Java Fundamentals

Lesson 5: Flow Control

Speaker: Nicolae Sîrbu
Alexandru Umaneț

Lesson Objectives

- Understand the `if`, `if-else`, and ternary constructs
- Work with `switch` statements
- Learn how to use the `for` loop and the enhanced `for` loop
- Execute some actions in a `while` and `do-while` loops
- Understand the difference between loop constructs
- Learn how to exit a loop with `break` and hot to skip an iteration with `continue`

The `if` and `if-else` constructs

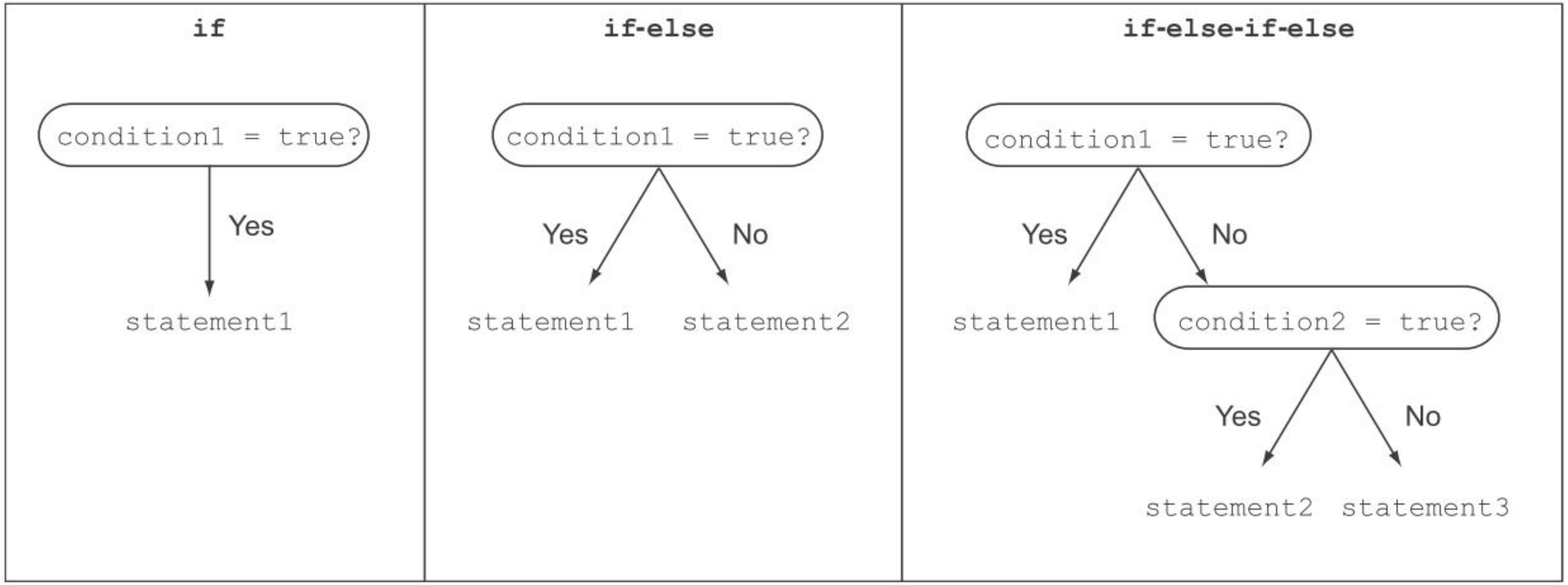
The `if` construct

An `if` construct enables you to execute a set of statements in your code based on the result of a condition. This condition must always evaluate to a `boolean` or a `Boolean` value. You can specify a set of statements to execute when this condition evaluates to `true` or `false`.

There are multiple flavors of the `if` statement:

- `if`
- `if-else`
- `if-else-if-else`

Multiple flavors of the `if` statement



Example of constructs

if

```
if (name.equals("Lion"))
    score = 200;
```

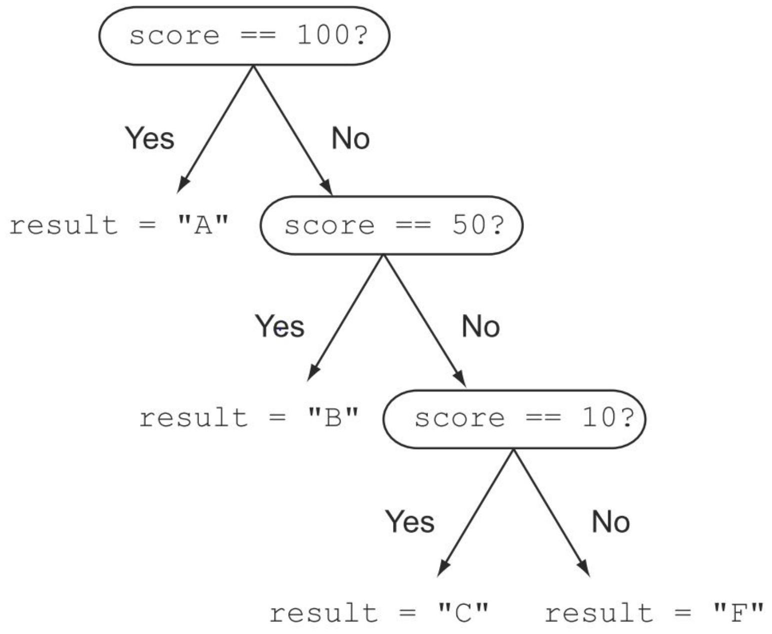
if-else

```
if (name.equals("Lion"))
    score = 200;
else
    score = 300;
```

if-else-if-else

```
if (score == 100)
    result = "A";
else if (score == 50)
    result = "B";
else if (score == 10)
    result = "C";
else
    result = "F";
```

Execution of a `if-else-if-else` flavour



Execution of a `if-else-if-else` flavour

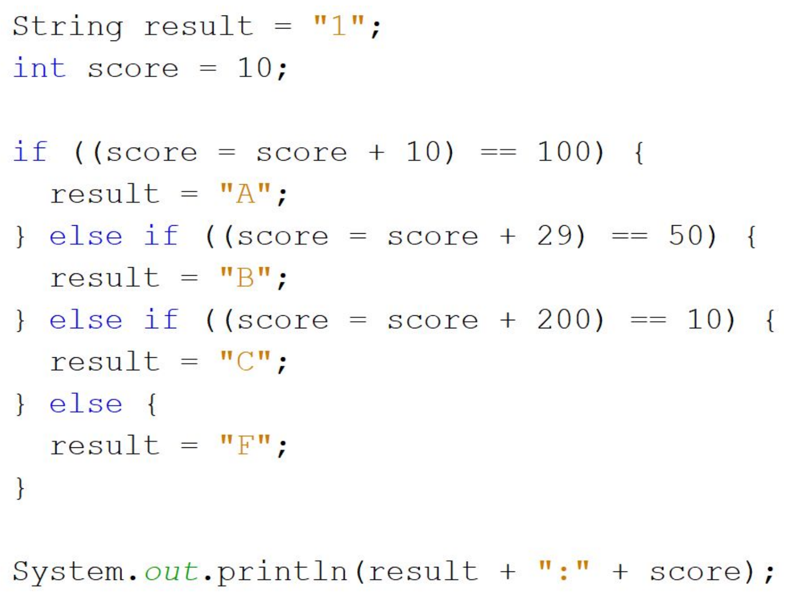
What will be the output of this block of code?

```
String result = "1";
int score = 10;

if (score == 100) {
  result = "A";
} else if (score == 50) {
  result = "B";
} else if (score == 10) {
  result = "C";
} else {
  result = "F";
}

System.out.println(result + ":" + score);
```

Output: `C:10`

Execution of a `if-else-if-else` flavour

What will be the output of this block of code?



```java
String result = "1";
int score = 10;

if ((score = score + 10) == 100) {
  result = "A";
} else if ((score = score + 29) == 50) {
  result = "B";
} else if ((score = score + 200) == 10) {
  result = "C";
} else {
  result = "F";
}

System.out.println(result + ":" + score);
```

Output: `F:249`

Missing `else` blocks

It’s acceptable to define one course of action for an `if` construct without defining the `else` statement as follows:

```
boolean testValue = false;

if (testValue == true)
    System.out.println("value is true");
```

Missing `else` blocks

You can’t define the `else` part for an `if` construct, skipping the `if` code block. The following code won’t compile:

```
boolean testValue = false;

if (testValue == true)
else
     System.out.println("value is false");
```

Presence and absence of `{ }`

You can execute a single statement or a block of statements when an `if` condition evaluates to `true` or `false`.

An `if` block is marked by enclosing one or more statements within a pair of curly braces `{ }`.

An `if` block will execute a single line of code if there are no braces, but will execute an unlimited number of lines if they're contained within a block (defined using braces).

The braces are optional if there's only one line in the `if` statement.

