

## Module 2

### Control Statements

- Java's program control statements can be put into the following categories: selection, iteration, and jump.
- *Selection* statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.
- *Iteration* statements enable program execution to repeat one or more statements (that is, iteration statements form loops).
- *Jump* statements allow your program to execute in a nonlinear fashion.

#### Java's Selection Statements

- Java supports two selection statements: **if** and **switch**.

#### The if statement

- The **if** statement executes a block of code only if the specified expression is true.
- If the value is false, then the **if** block is skipped and execution continues with the rest of the program.
- You can either have a single statement or a block of code within an **if** statement.
- Note that the conditional expression must be a Boolean expression.

#### **Syntax:**

```
if (<conditional expression>) {  
    <statements>  
}
```

#### **Example:**

```
public class Example {  
    public static void main(String[] args) {  
        int a = 10, b = 20;  
        if (a > b)  
            System.out.println("a > b");  
        if (a < b)
```

```
        System.out.println("b > a");
    }
}
```

### The if else statement

- The **if** statement is Java's conditional branch statement. It can be used to route program execution through two different paths.
- Here is the general form of the **if** statement:

**Syntax:**

```
if (condition)
    statement1;
else statement2;
```

- Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a *block*).
- The *condition* is any expression that returns a **boolean** value. The **else** clause is optional.
- The **if** works like this: If the *condition* is **true**, then *statement1* is executed. Otherwise, *statement2* (if it exists) is executed.

Example:

```
public class Example {
    public static void main(String[] args) {
        int a = 10, b = 20;
        if (a > b)
            System.out.println("a > b");
        else
            System.out.println("b > a");
    }
}
```

### Nested ifs

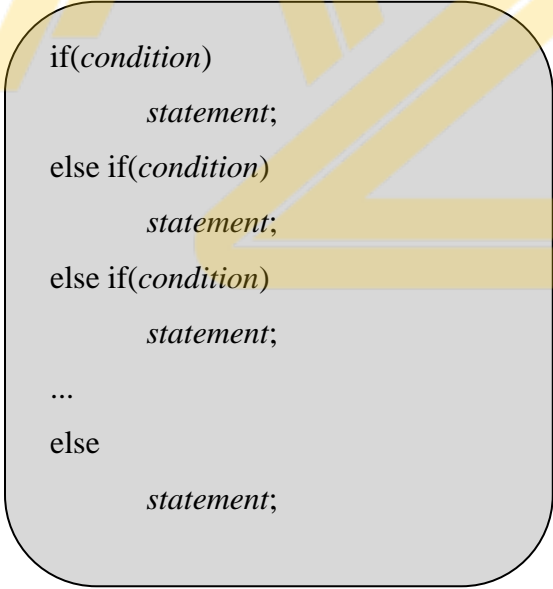
- A *nested if* is an **if** statement that is the target of another **if** or **else**.
- When you nest **ifs**, the main thing to remember is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**.

Here is an example:

```
if(i == 10) {  
    if(j < 20) a = b;  
    if(k > 100) c = d; // this if is  
                      else a = c; // associated with this else  
}  
else a = d; // this else refers to if(i == 10)
```

### The if-else-if Ladder

- A common programming construct that is based upon a sequence of nested **ifs** is the *if-else-if ladder*.
- It looks like this:



```
if(condition)  
    statement;  
else if(condition)  
    statement;  
else if(condition)  
    statement;  
...  
else  
    statement;
```

The diagram shows a rounded rectangular box containing the code for an if-else-if ladder. The code is formatted with indentation for the statements. A large, faint yellow 'A' watermark is visible in the background of the slide.

- 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.

Example:

```
class IfElse {  
    public static void main(String args[]) {  
        int month = 4; // April  
        String season;  
        if(month == 12 || month == 1 || month == 2)  
            season = "Winter";  
        else if(month == 3 || month == 4 || month == 5)  
            season = "Spring";  
        else if(month == 6 || month == 7 || month == 8)  
            season = "Summer";  
        else if(month == 9 || month == 10 || month == 11)  
            season = "Autumn";  
        else  
            season = "Bogus Month";  
        System.out.println("April is in the " + season + ".");  
    }  
}
```

### **The switch statement**

- The **switch case** statement is a multi-way branch with several choices. A switch is easier to implement than a series of if/else statements.

### **Structure of Switch:**

- The switch statement begins with a keyword, followed by an expression that equates to a no long integral value.
- Following the controlling expression is a code block that contains zero or more labeled cases.
- Each label must equate to an integer constant and each must be unique.

