Chapter 4:
Loops and Files

Starting Out with Java:
From Control Structures through Objects

Fifth Edition

by Tony Gaddis

Chapter Topics

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

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

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

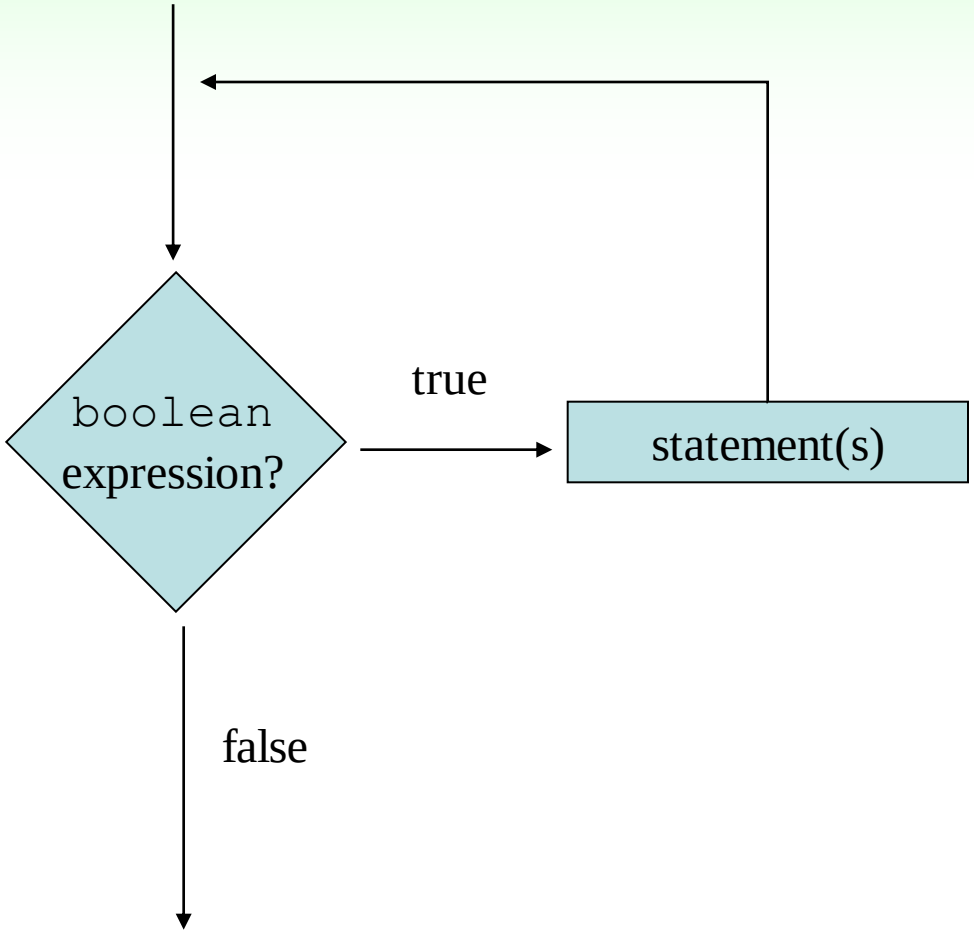
```
while(condition)
{
    statements;
}
```

