Flow of Control: Loops

Reading: Appendix A

- Event-controlled loops.
- Looping techniques.
- `while` loops.
- Keyboard and file input using `Scanner`.
- `do while` loops.
- Count-controlled loops: `for` loops.
- Nested loops.

Flow of Control:
- Programs can control the order in which their instructions are executed.
  - A major feature of software and why it is useful!
- Four types of flow:
  - Sequential:
    - Execute instructions in the order listed in the code.
  - Method calls:
    - Transfer flow control to the code inside the method; i.e.
      ```java
      String zap = "Once upon a time in a galaxy far, far away...";
      String word = zap.substring(22, 28);
      ```
    - Control returns back to the point of the call. Some calls also return a value.
  - Selection: (just finished this)
    - Which set of instructions are executed depends on the data.
  - Looping: (now!)
    - Repeat a set of instructions, changing some of the data each time through the set of instructions.
Loop Basics:

- Grocery cashier:
  - Total cost starts at zero.
  - While there are more item(s) to scan:
    - Scan item and add cost to total
  - Print total.

- Exam grading:
  - While the grader next to you keeps handing you exams:
    - Turn to page 3 of the exam.
    - Grade questions 4 and 5.
  - Done — can go home!

- Drive-up window at Coffee Haunt:
  - While shift has not ended:
    - Take order from next car.
    - Fill haunted coffee cups.
    - Add tip to Tip Jar.
  - Done — count tips.

Basic idea:

- While the condition is true:
  - Get next item.
  - Perform one or more operations.

While Loop:

- The **while** loop is designed for repeating a set of operations on data items.
  - Especially when we do not know how many data items there will be.
- We will get **some signal** when we have reached the end of the items.
  - Divider bar for grocery clerk.
  - Grader next to us runs out of papers.
  - Our shift ends.
  - The user gets tired of entering numbers at the keyboard.
  - All the data in a file has been read.
  - Etc.
- This condition or value that marks the end of the input is called a **sentinel value**.
  - When reading from a file, the operating system will signal **end-of-file**.
- Receiving the signal is an **event**.
  - Thus, this technique is known as **event-controlled looping**.
While Loop (continued):
- **Initialize variable(s)** — set starting values of variables used in the body of the loop.

- While (condition is true)
  - True: Loop Body.
    - Contains instruction(s) to be performed each time.
  - False:
    - Loop is now done.
    - Usually do something with the results from the loop.
    - Then continue with the rest of the program.

While Loop (continued):
- Syntax for the `while` loop in Java:
  ```java
  // initialize variables here
  while ( boolean expression ) { 
    // process data (loop body)
    // can be as few or as many instructions as needed
  }
  // Loop has ended. Do something with the results.
  ```
  - The `{ }` are optional if only one statement is in the loop body.
  - They are required if two or more statements are in the loop body.

- The loop body is executed if the `boolean expression` is `true`.
- The loop ends when the `boolean expression` is `false`.
- Indent the body of the `while` loop to clearly illustrate to others (and yourself!) the logic of the program.
While Loop (continued):

Sentinel-controlled while Loop:

- The boolean expression in the while statement is evaluated each time.
- The loop executes as long as the boolean expression is true.
- Implies that something has to happen to make the boolean expression be false.
  - Otherwise, the loop runs forever!
    - (which makes it difficult to turn it in on-time! :-)
- The loop body must “update the loop condition”.
  - Some operation has to be performed that will (eventually) cause the boolean expression to be false.
  - Typically, the loop update will be an attempt to read the next input value, in order to detect the sentinel value.

```java
import java.util.Scanner;
public class OddsAndEvens1 {
    public static void main(String[] args) {
        Scanner inputScan = new Scanner(System.in);
        int number;
        int oddCount  = 0;
        int evenCount = 0;
        while ( number != -1 ) {
            System.out.print("Enter an integer, use -1 to stop: ");
            number = inputScan.nextInt();
            if ( number % 2 == 0 )
                evenCount++;
            else
                oddCount++;
        }
        System.out.println("Count of even numbers: " + evenCount);
        System.out.println("Count of odd numbers:  " + oddCount);
    }
} // main
} // OddsAndEvens1
```
While Loop (continued):

Sentinel-controlled while Loop (continued):

• "Priming the pump":
  • Need to get one value from the user before the loop starts.
    • Prompt user for a value and read that value before the while statement.
      