### **Working of switch case:**

- When the switch statement executes, it compares the value of the controlling expression to the values of each case label.

- The program will select the value of the case label that equals the value of the controlling expression and branch down that path to the end of the code block.
- If none of the case label values match, then none of the codes within the switch statement code block will be executed. Java includes a **default** label to use in cases where there are no matches.
- We can have a nested switch within a case block of an outer switch.

**Syntax:**

```
switch (<non-long integral expression>) {  
    case label1: <statement1> ; break;  
    case label2: <statement2> ; break;  
    ...  
    case labeln: <statementn> ; break;  
    default: <statement>  
}
```

**Example:**

```
public class Example {  
    public static void main(String[] args) {  
        int a = 10, b = 20, c = 30;  
        int status = -1;  
        if (a > b && a > c) {  
            status = 1;  
        } else if (b > c) {  
            status = 2;  
        } else {  
            status = 3;  
        }  
        switch (status) {  
            case 1:  
                System.out.println("a is the greatest");  
                break;  
            case 2:
```

```
        System.out.println("b is the greatest");
        break;
    case 3:
        System.out.println("c is the greatest");
        break;
    default:
        System.out.println("Cannot be determined");
    }
}
}
```

- The **break** statement is optional. If you omit the **break**, execution will continue on into the next **case**.
- It is sometimes desirable to have multiple **cases** without **break** statements between them.
- For example, consider the following program:

// In a switch, break statements are optional.

```
class MissingBreak {
    public static void main(String args[]) {
        for(int i=0; i<12; i++)
            switch(i) {
                case 0:
                case 1:
                case 2:
                case 3:
                case 4:
                    System.out.println("i is less than 5");
                    break;
                case 5:
                case 6:
                case 7:
                case 8:
                case 9:
```

```
        System.out.println("i is less than 10");
        break;
    default:
        System.out.println("i is 10 or more");
    }
}
}
```

### Nested switch Statements

- You can use a **switch** as part of the statement sequence of an outer **switch**. This is called a *nested switch*.
- Since a **switch** statement defines its own block, no conflicts arise between the **case** constants in the inner **switch** and those in the outer **switch**.
- For example, the following fragment is perfectly valid:

```
switch(count) {
    case 1:
        switch(target) { // nested switch
            case 0:
                System.out.println("target is zero");
                break;
            case 1: // no conflicts with outer switch
                System.out.println("target is one");
                break;
        }
        break;
    case 2: // ...
}
```

In summary, there are three important features of the **switch** statement to note:

- The **switch** differs from the **if** in that **switch** can only test for equality, whereas **if** can evaluate any type of Boolean expression. That is, the **switch** looks only for a match between the value of the expression and one of its **case** constants.
- No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement and an enclosing outer **switch** can have **case** constants in common.

- A **switch** statement is usually more efficient than a set of nested **ifs**.

## Iteration Statements

### The while loop

- The **while** statement is a looping construct control statement that executes a block of code while a condition is true.
- You can either have a single statement or a block of code within the while loop.
- The loop will never be executed if the testing expression evaluates to false.
- The loop condition must be a **boolean** expression.

#### **Syntax:**

```
while (<loop condition>) {  
    <statements>  
}
```

#### **Example:**

```
public class Example {  
    public static void main(String[] args) {  
        int count = 1;  
        System.out.println("Printing Numbers from 1 to 10");  
        while (count <= 10) {  
            System.out.println(count++);  
        }  
    }  
}
```

### The do-while loop

- The **do-while** loop is similar to the **while** loop, except that the test is performed at the end of the loop instead of at the beginning.
- This ensures that the loop will be executed at least once.



**Syntax:**

```
do {  
    <loop body>  
} while (<loop condition>);
```

**Example:**

```
public class Example {  
    public static void main(String[] args) {  
        int count = 1;  
        System.out.println("Printing Numbers from 1 to 10");  
        do {  
            System.out.println(count++);  
        } while (count <= 10);  
    }  
}
```

**The for loop**

- The **for** loop is a looping construct which can execute a set of instructions a specified number of times. It's a counter controlled loop.