- 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

- 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

- 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

- 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

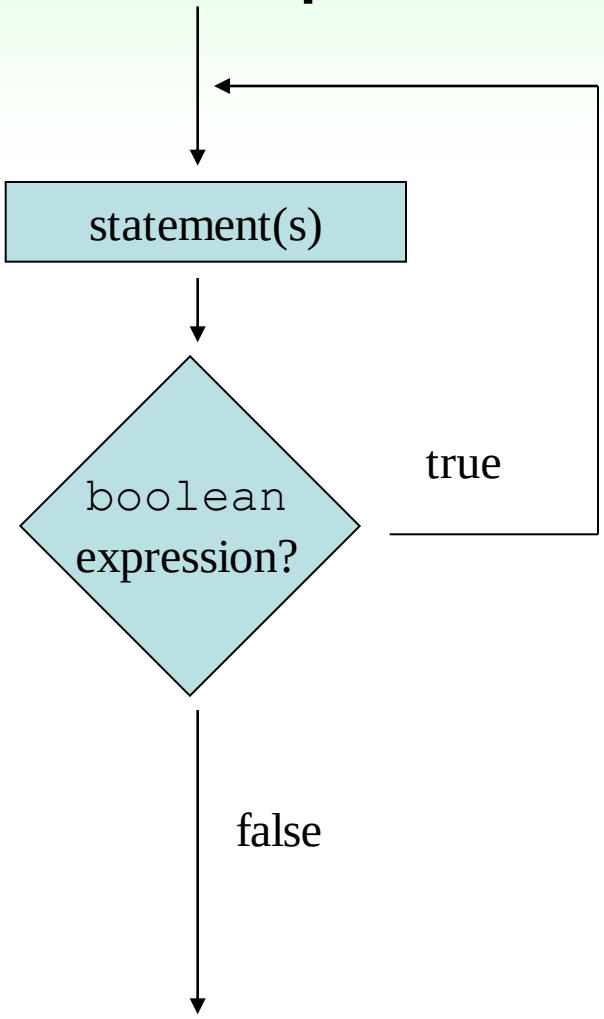
The `do-while` Loop

- 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 called a `do` loop) takes the form:

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

- Example: TestAverage1.java

The `do-while` Loop Flowchart



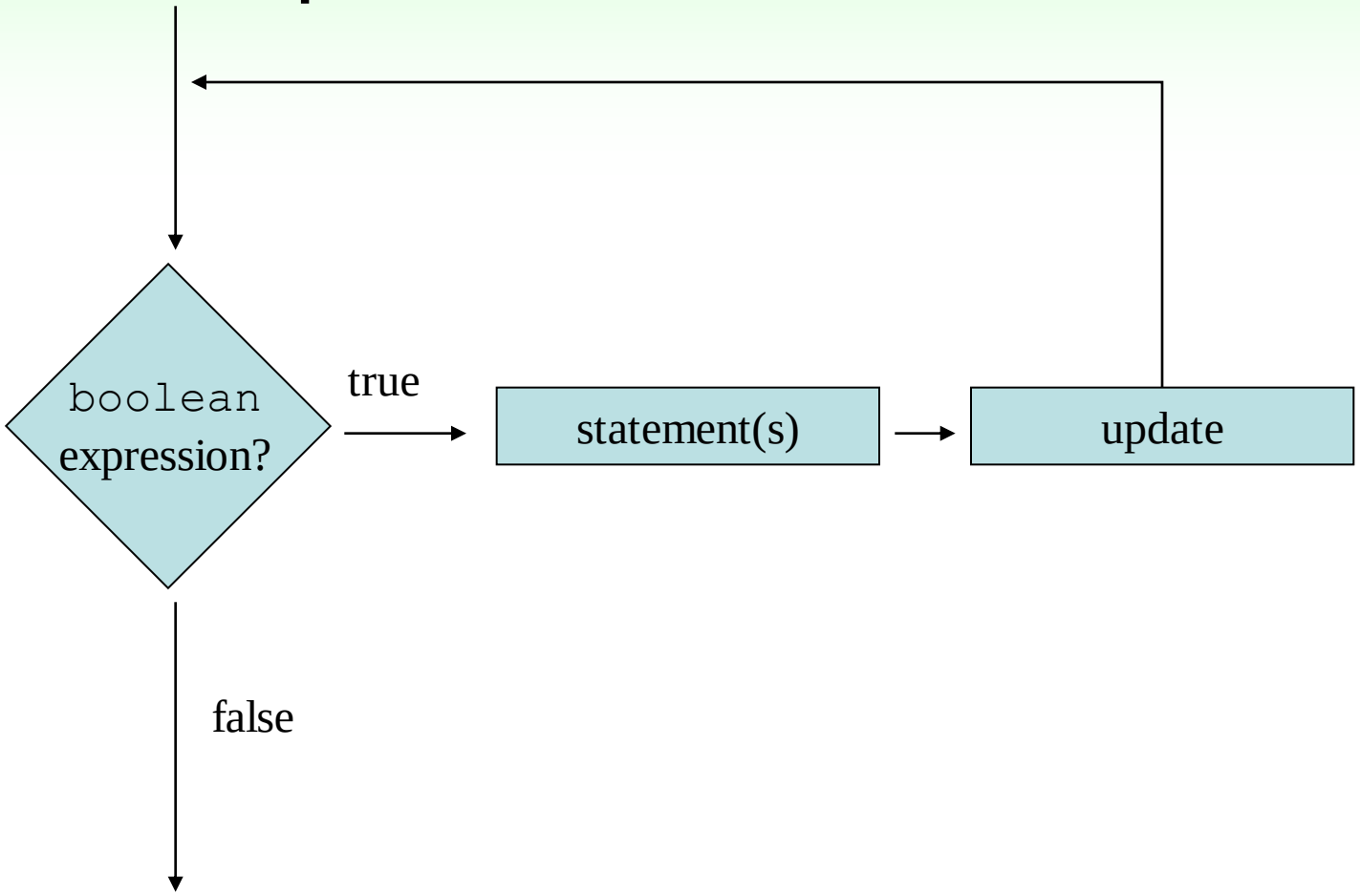
The `for` Loop

- 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.