      System.out.print("Enter an integer, use -1 to stop: ");
      number = inputScan.nextInt();

      while ( number != -1 ) {

    • Inside the loop, the order is changed.
      • Process the data.
        if ( number % 2 == 0 )
          evenCount++;
        else
          oddCount++;
      • Then, read the next value.
        System.out.print("Enter an integer, use -1 to stop: ");
        number = inputScan.nextInt();

  }

import java.util.Scanner;
public class OddsAndEvens2{

  public static void main(String[] args)
  {
    Scanner inputScan = new Scanner( System.in );
    int number, oddCount = 0, evenCount = 0;

    System.out.print("Enter an integer, use -1 to stop: ");
    number = inputScan.nextInt();

    while ( number != -1 ) {
      if ( number % 2 == 0 )
        evenCount++;
      else
        oddCount++;

      System.out.print("Enter an integer, use -1 to stop: ");
      number = inputScan.nextInt();
    }

    System.out.println("Count of even numbers: "+ evenCount);
    System.out.println("Count of odd numbers: "+ oddCount);
  } // end of method main
} // end of class OddsAndEvens2
While Loop (continued):

Sentinel-controlled while Loop (continued):

• Some Definitions:
  
  • Iteration:
    • One execution of the loop body.
  
  • Loop Update:
    • One or more statements that could cause the loop condition to evaluate to false (to end the loop).
  
  • Loop Termination Condition:
    • The specific event that causes the loop condition to evaluate to false.

While Loop (continued):

Sentinel-controlled while Loop (continued):

• Common Problems:
  
  • Forgetting to “prime the pump”.

  • Putting a semi-colon after the boolean expression:
    
    ```
    while ( number != -1 ) {
    
    
    ```

  • Omitting the read of the next value.
While Loop (continued):

Reading from a Text File:

- When a large amount of data needs to be processed, it is better to have the data in a file.
  - Saves the user having to type it in each time.
  - Reduces errors by the user in entering the data.
- Basic idea
  
  ```java
  initialize variables
  while ( there is more data in the file ) {
    read the next data item from the file
    process the data
  }
  report the results
  ```

While Loop (continued):

Reading from a Text File (continued):

- The `Scanner` class can be used to read from a file.
  - However, `Scanner` cannot open the file :-(
- The `File` class can be used to open a text file given the name of the file.

- The `File` class:
  - Needs an import statement:
    ```java
    import java.io.File;
    ```
  - Has a constructor whose argument is a `String` that contains the name of the file.
    ```java
    File inputFile = new File( "someFileName.txt" );
    ```
  - This example assumes the file, `someFileName.txt`, is in the same folder as your Java program.
While Loop (continued):

Reading from a Text File (continued):

• Creating a `Scanner` object that is connected to the text file:
  
  • The `Scanner` class has a constructor whose argument is a `File` object.
    ```java
    Scanner fileScan = new Scanner( inputfile );
    ```
  
  • Now, we can read input from the text the same way we have been reading from the keyboard:
    ```java
    int number;
    number = fileScan.nextInt();
    ```

• Almost, but not quite there...

  
  While Loop (continued):

  Reading from a Text File (continued):

  • There are errors that can occur when getting input from a text file.
    
    • Java creates an `IOException` when an error occurs when reading the file.
    
    • Java requires us to acknowledge that these exceptions may be generated.
    
    • One way to handle this:
      
      • Add the phrase `throws IOException` to the definition of `main`:
        ```java
        public static void main( String[] args ) throws IOException
        ```
      
      • And, to make this work, you will need to import the `IOException` class:
        ```java
        import java.io.IOException;
        ```
While Loop (continued):

Reading from a Text File (continued):

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

public class OddsAndEvens3
{
    public static void main(String[] args) throws IOException
    {
        int number, oddCount = 0, evenCount = 0;

        File inputFile = new File("sample.txt");
        Scanner fileScan = new Scanner(inputFile);

        number = fileScan.nextInt();
        while ( number != -1 ) {
            System.out.println("number = " + number);
            if ( number % 2 == 0 )
                evenCount++;
            else
                oddCount++;
            number = fileScan.nextInt();
        }

