomNumbers.nextInt(int n);

// This method accepts two integer arguments, start and bound.
// It returns a random number as an int within the range of start to boundary-1.
randomNumbers.nextInt(int start, int bound);
```

Random Class vs. Math.random()

The `Random` class and the static `Math.random()` method differ in their features and control.

Feature	Random Class	Math.random() Method
Instantiation	Instance-based (requires `new Random()`)	Static method (call without creating an instance)
Return Type	Provides methods for `int`, `double`, `boolean`, `float`, and `long`	Only generates a `double`
Range	Customizable range and type	Always between 0.0 (inclusive) and 1.0 (exclusive)

Feature	Random Class	Math.random() Method
Seeding	Allows setting a seed for reproducibility	No control over seed

The following example shows the differences between `Math.random()` and the `Random` class:

```
// Must import the Random class:
import java.util.Random;
// The Random class can generate a random int or double:
Random r = new Random();
int randomInt = r.nextInt(100);  // Random int between 0 and 99
double randomDouble = r.nextDouble();  // Random double between 0.0 and 1.0


// Math.random() can only generate a double:
double rMath = Math.random();  // Random double between 0.0 and 1.0
// You can multiply the value by 100 to turn it into an int.
// there are many ways to represent this:
int rMathInt = (int) rMath * 100
rMathInt = Math.round(rMath*100));
rMathInt = Integer.parseInt(String.format("%.0f", rMath*100));
```

In summary:

- **`Random` class**: Offers more flexibility, including generating random integers, floats, booleans, and long values, with an optional seed.
- **`Math.random()`**: A simpler way to generate a random `double` between `0.0` and `1.0` with no need for creating an instance but ßless control over the result.

For most purposes, if you need various types of random values, use the `Random` class. If you just need a quick random double, `Math.random()` is convenient. # 4.5 File Input and Output Overview

File input and output (I/O) allows data to be saved to files for persistence, meaning the data remains after the program terminates. File operations involve three steps: opening, reading/writing, and closing.

The core Java I/O objects are `PrintWriter`, `FileWriter`, `File`, and `Scanner`.

The objects used for file I/O must be imported from the `java.io` package, while Scanner is imported from `java.util`:

```
import java.util.Scanner;   // Needed for Scanner class
import java.io.*;           // Needed for File I/O classes
```

4.5.1 PrintWriter: Writing Data to a File

The `PrintWriter` class is used to open a file for text output and write data using the familiar `print`, `println`, and `printf` methods.

> **Warning:** If the file already exists, creating a new `PrintWriter` instance by passing only the filename argument will **OVERWRITE** the existing file.

The process for writing data to a file using PrintWriter is as follows:

1. To open a file for text output you create an instance of the PrintWriter class. You pass the name of a file that you want to open as an argument to the PrintWriter object constructor. For example: `PrintWriter outputFile = new PrintWriter("StudentData.txt");`
   - Unless you specify a directory, the file is written in the location where the Java class is run from.
2. The PrintWriter class allows you to write data to a file using the `print`, `println`, and `printf` methods.
   - 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 `printf` method is also supported, allowing you to insert formatted Strings into the file.

> **Mnemonics tip**: It's called **PRINT**-writer because it has `print` and `println` methods like `System.output`.

Basic Example of using PrintWriter (Overwriting):

```
import java.io.*; // Required import
public class WriteStudentData {
    public static void main(String[] args) throws FileNotFoundException {
        PrintWriter outputFile = new PrintWriter("StudentData.txt");
        // OVERWRITES if file exists

        outputFile.println("Stephen"); // Writes "Stephen" followed by a newline
        outputFile.println("Patrick");
        outputFile.printf("%20s %n", "Morrissey"); //printf syntax is supporrted.