**Syntax:**

```
for (<initialization>; <loop condition>; <increment expression>) {  
    <loop body>  
}
```

**Example:**

```
public class Example {  
    public static void main(String[] args) {  
        System.out.println("Printing Numbers from 1 to 10");  
        for (int count = 1; count <= 10; count++) {  
            System.out.println(count);  
        }  
    }  
}
```

### Declaring Loop Control Variables Inside the for Loop

- Often the variable that controls a **for** loop is only needed for the purposes of the loop and is not used elsewhere.
- When this is the case, it is possible to declare the variable inside the initialization portion of the **for**.

```
class ForTick {  
    public static void main(String args[]) {  
        // here, n is declared inside of the for loop  
        for(int n=10; n>0; n--)  
            System.out.println("tick " + n);  
    }  
}
```

- When you declare a variable inside a **for** loop, there is one important point to remember: the scope of that variable ends when the **for** statement does

### Using the Comma

- There will be times when you will want to include more than one statement in the initialization and iteration portions of the **for** loop.

```
class Comma {  
    public static void main(String args[]) {  
        int a, b;  
        for(a=1, b=4; a<b; a++, b--) {  
            System.out.println("a = " + a);  
            System.out.println("b = " + b);  
        }  
    }  
}
```

### Some for Loop Variations

- The **for** loop supports a number of variations that increase its power and applicability. The reason it is so flexible is that its three parts—the initialization, the conditional test, and the iteration—do not need to be used for only those purposes can be used for any purpose you desire.

- One of the most common variations involves the conditional expression.
- Specifically, this expression does not need to test the loop control variable against some target value. In fact, the condition controlling the **for** can be any Boolean expression. For example, consider the following fragment:

```
boolean done = false;
for(int i=1; !done; i++) {
    // ...
    if(interrupted()) done = true;
}
```

In this example, the **for** loop continues to run until the **boolean** variable **done** is set to **true**. It does not test the value of **i**.

- Here is another interesting **for** loop variation. Either the initialization or the iteration expression or both may be absent, as in this next program:

// Parts of the for loop can be empty.

```
class ForVar {
    public static void main(String args[]) {
        int i;
        boolean done = false;
        i = 0;
        for( ; !done; ) {
            System.out.println("i is " + i);
            if(i == 10) done = true;
            i++;
        }
    }
}
```

Here, the initialization and iteration expressions have been moved out of the **for**. Thus, parts of the **for** are empty

- Here is one more **for** loop variation. You can intentionally create an infinite loop (a loop that never terminates) if you leave all three parts of the **for** empty.
- For example:

```
for(;;) {  
    // ...  
}
```

This loop will run forever because there is no condition under which it will terminate.

### The For-Each Version of the for Loop

- Beginning with JDK 5, a second form of **for** was defined that implements a “for-each” style loop.
- The general form of the for-each version of the **for** is shown here:

```
for(type itr-var : collection)  
    statement-block
```

- Here, *type* specifies the type and *itr-var* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end.
- The collection being cycled through is specified by *collection*.
- There are various types of collections that can be used with the **for**, but the only type used in this chapter is the array.

#### **Working:**

- With each iteration of the loop, the next element in the collection is retrieved and stored in *itr-var*.
- The loop repeats until all elements in the collection have been obtained.
- Because the iteration variable receives values from the collection, *type* must be the same as (or compatible with) the elements stored in the collection.
- Thus, when iterating over arrays, *type* must be compatible with the base type of the array.

```
class ForEach {  
    public static void main(String args[]) {  
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
        int sum = 0;  
        for(int x : nums) {  
            sum += x;  
        }  
    }  
}
```

```
        System.out.println("Summation: " + sum);
    }
}
```

- With each pass through the loop, **x** is automatically given a value equal to the next element in **nums**. Thus, on the first iteration, **x** contains 1; on the second iteration, **x** contains 2; and so on.
- Not only is the syntax streamlined, but it also prevents boundary errors.

For example, this program sums only the first five elements of **nums**:

```
class ForEach2 {
    public static void main(String args[]) {
        int sum = 0;
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        // use for to display and sum the values
        for(int x : nums) {
            sum += x;
            if(x == 5) break; // stop the loop when 5 is obtained
        }
        System.out.println("Summation of first 5 elements: " + sum);
    }
}
```

### Iterating Over Multidimensional Arrays

- The enhanced version of the **for** also works on multidimensional arrays.
- Remember, however, that in Java, multidimensional arrays consist of *arrays of arrays*. (For example, a two-dimensional array is an array of one-dimensional arrays.)

```
class ForEach3 {
    public static void main(String args[]) {
        int sum = 0;
        int nums[][] = new int[3][5];
        // give nums some values
        for(int i = 0; i < 3; i++)
            for(int j=0; j < 5; j++)
```

```
        nums[i][j] = (i+1)*(j+1);  
    // use for-each for to display and sum the values  
    for(int x[] : nums) {  
        for(int y : x) {  
            sum += y;  
        }  
    }  
    System.out.println("Summation: " + sum);  
}  
}
```