        System.out.println("Count of even numbers: " + evenCount);
        System.out.println("Count of odd numbers: " + oddCount);
    } // end of method main
} // end of class OddsAndEvens3
```
While Loop (continued):

Reading from a Text File (continued):

- The program requires us to still put -1 at the end of the file as the sentinel.
  - It can sometimes be difficult to decide on a sentinel value.

- A better approach:
  - The `Scanner` class has a set of methods to help us in checking if we have reached the end of the input.
  - The `hasNext()` method returns a boolean.
    - `true` if there are more tokens on the input.
    - `false` if there are no additional tokens available.

- We can re-write the program:
  - No longer need to “prime the pump”.
  - Use `hasNext()` as the condition for the `while` loop.

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

public class OddsAndEvens4
{
    public static void main(String[] args) throws IOException
    {
        File inputFile = new File( "sample2.txt" );
        Scanner fileScan = new Scanner( inputFile );
        int number, oddCount = 0, evenCount = 0;
        while ( fileScan.hasNext() )
        {
            number = fileScan.nextInt();
            System.out.println("number = " + number);
            if ( number % 2 == 0 )
                evenCount++;
            else
                oddCount++;
        }
    }
}
```
While Loop (continued):

Reading from a Text File (continued):

```java
System.out.println("Count of even numbers: " + evenCount);
System.out.println("Count of odd numbers: " + oddCount);
} // end of method main
} // end of class OddsAndEvens4
```

While Loop (continued):

**Loopying Techniques:**

- Many common operations on a set of input values can be solved with a `while` loop.
- Here are four categories of problems for which `while` is a good solution:
  - Accumulation.
  - Counting Items.
  - Finding an Average.
  - Finding Maximum and/or Minimum Values.
While Loop (continued):

Looping Techniques (continued):

- Accumulation.
  - Generally used when there is a running total.
  - Start by initializing a variable to 0 to hold the sum.
  - Each iteration of the loop, read a value.
    - Add it to the total.

- This is the pattern used by the grocery cashier example.

- Odds and Evens: modify the program:
  - Find the sum of the even numbers and the sum of the odd numbers.
  - Count the number of evens and the number of odds (was in the original version).

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

public class SumOddsEvens {
    public static void main(String[] args) throws IOException {
        File inputFile = new File( "sample2.txt" );
        Scanner fileScan = new Scanner( inputFile );
        int number, oddCount = 0, evenCount = 0;
        int sumOdds = 0, sumEvens = 0;
        while ( fileScan.hasNext() ) {
            number = fileScan.nextInt();
            System.out.println("number = " + number);
            if ( number % 2 == 0 ) {
                evenCount++;
                sumEvens = sumEvens + number;
            } else {
                oddCount++;
                sumOdds = sumOdds + number;
            }
        }
    }
}
```
While Loop (continued):

Looping Techniques (continued):

```java
    System.out.println();
    System.out.println("Count of even numbers: " + evenCount);
    System.out.println("Sum of even numbers: " + sumEvens);
    System.out.println();
    System.out.println("Count of odd numbers:  " + oddCount);
    System.out.println("Sum of odd numbers:    " + sumOdds);

} // end of method main
} // end of class SumOddsEvens
```

While Loop (continued):

Looping Techniques (continued):

- Counting items.
  - Used when there is a running total.
  - Start by initializing a variable to 0 to hold the count.
  - Each iteration of the loop, read a value.
    - If the value is something we want to count:
      - Add one to the count.

- This is the pattern used by the various odds and evens examples.
While Loop (continued):

Looping Techniques (continued):

- Calculating an Average.
  - Combines the accumulation and counting techniques.
  - Start by initializing a variable to 0 to hold the count and a second variable to 0 to hold the sum.
  - Each iteration of the loop, read a value.
    - Add the value to the sum.
    - Add 1 to the count.
  - After the loop ends, compute the average.

- Example: Baseball batting average.
  - Input file contains String's: “walk”, “single”, “double”, “triple”, “home run”, “out”.
    - One String to a line.
  - Walks do not count as a “plate appearance” in computing the batting average.
  - Singles, doubles, triples and home runs all count as hits.
  - Divide the number of hits by the number of appearances.

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

public class BattingAverage
{
    public static void main(String[] args) throws IOException
    {
        int hits = 0, atBats = 0;
        double average = 0.0;
        File inputFile = new File( "baseball.txt" );
        Scanner fileScan = new Scanner( inputFile );
        String appearance;
        DecimalFormat averageFormat = new DecimalFormat( ".000" );
        Does average have to be initialized?
    }
```
While Loop (continued):