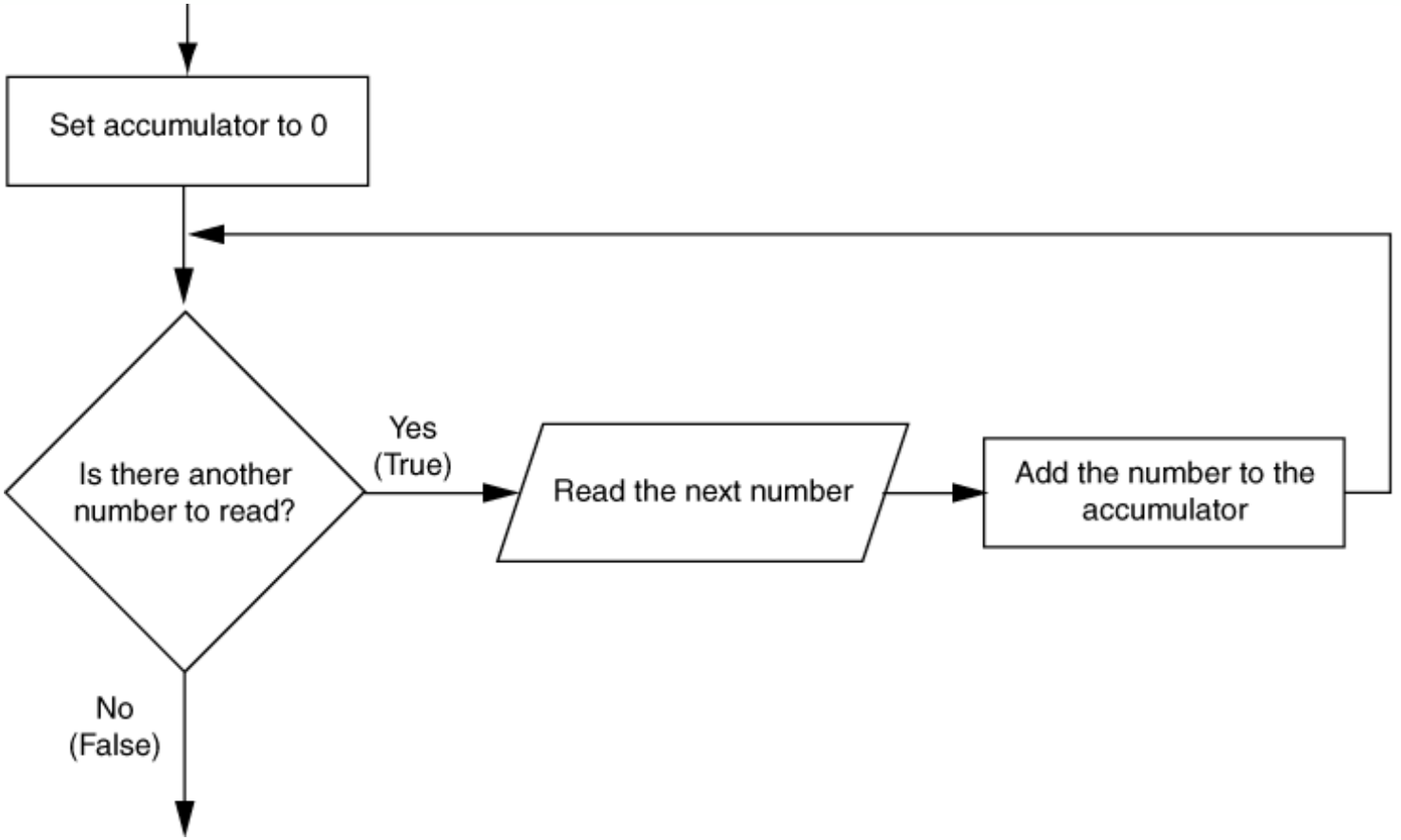
```
for(;;)
{
    statement(s);
} // infinite loop
```

