Chapter 4 - Loops

Gaddis Chapter 4 slides

- Increment/Decrement operators
- Loops
  - While Loop
  - Do-While Loop
  - For Loop
    - Multi-var loops.
    - Nested For Loops
  - Break and continue
  - Deciding which Loop to Use

Increment and Decrement Operators

Four ways to write these operations:

```
number = number + 1 // is the same as:
number++ // or
++number

number = number - 1 // is the same as:
number-- // or
--number
```

Here's an example of how it's used:

```
int number = 5;
number = number + 1;
// number is 6
number++;
// number is 7

int newNumber = 9;
newNumber = newNumber - 1;
// newNumber is 8
newNumber--;
// newNumber is 7
```

For another example, see: `IncrementDecrement.java`.

Differences between prefix and postfix notation

Prefix (`++number`) and postfix (`number++`) increments and decrements operate differently when used in an expression. **Postfix is most common!**

> There is an extremely popular older language called `C++`. There is a reason it's not called `++C`!

When used in a statement (as in the code example above), you can write the increment/decrement either way (See: `prefix.java`).

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.

Prefix Example

```
int a = 5;
int b = ++a;
System.out.println("b is: " + b);
// 6
```

Use prefix notation:

- When the updated value is needed immediately.

Postfix Example

```
int a = 5;
int b = a++;
System.out.println("b is: " + b);
// 5
```

Use postfix notation:

- When the original value is needed first.
- In expressions where the original value is important.
- Postfix is almost always used in loops!

Loops

Java provides three different looping structures

- **While**
  - Use when you need to repeat a block of code as long as a condition is true.
  - Best used for input validation.
- **Do While**
  - Similar to the `while` loop, but the condition is evaluated after the loop has executed. Use a `do-while` loop when you need to ensure that the loop body is executed at least once, regardless of the condition.
  - This is useful for scenarios like displaying a menu to a user at least once and then repeating based on user input.
- **For**
  - Use when you know the exact number of iterations beforehand.
  - Best used for looping through an array!

while Loop

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

While the `condition` is true, the statements will execute repeatedly.

Example:

```
int i = 0;
while (i < 5) {
    System.out.println(i);
    i++;
}
```

The while loop is a **pretest loop**, which means that it will test the value of the condition prior to executing the loop.

Example: `WhileLoop.java`

- infinite loops

- While Loop for Input Validation (`SoccerTeams.java`)

do-while Loop

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

The do-while loop is a post-test loop, which means it will execute the loop prior to testing the condition.

Example:

```
int i = 0;
do {
    System.out.println(i);
    i++;
} while (i < 5);
```

Example: `TestAverage1.java`

for 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.

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

The for loop is a pre-test loop. Example: `Squares.java`.

Example:

```
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

- 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`

do-while vs. while

A `do-while` loop is used when you need the loop body to execute at least once, regardless of the condition.

- **User Input Validation**

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.

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

- **Menu-Driven Programs**
  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.

```
int choice;
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 = scanner.nextInt();

    switch (choice) {
        case 1:
            // Perform action for option 1
            break;
        case 2:
            // Perform action for option 2
            break;
```

```
            case 3:
                System.out.println("Exiting...");
                break;
            default:
                System.out.println("Invalid choice. Try again.");
        }
    } while (choice != 3);
```

- **Retry Mechanism**
  When you need to retry an operation until it succeeds, but you want to attempt it at least once.

```
boolean success;
do {
    success = attemptOperation();
    if (!success) {
        System.out.println("Operation failed. Retrying...");
    }
} while (!success);
```

- **Initial Setup or Configuration**
  When you need to perform an initial setup or configuration step that must be done at least once, and then repeat based on certain conditions.

```
boolean setupComplete;
do {
    setupComplete = performSetup();
    if (!setupComplete) {
        System.out.println("Setup incomplete. Please try again.");
    }
} while (!setupComplete);
```

These examples illustrate situations where the `do-while` loop ensures that the code inside the loop executes at least once, which is not guaranteed with a `while` loop.

More on for loops

Init multiple variables

I haven't seen this used often..

Initializing multiple variables in a Java `for` loop can be quite useful for several reasons:

1. **Efficiency**: It allows you to manage multiple variables within a single loop, reducing the need for additional loops or separate variable declarations.
2. **Readability**: Grouping related variables together in the loop initialization can make your code more readable and easier to understand.
3. **Synchronization**: When you need to update multiple variables in tandem, initializing them together ensures they are synchronized throughout the loop's execution.

Here's an example:

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

In this example, `i` and `j` are both initialized in the `for` loop. The loop continues to run as long as `i` is less than `j`, and both variables are updated in each iteration.

Nested for loops

Nested for loops are used quite often.

Nested `for` loops in Java are used when you need to perform operations that require multiple levels of iteration. Here are some common scenarios where nested `for` loops are useful:

1. **Multidimensional Arrays**: When working with 2D arrays (like matrices), nested loops are essential for accessing and manipulating elements.

```
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {
        System.out.print(matrix[i][j] + " ");
    }
    System.out.println();
}
```

2. **Complex Iterations**: When you need to compare elements in a collection or perform operations that require multiple passes over the data.

```
int[] array = {1, 2, 3, 4, 5};

for (int i = 0; i < array.length; i++) {
    for (int j = i + 1; j < array.length; j++) {
        System.out.println("Comparing " + array[i] + " and " + array[j]);
    }
}
```

3. **Generating Combinations**: When you need to generate all possible pairs or combinations of elements from a set.

```
char[] chars = {'A', 'B', 'C'};

for (int i = 0; i < chars.length; i++) {
    for (int j = 0; j < chars.length; j++) {
        System.out.println(chars[i] + " " + chars[j]);
    }
}
```

Here are a few more examples of nested `for` loops in Java to illustrate different use cases:

1. **Printing Patterns**: Nested loops are often used to print various patterns.

```
// Printing a right-angled triangle
int n = 5;
for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= i; j++) {
                System.out.print("* ");
        }
        System.out.println();
}
```

2. **Multiplication Table**: Generating a multiplication table.

```
int size = 10;
for (int i = 1; i <= size; i++) {
        for (int j = 1; j <= size; j++) {
                System.out.print(i * j + "\t");
        }
```

```
        System.out.println();
}
```

3. **Matrix Addition**: Adding two matrices.

```
int[][] matrixA = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
};
int[][] matrixB = {
        {9, 8, 7},
        {6, 5, 4},
        {3, 2, 1}
};
int[][] result = new int[3][3];

for (int i = 0; i < matrixA.length; i++) {
        for (int j = 0; j < matrixA[i].length; j++) {
                result[i][j] = matrixA[i][j] + matrixB[i][j];
        }
}

// Printing the result matrix
for (int i = 0; i < result.length; i++) {
        for (int j = 0; j < result[i].length; j++) {
                System.out.print(result[i][j] + " ");
        }
        System.out.println();
}
```

4. **Finding Pairs with a Given Sum**: Identifying pairs in an array that add up to a specific sum.

```
int[] numbers = {1, 2, 3, 4, 5, 6};
int targetSum = 7;

for (int i = 0; i < numbers.length; i++) {
        for (int j = i + 1; j < numbers.length; j++) {
                if (numbers[i] + numbers[j] == targetSum) {
                        System.out.println("Pair: (" + numbers[i] + ", " +
numbers[j] + ")");
                }
```

```
        }
}
```