Looping Techniques (continued):

```java
while ( fileScan.hasNext() ) {
    appearance = fileScan.nextLine();
    if ( appearance.equals("out") ) {
        atBats++;
        if ( appearance.equals("walk") ) {
            hits++;
        }
        if ( appearance.equals("single") || appearance.equals("double") ||
            appearance.equals("triple") || appearance.equals("home run") ) {
            atBats++;
            hits++;
        }
    }
    System.out.println("at bats = " + atBats);
    System.out.println("hits = " + hits);
}
if ( atBats > 0 )
    average = (double) hits / atBats;
System.out.println("Batting average: " + averageFormat.format(average));
} // end of method main
```
Finding Maximum and/or Minimum Values.

- Approach: the running maximum and/or minimum.
- For the maximum:
  - Read the first item. Save it as the “current maximum”.
  - Each iteration:
    - Read a new value.
    - Compare it to the “current maximum”.
    - If the new value is larger, make it the “current maximum”.
  - When we have no more items to read, the “current maximum” is the maximum.

- Finding the minimum is similar.

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

public class MaxMinNames {
    public static void main(String[] args) throws IOException {
        File inputFile = new File( "names.txt" );
        Scanner fileScan = new Scanner( inputFile );
        String name;
        String currentMax, currentMin;
```
While Loop (continued):

```java
if ( ! fileScan.hasNext() )
    System.out.println("The input file has no Strings.");
else {
    name = fileScan.nextLine();
    currentMax = name;
    currentMin = name;
    System.out.println("currentMax is " + currentMax);
    System.out.println("currentMin is " + currentMin + \\
        "\n");
    while ( fileScan.hasNext() ) {
        name = fileScan.nextLine();
        System.out.println("name is " + name);
        if ( name.compareTo(currentMax) > 0 ) {
            currentMax = name;
            System.out.println("currentMax is " + currentMax);
        }
        if (name.compareTo(currentMin) < 0 ) {
            currentMin = name;
            System.out.println("currentMin is " + currentMin);
        }
        System.out.println();
    }
    System.out.println("The minimum name is '" + currentMin + \\
        "'");
    System.out.println("The maximum name is '" + currentMax + \\
        "\n");
}
```  

Why are these print’s here?

While Loop (continued):

Looping Techniques (continued):

- Finding Maximum and/or Minimum Values.
- Problems to watch out for:
  - Initializing a maximum or a minimum to an arbitrary value, such as 0 or 100, is a logic error.
    - Sometimes it works.
    - Most of the time, it leads to incorrect results that may be hard to debug.
- Examples:
  - An initial maximum of 0 when all the numbers entered are negative.
  - What is the effect of the following in the MaxMinNames program instead of reading the first line?
    ```java
    currentMax = new String( "" );
    currentMin = new String( "" );
    ```
While Loop (continued):

Input Problems:

- What happens if the user enters the wrong type of data?
  - For example, the user enters `twelve` rather than `12` when the program is expecting an `int`.
  - Using:
    ```java
    while ( fileScan.hasNext() ) {
    ```
  - Only tells us there is something on the input, not what the something is.
- The `Scanner` class contains a number of `has…()` methods that allow you to inquire as to specific types.
  - Each `hasNext…()` method will return `false` if there is no input available.
  - Each `hasNext…()` method will return `false` if the next token on the input is not the right type.
  - Except for `hasNext()`, which returns `true` as long as there is something remaining on the input.

<table>
<thead>
<tr>
<th>Return type</th>
<th>Method name and argument list</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>hasNextInt()</td>
</tr>
<tr>
<td>boolean</td>
<td>hasNextDouble()</td>
</tr>
<tr>
<td>boolean</td>
<td>hasNextFloat()</td>
</tr>
<tr>
<td>boolean</td>
<td>hasNextByte()</td>
</tr>
<tr>
<td>boolean</td>
<td>hasNextShort()</td>
</tr>
<tr>
<td>boolean</td>
<td>hasNextLong()</td>
</tr>
<tr>
<td>boolean</td>
<td>hasNextBoolean()</td>
</tr>
<tr>
<td>boolean</td>
<td>hasNext()</td>
</tr>
</tbody>
</table>

While Loop (continued):

Testing Techniques:

- Does the program produce correct results with a set of known input values?
  - You need to “know” what the correct answer is. May need to get the calculator out…
- Does the program produce correct results if the sentinel value is the first and only input?
  - What is the correct result if the sentinel is the first input?
  - For a text file, the question becomes: What should the program do if the input file is empty?
- Does the program deal appropriately with invalid input?
While Loop (continued):

Testing Techniques (continued):

• Does the program produce correct results with a set of known input values?
  • To verify the program’s results:
    • Select one or more sets of input values.
    • Calculate by hand the expected results, then compare with the program’s results.
  • Check boundary values.
    • Such as using the highest or lowest values possible on the input.
    • What if there is only one value on the input?
  • Check “edge” values of if statements.
    • Consider a program that reads grades and that grades are to be values in the range 0 to 100.
    • There should be one or more places in the program where the expression \((\text{grade} \leq 100)\) appears.
      • Use the values 99, 100, and 101 to test this condition.
    • And the expression \((\text{grade} \geq 0)\) would appear as well.
      • Use the values 1, 0, and -1 to test this condition.

While Loop (continued):

Testing Techniques (continued):

• Does the program produce correct results if the sentinel value is the first and only input?
  • Implies that the while loop will not be executed.
  • What should the program do when the sentinel is the first value? Does your program do this?
  • For a text file, the question becomes: “What happens if the text file is empty?”
• Does the program deal appropriately with invalid input?
  • Possible results:
    • An exception is generated: this will happen if \(\text{nextInt()}\) is called but a double is the next token.
    • The exception can be avoided in many cases by using the appropriate has…() method to test for the correct token.
      • But, what should the program do?
        • Just halt?
        • Print the results to this point, then halt?
        • Scold the user for being an idiot?
Do/While Loop:
- The condition for the `do/while` loop is evaluated at the end of the loop.
- The `do/while` loop executes the loop body at least once.
  - Compare to the `while` loop which might not execute the loop body at all.
- Some uses for a `do/while` loop:
  - Validate user input.
  - Ask if the user wants to repeat an operation.

Do/While Loop (continued):

`do/while` Syntax:
- The condition for the `do/while` loop is evaluated at the end of the loop.

