**JAVA Control Statements**

Decision Making in programming is similar to decision-making in real life. In programming also face some situations where we want a certain block of code to be executed when some condition is fulfilled.

A programming language uses control statements to control the flow of execution of a program based on certain conditions. These are used to cause the flow of execution to advance and branch based on changes to the state of a program.

**Java’s Selection statements:**

* if
* if-else
* nested-if
* if-else-if
* switch-case
* jump – break, continue, return

1. **if**: if statement is the simplest decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e., if a certain condition is true then a block of statement is executed otherwise not.

**Syntax**:

if(condition)

{

// Statements to execute if

// condition is true

}

Here, the **condition** after evaluation will be either true or false. if statement accepts boolean values – if the value is true then it will execute the block of statements under it.   
If we do not provide the curly braces ‘{‘ and ‘}’ after **if( condition )** then by default if statement will consider the immediate one statement to be inside its block. For example,

if(condition)

statement1;

statement2;

// Here if the condition is true, if block will consider only statement1

// to be inside its block.

1. **if-else**: The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won’t. But what if we want to do something else if the condition is false. Here comes the else statement. We can use the else statement with if statement to execute a block of code when the condition is false.

**Syntax**:

if (condition)

{

// Executes this block if

// condition is true

}

else

{

// Executes this block if

// condition is false

}

1. **nested-if:** A nested if is an if statement that is the target of another if or else. Nested if statements mean an if statement inside an if statement. Yes, java allows us to nest if statements within if statements. i.e., we can place an if statement inside another if statement.

**Syntax:**

if (condition1)

{

// Executes when condition1 is true

if (condition2)

{

// Executes when condition2 is true

}

}

1. **if-else-if ladder:** Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

if (condition)

statement;

else if (condition)

statement;

.

.

else

statement;

1. **switch-case:** The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

**Syntax:**

switch (expression)

{

case value1:

statement1;

break;

case value2:

statement2;

break;

.

.

case valueN:

statementN;

break;

default:

statementDefault;

}

* The expression can be of type byte, short, int char, or an enumeration. Beginning with JDK7, *expression* can also be of type String.
* Duplicate case values are not allowed.
* The default statement is optional.
* The break statement is used inside the switch to terminate a statement sequence.
* The break statement is optional. If omitted, execution will continue on into the next case.

You can use the **break** statement to end processing of a particular labeled statement within the **switch** statement. It branches to the end of the **switch** statement. Without **break**, the program continues to the next labeled statement, executing the statements until a **break** or the end of the statement is reached. This continuation may be desirable in some situations.

1. **jump:** Java supports three jump statements: **break, continue** and **return**. These three statements transfer control to another part of the program.
   * **Break:** In Java, a break is majorly used for:
     + Terminate a sequence in a switch statement (discussed above).
     + To exit a loop.
     + Used as a “civilized” form of goto.
   * **Continue:**Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop’s end. The continue statement performs such an action.
   * **Return**: The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

**Java Iteration Statement**

Java’s iteration statements are for, while, and do-while. These statements create what we commonly call loops. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. As you will see, Java has a loop to fit any programming need.

1. **while**:  A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.  
   **Syntax:**

while (boolean condition)

{

loop statements...

}

* While loop starts with the checking of condition. If it evaluated to true, then the loop body statements are executed otherwise first statement following the loop is executed. For this reason it is also called **Entry control loop**
* Once the condition is evaluated to true, the statements in the loop body are executed. Normally the statements contain an update value for the variable being processed for the next iteration.
* When the condition becomes false, the loop terminates which marks the end of its life cycle.

1. **for-loop:** for loop provides a concise way of writing the loop structure. Unlike a while loop, a for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping.

**Syntax:**

for (initialization condition; testing condition; increment/decrement)

{

statement(s)

}

* **Initialization condition:**Here, we initialize the variable in use. It marks the start of a for loop. An already declared variable can be used or a variable can be declared, local to loop only.
* **Testing Condition:** It is used for testing the exit condition for a loop. It must return a boolean value. It is also an **Entry Control Loop** as the condition is checked prior to the execution of the loop statements.
* **Statement execution:** Once the condition is evaluated to true, the statements in the loop body are executed.
* **Increment/ Decrement:**It is used for updating the variable for next iteration.
* **Loop termination:** When the condition becomes false, the loop terminates marking the end of its life cycle.

1. do while: do while loop is similar to while loop with only difference that it checks for condition after executing the statements, and therefore is an example of **Exit Control Loop.**

**Syntax:**

do

{

statements..

}

while (condition);

* do while loop starts with the execution of the statement(s). There is no checking of any condition for the first time.
* After the execution of the statements, and update of the variable value, the condition is checked for true or false value. If it is evaluated to true, next iteration of loop starts.
* When the condition becomes false, the loop terminates which marks the end of its life cycle.
* It is important to note that the do-while loop will execute its statements at least once before any condition is checked, and therefore is an example of exit control loop.

**for-each loop**

For-each is another array traversing technique like for loop, while loop, do-while loop introduced in Java5. 

* It starts with the keyword **for** like a normal for-loop.
* Instead of declaring and initializing a loop counter variable, you declare a variable that is the same type as the base type of the array, followed by a colon, which is then followed by the array name.
* In the loop body, you can use the loop variable you created rather than using an indexed array element.
* It’s commonly used to iterate over an array or a Collections class (eg, ArrayList)

**Syntax:**

for (type var : array)

{

statements using var;

}

//Java program to illustrate for-each loop

**class** For\_Each

{

**public** **static** **void** main(String[] arg)

{

{

**int**[] marks = { 125, 132, 95, 116, 110 };

**int** highest\_marks = *maximum*(marks);

System.***out***.println("The highest score is " + highest\_marks);

}

}

**public** **static** **int** maximum(**int**[] numbers)

{

**int** maxSoFar = numbers[0];

// for each loop

**for** (**int** num : numbers)

{

**if** (num > maxSoFar)

{

maxSoFar = num;

}

}

**return** maxSoFar;

}

}

**//** output: The highest score is 132

Limitations of for-each loop

* For-each loops are**not appropriate when you want to modify the array**.
* For-each loops **do not keep track of index**. So we can not obtain array index using For-Each loop.
* For-each **only iterates forward over the array in single steps, but with for loop we can iterate with reverse direction as well.**