Example of using braces

```
String name = "Lion";
int score = 100;

if (name.equals("Lion")) {
     score = 200;
     name = "Larry";
} else {
     score = 129;
}
```

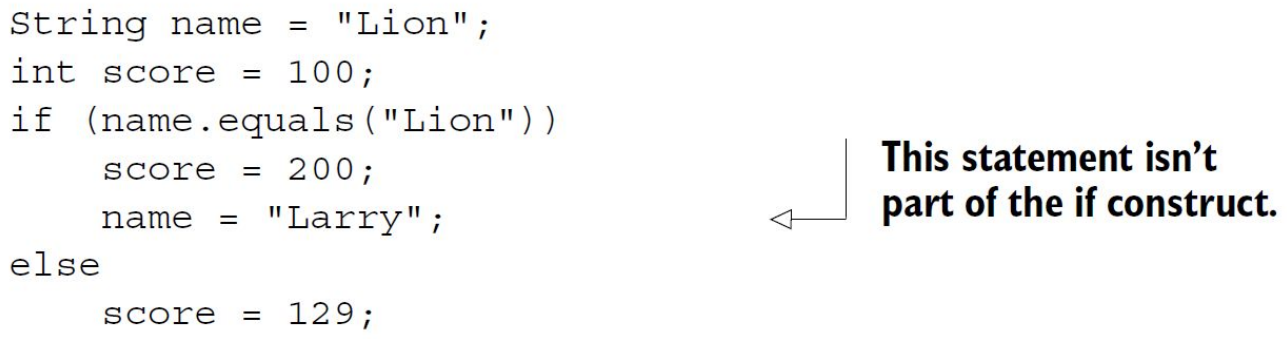
What is the output?

```
String name = "Lion";


if (name.equals("Lion"))
     System.out.println("Lion");
else
     System.out.println("Not a Lion");
     System.out.println("Again, not a Lion");
```

Output:
```
Lion
Again, not a Lion
```

Example of using braces

What happens to the code if you define an `else` part for your `if` construct, as follows?



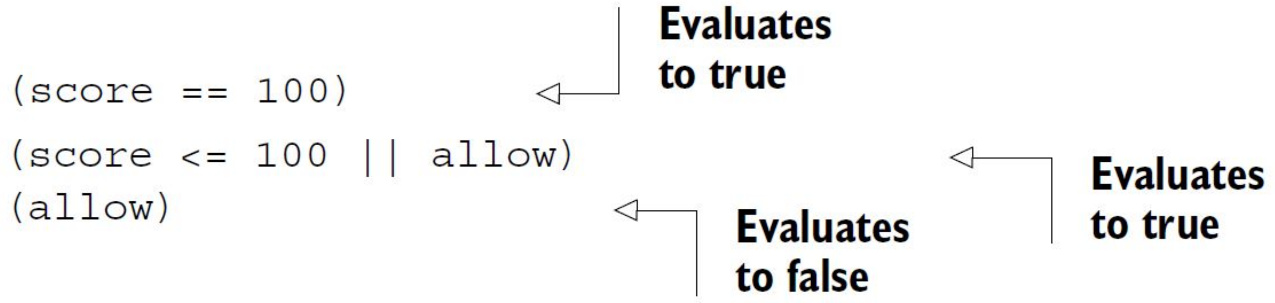
```
String name = "Lion";
int score = 100;
if (name.equals("Lion"))
    score = 200;
    name = "Larry";
else
    score = 129;
```

This statement isn’t part of the if construct.

Expressions passed as arguments to an `if` statement

The result of a variable or an expression used in an `if` construct must evaluate to `true` or `false`. Assume the following definitions of variables:

```
int score = 100;
boolean allow = false;
```



```
(score == 100)
(score <= 100 || allow)
(allow)
```

What is the output?

```
boolean allow = false;


if (allow = true)
     System.out.println("value is true");
else
     System.out.println("value is false");
```

Output: `value is true`

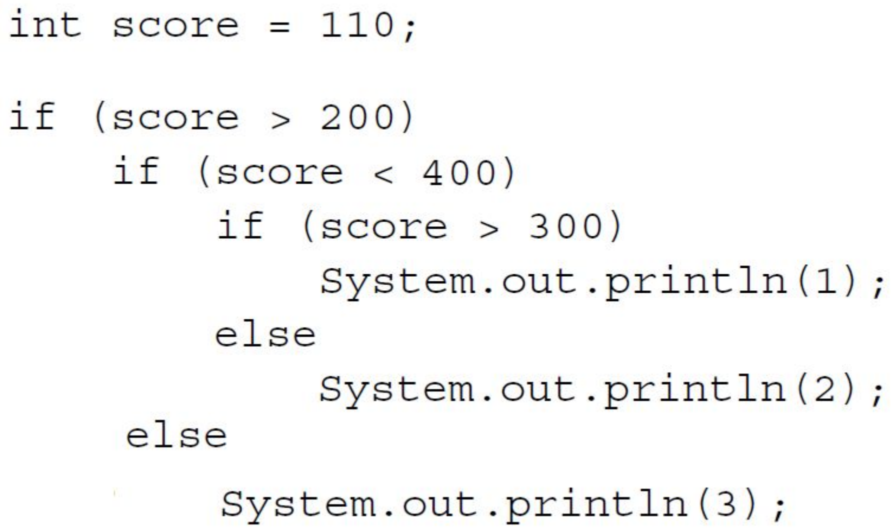
What is the output?

```
boolean allow = false;


if (allow == true)
     System.out.println("value is true");
else
     System.out.println("value is false");
```

