**CHAPTER 5: LOOPS**

* 1. **INTRODUCTION**

A loop can be used to tell a program to execute statements repeatedly.

Java provides a powerful construct called a loop that controls how many times an operation or a sequence of operations is performed in succession. Using a loop statement, you can simply tell the computer to display a string a hundred times without having to code the print statement a hundred times.

For example:

int count = 0;

while (count < 100) {

System.out.println("Welcome to Java!");

count++;

}

The variable **count** is initially **0**. The loop checks whether count **< 100** is **true**. If so, it executes the loop body to display the message **Welcome to Java!** and increments **count** by **1**.

**Loops** are constructs that control repeated executions of a block of statements.

Java provides three types of loop statements:

* **while** loops,
* **do-while** loops
* **for** loops.
  1. **The while Loop**

A while loop executes statements repeatedly while the condition is true.

The syntax for the while loop is as follows:

while (loop-continuation-condition) {

Statement(s);

}

The **while** loop repeatedly executes the statements in the loop body when the **loop-continuation-condition** evaluates to **true**.

**NOTE: The loop-continuation-condition** must always appear inside the parentheses.

The braces enclosing the loop body can be omitted only if the loop body contains one or no statement.

* 1. **LOOP DESIGN STRATEGIES**

The key to designing a loop is to identify the code that needs to be repeated and write a condition for terminating the loop.

Writing a correct loop is not an easy task for novice programmers.

Consider three steps when writing a loop.

* Step 1: Identify the statements that need to be repeated.
* Step 2: Wrap these statements in a loop as follows:

while (true) {

Statements;

}

* Step 3: Code the **loop-continuation-condition** and add appropriate statements for controlling the loop.

while (loop-continuation-condition) {

Statements;

Additional statements for controlling the loop;

}

* 1. **CONTROLLING A LOOP WITH USER CONFIRMATION OR A SENTINEL VALUE**

It is a common practice to use a sentinel value to terminate the input.

The preceding example executes the loop five times.

If you want the user to decide whether to continue, you can offer a user **confirmation**.

The template of the program can be coded as follows:

char continueLoop = 'Y';

while (continueLoop == 'Y') {

// Execute the loop body once

...

// Prompt the user for confirmation

System.out.print("Enter Y to continue and N to quit: ");

continueLoop = input.getLine().charAt(0);

}

* 1. **THE do-while LOOP**

A **do-while** loop is the same as a **while** loop except that it executes the loop body first then checks the loop continuation condition.

The **do-while** loop is a variation of the **while** loop.

Its syntax is as follows:

do {

// Loop body; Statement(s);

} while (loop-continuation-condition);

* 1. **THE for LOOP**

A **for** loop has a concise syntax for writing loops.

Often you write a loop in the following common form:

i = initialValue; // Initialize loop control variable

while (i < endValue) {

// Loop body ...

i++; // Adjust loop control variable

}

This loop is intuitive and easy for beginners to grasp.

However, programmers often forget to adjust the control variable, which leads to an infinite loop.

A **for** loop can be used to avoid the potential error and simplify the preceding loop.

In general, the syntax for a **for** loop is:

for (initial-action; loop-continuation-condition; action-after-each-iteration) {

// Loop body;

Statement(s);

}

* 1. **WHICH LOOP TO USE**

You can use a **for** loop, a **while** loop, or a **do-while** loop, whichever is convenient.

The **while** loop and **do-while** loop are easier to learn than the **for** loop.

However, you will learn the **for** loop quickly after some practice.

A **for** loop places control variable initialization, loop continuation condition, and adjustment after each iteration all together.

It is more concise and enables you to write the code with less errors than the other two loops.

* 1. **NESTED LOOPS**

A loop can be nested inside another loop.

**Nested loops** consist of an outer loop and one or more inner loops.

Each time the outer loop is repeated, the inner loops are reentered, and started a new.

For example:

for (int i = 0; i < 10000; i++)

for (int j = 0; j < 10000; j++)

for (int k = 0; k < 10000; k++)

Perform an action

* 1. **MINIMIZING NUMERIC ERRORS**

Using floating-point numbers in the loop continuation condition may cause numeric errors.

Numeric errors involving floating-point numbers are inevitable, because floating-point numbers are represented in approximation in computers by nature.

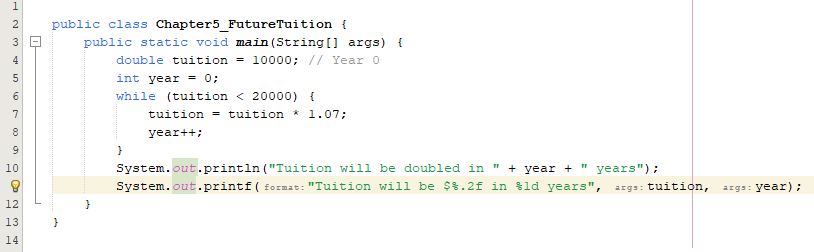
* 1. **CASE STUDIES**

Loops are fundamental in programming. The ability to write loops is essential in learning Java programming.

**Case Study: Predicting the Future Tuition**

Suppose the tuition for a university is $10,000 this year and tuition increases 7% every year. In how many years will the tuition be doubled?

Here is the code for FutureTuition.java

****

And the result is:



* 1. **KEYWORDS break AND continue**

The break and continue keywords provide additional controls in a loop.

You can use **break** in a loop to immediately terminate the loop.

You can use the **continue** keyword in a loop.

When it is encountered, it ends the current iteration and program control goes to the end of the loop body.

In other words, **continue** breaks out of an iteration, while the **break** keyword breaks out of a loop.

Example where break keyword:

int balance = 10;

while (true) {

if (balance < 9)

break;

balance = balance – 9;

}

System.out.println("Balance is " + balance);

Example where break keyword:

int balance = 10;

while (true) {

if (balance < 9)

continue;

balance = balance – 9;

}

System.out.println("Balance is " + balance);

**THE END!**