- In the program, pay special attention to this line:

```
    for(int x[] : nums) {
```

- Notice how **x** is declared. It is a reference to a one-dimensional array of integers.
- This is necessary because each iteration of the **for** obtains the next *array* in **nums**, beginning with the array specified by **nums[0]**.
- The inner **for** loop then cycles through each of these arrays, displaying the values of each element.

### *Java program to search given key element*

```
class Search {  
    public static void main(String args[]) {  
        int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };  
        int val = 5;  
        boolean found = false;  
        // use for-each style for to search nums for val  
        for(int x : nums) {  
            if(x == val) {  
                found = true;  
                break;  
            }  
        }  
        if(found)  
            System.out.println("Value found!");  
    }  
}
```

```
}  
}
```

### Nested Loops

- Like all other programming languages, Java allows loops to be nested.
- That is, one loop may be inside another. For example, here is a program that nests **for** loops:

```
class Nested {  
    public static void main(String args[]) {  
        int i, j;  
        for(i=0; i<10; i++) {  
            for(j=i; j<10; j++)  
                System.out.print(".");  
            System.out.println();  
        }  
    }  
}
```

### Jump Statements

- Java supports three jump statements: **break**, **continue**, and **return**. These statements transfer control to another part of your program.

#### The break statement

- The **break** statement transfers control out of the enclosing loop (for, while, do or switch statement).
- You use a **break** statement when you want to jump immediately to the statement following the enclosing control structure.
- You can also provide a loop with a label, and then use the label in your **break** statement.
- The label name is optional, and is usually only used when you wish to terminate the outermost loop in a series of nested loops.

#### **Syntax:**

```
break; // the unlabeled form  
break <label>; // the labeled form
```

**Example for break:**

```
public class Example {  
    public static void main(String[] args) {  
        System.out.println("Numbers 1 - 10");  
        for (int i = 1; ++i) {  
            if (i == 11)  
                break;  
            System.out.println(i + "\t");  
        }  
    }  
}
```

**Example for labeled break:**

```
class Break {  
    public static void main(String args[]) {  
        boolean t = true;  
        first: {  
            second: {  
                third: {  
                    System.out.println("Before the break.");  
                    if(t) break second; // break out of second block  
                    System.out.println("This won't execute");  
                }  
                System.out.println("This won't execute");  
            }  
            System.out.println("This is after second block.");  
        }  
    }  
}
```

Running this program generates the following output:

Before the break.

This is after second block.



### The continue statement

- A **continue** statement stops the iteration of a loop (while, do or for) and causes execution to resume at the top of the nearest enclosing loop.
- You use a **continue** statement when you do not want to execute the remaining statements in the loop, but you do not want to exit the loop itself.
- You can also provide a loop with a label and then use the label in your **continue** statement.
- The label name is optional, and is usually only used when you wish to return to the outermost loop in a series of nested loops.

**Syntax:**

```
continue; // the unlabeled form
continue <label>; // the labeled form
```

**Example for continue:**

```
public class Example {
    public static void main(String[] args) {
        System.out.println("Odd Numbers");
        for (int i = 1; i <= 10; ++i) {
            if (i % 2 == 0)
                continue;
            System.out.println(i + "\t");
        }
    }
}
```

**Example for labelled continue:**

```
class ContinueLabel {
    public static void main(String args[]) {
        outer: for (int i=0; i<10; i++) {
            for(int j=0; j<10; j++) {
                if(j > i) {
                    System.out.println();
                    continue outer;
                }
            }
        }
    }
}
```

```
        System.out.print(" " + (i * j));  
    }  
}  
System.out.println();  
}  
}
```

### The return statement

- The **return** statement exits from the current method, and control flow returns to where the method was invoked.

#### **Syntax:**

The **return** statement has two forms:

One that returns a value

**return val;**

One that doesn't return a value

**return;**

#### **Example:**

```
public class Example {  
    public static void main(String[] args) {  
        int res = sum(10, 20);  
        System.out.println(res);  
    }  
    private static int sum(int a, int b) {  
        return (a + b);  
    }  
}
```