        // Crucial step: must close the PrintWriter object!
        outputFile.close();
    }
}
```

Remember, if you run the program multiple times, you overwrite the file each time.

Here's another example of writing data to a file using PrintWriter:

```
import java.io.*;
public class CreateMusicArtistsFile {
    public static void main(String[] args) throws FileNotFoundException {
        PrintWriter outputFile = new PrintWriter("MusicArtists.txt");
        outputFile.println("The Cure");
        outputFile.println("The Smiths");
        outputFile.println("Joy Division");
        outputFile.println("Green Day");
        outputFile.println("The The");
        outputFile.println("The Beach Boys");
        outputFile.println("Talking Heads");
        outputFile.println("Neutral Milk Hotel");
        outputFile.println("The Magnetic Fields");
        outputFile.println("The Velvet Underground");
        outputFile.close();
    }
}
```

Try to work through the following example, which uses a Scanner, a `for` loop, and non-self-contained initializer variable (because we want to access it outside the scope of the loop):

```
import java.io.*;
import java.util.Scanner;

public class PrintWriterTesting {
    public static void main(String[] args) throws FileNotFoundException {
        int i;
        Scanner kbd = new Scanner(System.in);
        PrintWriter outputFile = new PrintWriter("PrintStudentData.txt");
        for (i=1;i<4;i++){
            System.out.print("Enter student " + i + "'s name: ");
            String studentName = kbd.nextLine();
            outputFile.println(studentName);
        }
        System.out.println("Wrote " + (i-1) + " student names to file.");
        outputFile.close();
    }
}
```

4.5.2 Handling File Exceptions

When performing file operations, unexpected events called **exceptions** can occur, such as the program being unable to create or access a file. Java requires methods that perform file I/O to handle these exceptions.

The simplest way for a method (like `main`) to handle an exception is to pass the

responsibility up the line by using a **`throws` clause** in the method header.

- The `PrintWriter` class throws a `FileNotFoundException`.
- The `FileWriter` class throws an `IOException`.

Since `FileNotFoundException` is a specific type of `IOException`, you can often declare only `throws IOException` to cover both cases.

Example using a throws clause:

```
public static void main(String[] args) throws IOException {
    // Use IOException to cover most file errors
    // File writing code here
}
```

4.5.3 Try / Catch Exception Handling

An alternative way to handle file exceptions (such as when your program can't access a file or it doesn't exist) is to wrap the accessor code in a try / catch block, for example:

```
import java.io.*;

public class PrintWriterDemo {
    public static void main(String[] args) {
        try {
            PrintWriter outputFile = new PrintWriter("");
            outputFile.println("Carrie");
            outputFile.println("Donnie");
            outputFile.println("Mike");
            outputFile.close();
        } catch (Exception e){
            System.out.println("Can't write to this location \n" + e );
        }
    }
}

// Output:

Can't write to this location
java.io.FileNotFoundException:  (No such file or directory)
```

The `catch` block works similarly to an `else` clause in an `if / else` statement. The program will try to open the file (in this case `""`, an invalid filename). Since the program can't open it, the Exception is "thrown", which is to say, printed to the console.

4.5.4 FileWriter: Appending Data to a File

To prevent overwriting a file and instead **append** new data to its end, you must use the `FileWriter` class in conjunction with `PrintWriter`.

The key difference is that the `FileWriter` constructor can accept a second boolean argument. Setting this argument to `true` enables appending:

Simple Example of Appending Data:

```
public static void main(String[] args) throws IOException {
    // 1. Create a FileWriter instance, specifying 'true' to append
    FileWriter fwriter = new FileWriter("names.txt", true);

    // 2. Pass the FileWriter object to the PrintWriter constructor
    PrintWriter outputFile = new PrintWriter(fwriter);

    outputFile.println("Charlie");
    outputFile.close();
}
// Running this multiple times adds "Charlie" each time instead of overwriting.
```

A slightly more complex example

In this program, a while loop continually prompts the user to enter usernames and passwords to a file called `UserData.txt`. The program ends when the user answers "No" to the `Add another user?` prompt.