```
// initialize variables
do {
  // body of the loop goes here
  // as many lines of code as needed
} while ( boolean expression );
// process the results
```

Unlike the `while` loop, the `{ }`'s are always required. The loop body can be empty. (But, the `{ }`'s are still required.)
Do/While Loop (continued):

do/while Example:

- Consider a guessing program that generates a random integer then accepts guesses until the user gets it right.

```java
import java.util.Scanner;
import java.util.Random;
public class GuessingInts
{
    public static void main(String[] args)
    {
        Random guess = new Random();
        Scanner inputScan = new Scanner(System.in);
        int userNumber, numGuesses = 0;
        int findIt = guess.nextInt(20) + 1;
        
        do {
            System.out.print("Guess a value between 1 and 20: ");
            userNumber = inputScan.nextInt();
            numGuesses++;
            if (userNumber > findIt)
                System.out.println("Too large
");
            if (userNumber < findIt)
                System.out.println("Too small
");
        } while (userNumber != findIt);
        System.out.println("Got it!");
        System.out.println("Took you "+numGuesses + " attempts.");
    } // end of method main
} // end of class GuessingInts
```

For Loop:

- Ideal when you know the number of iterations to perform before the loop begins.

- Examples:
  - Find the sum of 5 numbers. (or 50 numbers, or 5,000,000 numbers)
  - Find the maximum of 20 numbers.
  - Print the odd numbers from 1 to 10.
For Loop (continued):

**for Loop Syntax:**

- Syntax of the *for* loop:

  ```
  for ( initialization; loop condition; loop update ) {
      // loop body
      // goes here
  }
  ```

- Semi-colons separate terms in the loop header.
- There is not a semi-colon after the loop headers.
- The `{ }`'s are required only if more than one statement is in the loop body.

- The *for* loop flow of control:
  - The initialization statement is executed exactly once.
  - Repeat: The loop condition is evaluated. If the condition is true:
    - The loop body is executed.
    - The loop update is executed.
  - When the loop condition is evaluated as false:
    - Continue with code after the *for* loop.

**for Loop Examples:**

- Consider a *for* loop that prints the even numbers from 0 to 20:

  ```java
  int i = 0;
  for ( ; i <= 20; i = i + 2 ) {
      System.out.print(i + " ");
  }
  System.out.println();
  ```

- What needs to be changed to print the odd numbers from 0 to 20?