- 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

Find the results of following statements

```
int a=18;
for(int i = 1; i <= a; i++)
        if(a%i == 0)
                System.out.print(i);
```

```
for(int i = 0; i < 5; i++)
{
        for(int j = 0; j <= i; j++)
                System.out.print(j + "\t");
        System.out.println();
}
```

```
int i = 10, s = 0;
while(i > 0)
{
        if(i%2 == 0)
                s+=i;
        else
                if(i > 5)
                        s+=2*i;
        i--;
}
System.out.print("s = " + s);
```

```
int a = 18, i = 1;
do
{
        if(a%i == 0)
                System.out.print (i + "\t");
        i++;
 } while(i <= a);
```

```
int a = 11, b = 16, i = a;
while( i < b )
{
    if(i%2 == 0)
    {
        System.out.print (i + "\t");
        break;
    }
    i++;
}
```

```
int a = 10, s = 0, i = 0;
while( i < a )
{
        i++;
        if (i % 2 == 0)
                continue;
        s=s+i;
}
System.out.print("s = " + s);
```

```
int i = 1, s = 0;
while(true)
{
    s = s + i++;
    if (i % 2==0)
        i = i + 2;
    else
        i = i + 1;
    if (i > 5)
        break;
 }
System.out.print("s = " + s);
```

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.

Example: Let the user continuously enters positive integer. Stop when the user enter negative number.

```
static void Main(string []args)
{
        int n;
        while (true)
        {
            System.out.print ("Input a number: ");
            n = reader.nextInt();
             if (n <= 0)
            {
                    System.out.print("End while loop");
                    break;
            }
        }
 }
```

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.

Example: Print number from 10 to 20 except 13 and 17.

```
static void Main(string []args)
{
        for (int k = 10; k <= 20; k++)
        {
            if (k != 13 && k != 17)
                System.out.print(k);
         }
}
```

Example: Print number from 10 to 20 except 13 and 17– Use *continue*

```
static void Main(string []args)
{
        for (int k = 10; k <= 20; k++)
        {
            if (k == 13 || k == 17)
                continue;
            System.out.print(k);
         }
}
```

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

- 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

The `PrintWriter` Class

- 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

- 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

- 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

- 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);
```

Specifying a File Location

- 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("A:\\PriceList.txt");
```

Specifying a File Location

- 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

- 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

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

- 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

- 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.
File file = new File("Names.txt");
Scanner inputFile = new Scanner(file);
// 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

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

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

- See example: FileReadDemo.java

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
`nextDouble()`	Returns the next random number as a `double`. The number will be within the range of 0.0 and 1.0.
`nextFloat()`	Returns the next random number as a `float`. The number will be within the range of 0.0 and 1.0.
`nextInt()`	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.
`nextInt(int n)`	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.

See example: RollDice.java

1. Given 3 positive integers. Check if those 3 numbers form a triangle? If so, what kind of triangle is it? (Isosceles triangle, right triangle, ...).

2. Write a program to input a positive integer n. Check if n is a perfect square number? (This is a number that when taken root of 2 has an integer result).

3. Write a program to count the divisors of a number.

Example: *N=12*

12 have 6 divisors.

4. A perfect number is a number whose sum of its divisors (excluding itself) is equal to itself. Enter the positive integer n and check if n is a perfect number

Example: the number 6 is a perfect number because the sum of the divisors is 1+2+3 = 6..

5. Calculate n!.

Example: *N=5* → *5! = 120.*

6. Input 2 positive integer number X and N. Calculate X^N.

Example: $2^3 = 8$.

7. Find the greatest common divisor of 2 positive integer numbers a and b.

8. Input a positive integer number N. Check if N is a prime number or not.

9. Input a positive integer number N. Print out prime numbers from 2 to N.

10. Input a positive integer number N. Print out the first N primes.

11. Print the Fibonacci. N entered by the user

- $f_1 = f_0 = 1;$
- $f_n = f_{n-1} + f_{n-2};$ $(n>1)$

12. Print a rectangle have size Width x Height

13. Print a right triangle have the height h.

14. Input a positive integer N:

- How many digits of N?
- What is the greatest digit?
- The odd digits is more than even digit or vice versa
- How many digit equal to the greatest digit?
- Is there any odd digit?
- Is N a symmetric number?
- Ex: 12321 → Yes

Q&A