```
import java.io.*;
import java.util.Scanner;

public class CreatePasswordFile {
    public static void main(String[] args) throws IOException {
        FileWriter fwriter = new FileWriter("UserData.txt", true);
        PrintWriter outputFile = new PrintWriter(fwriter);
        Scanner kbd = new Scanner(System.in);
        boolean addMoreUsers = true;

        while (addMoreUsers){
            System.out.print("Enter username: ");
            String username = kbd.next();

            System.out.print("Enter password: ");
            String password = kbd.next();

            outputFile.printf("username: %10s | password: %10s %n", username, password);
            System.out.printf("Added %s. %nAdd another user? (Y/n) %n", username);
            if (kbd.next().toLowerCase().charAt(0) == 'n'){
                addMoreUsers = false;
```

```
            }
        }

        outputFile.close();
        kbd.close();
    }
}
```

On subsequent program executions, the file is continuously appended-to, rather than overwritten.

4.5.5 Specifying File Location

On Windows, file paths use the backslash (\) character. Because the backslash is Java's escape character (e.g., `\n`), you must use two backslashes (`\\`) in a string literal to represent a single backslash in a path.

For example:

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

This would save or create a file called `PriceList.txt` at the root of your Windows `D:` drive (if you have one).

A location you may want to use is your OneDrive account. You would specify the path similar to the following:

```
PrintWriter outFile = new PrintWriter("C:\\Users\\your_username\\OneDrive - Bentley Universi
```

To make file handling more portable across different operating systems, you can use system properties like `System.getProperty("user.home")` to find the user's home directory. For example:

```
System.getProperty("user.home")

// On Mac: /Users/your_username
// On Windows: C:\Users\your_username
```

Check Your Understanding (4.5)

1. What is the default behavior of the `PrintWriter` class if the output file already exists, and how can this be changed to append data instead?
2. When performing file I/O, why is it necessary to declare `throws IOException` or a similar exception in the method header?
3. What happens if you do not call the `close()` method on a `PrintWriter` object after writing data?

Practice Problems (4.5)

1. **File I/O Setup:** Write the code statements required to open a file named `log.txt` for *appending* data using `FileWriter` and `PrintWriter`, ensuring you include the correct exception handling in the `main` method header.
2. **Path Specification:** Why would the following file path declaration cause an error if not preceded by the `throws` clause? `PrintWriter logFile = new PrintWriter("C:\Users\Admin\Desktop\data.txt");`

4.6 File Input: Reading Data from a File

To read data from a file, you combine the `File` object and the `Scanner` object.

4.6.1 Reading Process

1. Create a `File` object, passing the filename of an existing file as an argument.
2. Create a `Scanner` object, passing the `File` object to its constructor.
3. Use `Scanner` methods (`nextInt()`, `nextDouble()`, `nextLine()`) to read data.

Example of Reading Data:

```
import java.io.*;
import java.util.Scanner;

public static void main(String[] args) throws FileNotFoundException {
    // Create a File object. In this case, the file should already exist.
    File myFile = new File("MusicArtists.txt");
    // Create a Scanner object to read the file.
    Scanner inputFile = new Scanner(myFile);
    // Assign the first line of the file to a String variable:
    String artist = inputFile.nextLine();
    System.out.println("First artist: " + artist);
    // Close the Scanner object!
    inputFile.close();
}
```

4.6.2 Checking for the End of the File

When reading data from a file, you typically use a loop to process items sequentially until there is no more data left. The `Scanner` object provides the `hasNext()` method to check if the file contains more data to read.

Example of Reading until End-of-File:

```
while (inputFile.hasNext()) { // Loop continues as long as there is more data
    String data = inputFile.nextLine();
    System.out.println(data);
}
// Note: Scanner also has methods like hasNextInt() and hasNextDouble()
```

Here's an example of reading the `MusicArtists` file until the end:

```
import java.io.*;
import java.util.Scanner;

public class ReadResources {
    public static void main(String[] args) throws IOException {
        int currentLine = 1;
        Scanner inputFile = new Scanner(new File("MusicArtists.txt"));
        while(inputFile.hasNext()){
            String str = inputFile.nextLine();
            System.out.printf("%-5d %s %n", currentLine, str);
            currentLine++;
        }
        inputFile.close();
    }
}
```

While you are reading a file, you can perform processing or calculations on each line in the file. For example, you could read each line for a specific keyword, and the print the line:

```
while (inputFile.hasNext()) {
    String str = inputFile.nextLine();
    if (str.contains("password")){
        System.out.println("Found password!!");
    }
    // Print out the line that contains the password:
    System.out.println(str);
}
```

If you have your `UserData.txt` file handy, you can apply this technique to print out each line that contains a password:

```
import java.io.File;
import java.io.IOException;
import java.util.Scanner;

public class PilferPasswords {
    public static void main(String[] args) throws IOException{
        Scanner inputFile = new Scanner(new File("UserData.txt"));
        while (inputFile.hasNext()) {
            String str = inputFile.nextLine();
```

```
            if (str.contains("password")){
                System.out.println("Found password!!");
            }
            // Print out the line that contains the password:
            System.out.println(str);
        }
        inputFile.close();
    }
}
```

4.6.3 File Object Methods

In previous sections, we created a File object to serve as input to a Scanner, then we used Scanner methods such as `.nextLine()` and `.hasNext()` to read from the file.

The File class itself has other interesting methods that are worthy of exploring. For example, in addition to creating files, you can also use the File class to create directories. You can also use File class methods to get information about files, such as the file size, the absolute filepath, and other attributes.

See the following code sample to learn more about File methods. Feel free to run the code on your computer and substitute your own files to keep exploring this powerful Java object:

```
import java.io.*;

/* Uses File methods such as:
      getName()
      getAbsolutePath()
      exists()
      mkdir()
      delete()
      renameTo()
      length() (size in bytes)
*/

public class FileMethodsDemo {
    public static void main(String[] args) throws IOException {
// Create or Overwrite data to a file called friends.txt in current directory.
        File oldFile = new File("friends.txt");
// A file can also be a directory if no file type is specified.
        File dir = new File("testDirectory");

// Add file contents using PrintWriter.
        PrintWriter fileContents = new PrintWriter(oldFile);
        fileContents.println("First line of the file.");
```

```
        fileContents.printf("Second line of the file called %s %n", oldFile.getName());
        fileContents.printf("Absolute path of this original file: %s %n", oldFile.getAbsolut
        fileContents.print("This is the last line of the file! \n");

// File isn't written until PrintWriter is closed.
        System.out.printf("Size of UNCLOSED file in bytes: %d %n", oldFile.length());

        fileContents.close();
// File gets written with new content after PrintWriter is closed.
        System.out.printf("Size of CLOSED file in bytes: %d %n", oldFile.length());

// The File object has exists() and mkdir() methods.
// Use a Unary operator to say: if directory does not exist, then make it.
        if (!dir.exists()){
            dir.mkdir();
        }
// Specify the destination location for the file
        File newFile = new File(dir + "/" + oldFile);

        if (newFile.exists()){
            newFile.delete();
// Move file to new directory.
            oldFile.renameTo(newFile);
        }

        System.out.println("Old filepath: " + oldFile.getAbsolutePath());
        System.out.println("New filepath: " + newFile.getAbsolutePath());

        System.out.println("Can new file be read? " + newFile.canRead());
        System.out.println("Can new file be written to? " + newFile.canWrite());
        System.out.println("Can new file be executed? " + newFile.canExecute());
    }
}
```

4.6.4 The try-with-resources Statement

The **`try-with-resources`** statement is highly recommended for file I/O because it automatically closes the resources (like `PrintWriter` or `Scanner` objects) once the code block finishes, regardless of whether an exception occurred. This eliminates the need for manual `close()` calls.

Example using try-with-resources for writing:

```
// File is automatically closed when the try block finishes
try (PrintWriter outputFile = new PrintWriter("Output.txt")) {
    outputFile.println("Data to write.");
}
```

Example using try-with-resources for reading:

```
File myFile = new File("Input.txt");

try (Scanner inputFile = new Scanner(myFile)) {
    while (inputFile.hasNext()) {
        System.out.println(inputFile.nextLine());
    }
}
```

In the following example (which is amended version of the earlier `CreateMusicArtistsFile` example program), the program opens (or creates) a file called `MusicArtists.txt` and writes data to it. When all of the statements inside the `try` block execute, the file is automatically closed and saved:

```
import java.io.*;
public class CreateMusicArtistsFile {
    public static void main(String[] args) throws FileNotFoundException {
        try (PrintWriter outputFile = new PrintWriter("MusicArtists.txt")){
            outputFile.println("The Cure");
            outputFile.println("The Smiths");
            outputFile.println("Joy Division");
            outputFile.println("Green Day");
            outputFile.println("The The");
            outputFile.println("The Beach Boys");
            outputFile.println("Talking Heads");
            outputFile.println("Neutral Milk Hotel");
            outputFile.println("The Magnetic Fields");
            outputFile.println("The Velvet Underground");
        }
    }
}
```

We can then use a Scanner object to read directly from this file and print out its contents to the console:

```
import java.io.*;
import java.util.Scanner;

public class TryReadWithResources {
    public static void main(String[] args) throws FileNotFoundException{
        int currentLine = 1;
        try (Scanner inputFile = new Scanner(new File("MusicArtists.txt"))){
            while(inputFile.hasNext()){
                String str = inputFile.nextLine();
                System.out.printf("%-5d %s %n", currentLine, str);
                currentLine++;
            }
```

```
        }
    }
}
```

We can do more complex data processing using our Java objects. For example, we could read from the MusicArtists file and count the number of bands with the word `The` in their name.

```
import java.io.*;
import java.util.Scanner;

public class TryReadAndProcessWithResources {
    public static void main(String[] args) throws FileNotFoundException{

        int theAccumulator = 0;
        int bandsWithTheInTheName = 0;
        int bandsWithoutTheInTheName = 0;
        int currentLine = 1;
        try (Scanner inputFile = new Scanner(new File("MusicArtists.txt"))){
            while(inputFile.hasNext()){
                String str = inputFile.nextLine();
                System.out.printf("%-5d %s %n", currentLine, str);
                if (str.contains("The")){
                    bandsWithTheInTheName++;

                    int firstThe = str.indexOf("The");
                    int secondThe = str.lastIndexOf("The");
                    if (firstThe != secondThe){
                        theAccumulator+=2;
                    } else {
                        theAccumulator++;
                    }
                } else {
                    bandsWithoutTheInTheName++;
                }
                currentLine++;
            }
        }

        System.out.printf("%n%-30s %d %n", "Bands with \"The\" in name:", bandsWithTheInThe
        System.out.printf("%-30s %d %n", "Bands without \"The\" in name:", bandsWithoutTheIn
        System.out.printf("%-30s %d %n", "Total \"The\"s:", theAccumulator);
    }
}
```

Check Your Understanding (4.6)

1. Which method should you use on a file `Scanner` object within a `while` loop to ensure you read data until the end of the file is reached?
2. Describe the primary benefit of using the `try-with-resources` statement for file I/O operations.

Practice Problems (4.6)

1. **File Reading Setup:** Write the minimum required Java statements (excluding imports and exception handling) to set up reading data from a file named `Scores.txt`.
2. **Data Retrieval:** Assuming `fileScanner` references a `Scanner` object opened for reading a file, write the single statement required to read the next integer value from the file and store it in an `int` variable named `value`.