  ```java
  int i = 3;
  for ( ; ; ) {
      System.out.print(i + " ");
  }
  System.out.println();
  ```
For Loop (continued):

**for Loop Examples** (continued):

- The loop index can be any of the primitive data types: `char`, `short`, `int`, `long`, `float`, `double`.

- A `char` example:
  ```java
  char ch;
  for ( ch = 'A'; ch <= 'z'; ch++ ) {
      System.out.print( ch );
  }
  System.out.println();
  ```
  Output: `abcdefghijklmnopqrstuvwxyz`

- A `double` example:
  ```java
  double dd;
  for ( dd = 0.0; dd <= 5.0 * Math.PI; dd += Math.PI ) {
      System.out.println( dd );
  }
  System.out.println();
  ```
  Output:
  ```
  0.0
  3.141592653589793
  6.283185307179586
  9.42477796076938
  12.566370614359172
  15.707963267948966
  ```

Suppose we used `ch += 2` here?

- A `double` example:
  ```java
  double dd;
  for ( dd = 0.0; dd <= 5.0 * Math.PI; dd += Math.PI ) {
      System.out.println( dd );
  }
  System.out.println();
  ```
  Output:
  ```
  0.0
  3.141592653589793
  6.283185307179586
  9.42477796076938
  12.566370614359172
  15.707963267948966
  ```

**Variable Scope and for Loops:**

- The scope of an identifier is the block in which it is declared.
- Where block is a piece of code within `{ }`’s.
- The loop index can be declared before the `for` loop:
  ```java
  char ch;
  for ( ch = 'a'; ch <= 'z'; ch++ ) {
      System.out.print( ch );
  }
  System.out.println();
  ```
- The loop index can be declared in the `for` statement:
  ```java
  for ( char ch = 'Z'; ch > 'A'; ch-- ) {
      System.out.print( ch );
  }
  System.out.println();
  ```
For Loop (continued):

**Variable Scope and for Loops** (continued):

- But, if we want to use the loop index after the loop:
  - This works:
    ```java
    char ch;
    for ( ch = 'a'; ch <= 'z'; ch++ ) {
        System.out.print( ch );
    }
    System.out.println();
    System.out.println("The next character is " + ch);
    ```
  - This gives a compiler error: `cannot find symbol`
    ```java
    for ( char ch = 'A'; ch > 'A'; ch-- ) {
        System.out.print( ch );
    }
    System.out.println();
    System.out.println("The next character is " + ch);
    ```

- If you have two loops with the same identifier as the loop index:
  - Declare the loop index inside the `for` in both loops
    ```java
    for (char ch = 'a'; ch <= 'z'; ch++ ) {
        System.out.print( ch );
    }
    System.out.println();
    for (char ch = 'Z'; ch >= 'A'; ch-- ) {
        System.out.print( ch );
    }
    System.out.println();
    ```
  - Or, declare the loop index once before the first use:
    ```java
    char ch;
    for (ch = 'a'; ch <= 'z'; ch++ ) {
        System.out.print( ch );
    }
    System.out.println();
    for (ch = 'Z'; ch >= 'A'; ch-- ) {
        System.out.print( ch );
    }
    System.out.println();
    ```
For Loop (continued):

Testing for Loops:

- An important test for for loops is to have the starting and ending values set correctly.
- Example: To iterate a for loop 5 times
  - Use:
    ```java
    for (int i = 1; i <= 5; i++ )
    ```
  - Or, use:
    ```java
    for (int i = 0; i < 5; i++ )
    ```
- Consider a loop that counts the number of times the letter P appears in a String named word:
  ```java
  int countP = 0;
  for ( int i = 0; i < word.length(); i++ ) {
      if ( word.charAt(i) == 'P' )
          countP++;
  }
  ```
- Here are some wrong ways to write the loop header for the same problem. What is wrong with each?
  ```java
  for ( int i = 1; i < word.length(); i++ )
  for ( int i = word.length(); i >= 0; i-- )
  ```

For Loop (continued):

Testing for Loops (continued):

- How many times will the following loop execute?
  ```java
  String sample1 = new String( "Hello today!" );
  for ( int i = 0; i < sample1.length(); i++ ) {
      // loop body here
  }
  ```

- How many times will the following loop execute?
  ```java
  String sample2 = new String( "" );
  for ( int i = 0; i < sample2.length(); i++ ) {
      // loop body here
  }
  ```
Nested Loops:
- The body of a loop can be any valid Java code.
  - This includes loops!
- Can put any type of loop inside another loop:
  - `while` loop inside a `while` loop.
  - `do-while` loop inside a `while` loop.
  - `for` loop inside a `while` loop.
  - `while` loop inside a `for` loop.
  - Etc. (to a total of 9 possibilities).

Nested Loops (continued):
**do-while Inside a while Loop:**
- Consider a program that asks the user for the number of iterations:
  - The number of iterations should be a positive integer.
  - The user would enter 0 to stop.
- Want to check the input for invalid user responses.
  - In this case, “invalid” means a negative number.
- Since this is a Sentinel Loop:
  - Need to get (and check) user’s answer before the `while` loop starts, AND
  - Get (and check) user’s answer at the end of the `while` loop.
Nested Loops (continued):

**do-while Inside a while Loop** (continued):

```java
import java.util.Scanner;
public class Iterations {
    public static void main(String[] args) {
        Scanner inputScan = new Scanner(System.in);
        int i;
        int number;
        do {
            System.out.print("Enter number of iterations (0 to stop): ");
            number = inputScan.nextInt();
        } while (number < 0);

        // Print "Greetings" until user gets tired of it :-)
        while (number != 0) {
            for (i = 0; i < number; i++)
                System.out.println("Greetings!");
            System.out.println();
        }
    }
} // end of class Iterations
```

Nested Loops (continued):

**Nested for Loops**:

- How do for loops behave when nested? Here is a small example:

```java
import java.util.Scanner;

public class TwoNestedFor {
    public static void main(String[] args) {
        Scanner inputScan = new Scanner(System.in);
        int iterationsForI, iterationsForJ;
        System.out.print("Enter number of iterations for i: ");
        iterationsForI = inputScan.nextInt();
        System.out.print("Enter number of iterations for j: ");
        iterationsForJ = inputScan.nextInt();

        int i, j;
        for (i = 0; i < iterationsForI; i++)
            for (j = 0; j < iterationsForJ; j++)
                System.out.println("i = " + i + "    j = " + j);
    }
} // end of class TwoNestedFor
```
Nested Loops (continued):

Nested for Loops (continued):

• Same idea, but with three nested loops:

```java
import java.util.Scanner;

public class ThreeNestedFor {
    public static void main(String[] args) {
        Scanner inputScan = new Scanner(System.in);
        int iterationsForI, iterationsForJ, iterationsForK;
        System.out.print("Enter number of iterations for i: ");
        iterationsForI = inputScan.nextInt();
        System.out.print("Enter number of iterations for j: ");
        iterationsForJ = inputScan.nextInt();
        System.out.print("Enter number of iterations for k: ");
        iterationsForK = inputScan.nextInt();

        int i, j, k;
        for (i = 0; i < iterationsForI; i++)
            for (j = 0; j < iterationsForJ; j++)
                for (k = 0; k < iterationsForK; k++)
                    System.out.println("i = " + i + "    j = " + j + "    k = " + k);
    }
} // end of class ThreeNestedFor
```
Nested Loops (continued):

**Nested for Loops** (continued):

- Same idea, but with three nested loops (continued):
  - Result when:
    - outer loop executes twice (0, 1)
    - middle loop execute three times (0, 1, 2)
    - inner loop execute four times (0, 1, 2, 3)

```java
import java.util.Scanner;
public class Lots {
    public static void main(String[] args) {
        Scanner inputScan = new Scanner(System.in);
        int iterations;
        System.out.print("Enter number of iterations: ");
        iterations = inputScan.nextInt();

        int i, j, k;
        int sum = 0;

        for (i = 0; i < iterations; i++)
            for (j = 0; j < iterations; j++)
                for (k = 0; k < iterations; k++) {
                    sum = sum + k;
                }

        System.out.println("sum is "+sum);
    }
} // end of class Lots
```

Enter number of iterations for i: 2
Enter number of iterations for j: 3
Enter number of iterations for k: 4

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**Cautionary notes:**

- Loops inside loops can easily run up the execution time of a program.