Output: `value is false`

Nested `if` constructs

A nested `if` construct is an `if` construct defined within another `if` construct. Theoretically, there's no limit on the number of levels of nested `if` and `if-else` constructs.



```
int score = 110;

if (score > 200)
    if (score < 400)
        if (score > 300)
            System.out.println(1);
        else
            System.out.println(2);
    else
        System.out.println(3);
```

Defining an `else` for an outer `if`

The key point is to use curly braces, as follows:

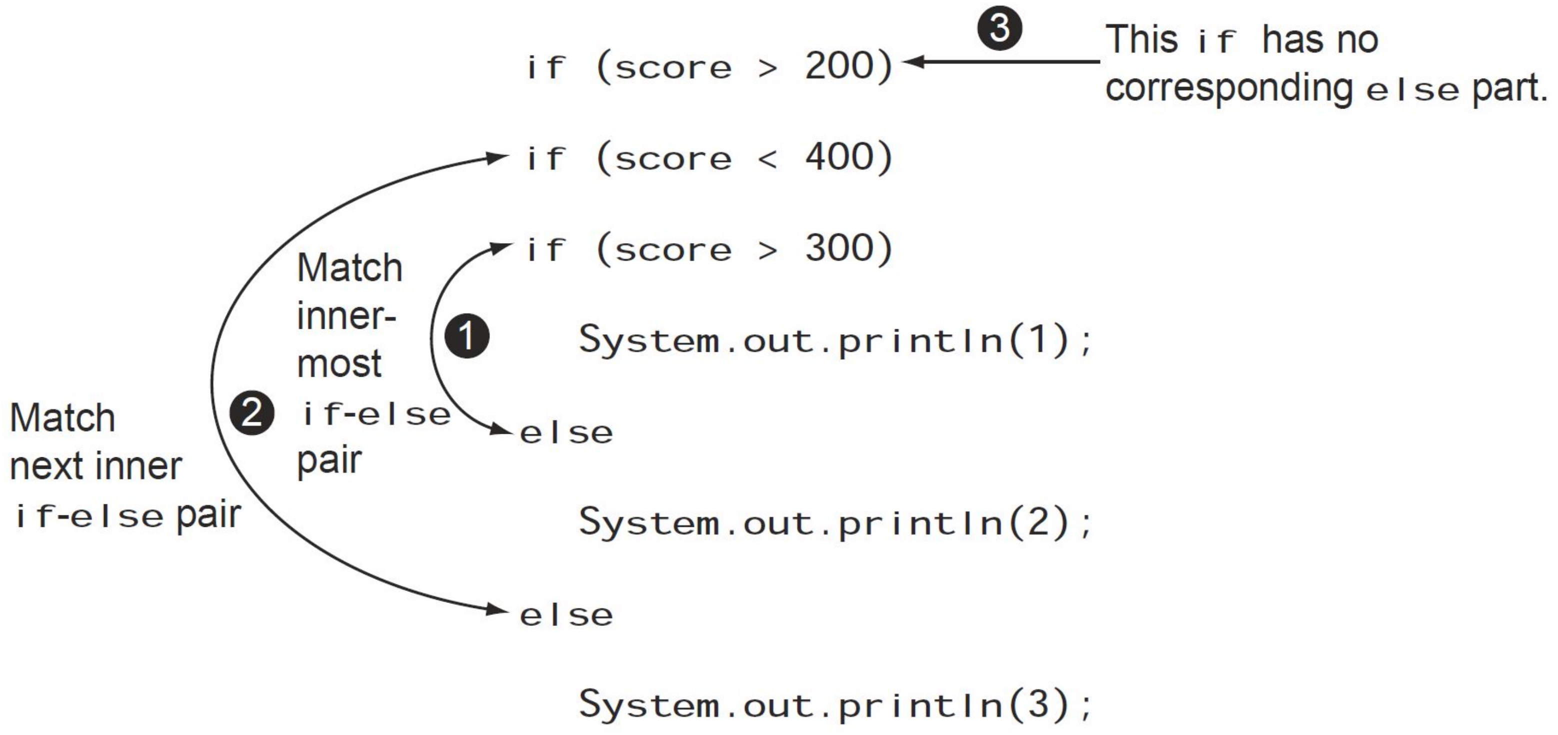
```
int score = 110;

if (score > 200) {
    if (score < 400)
        if (score > 300)
            System.out.println(1);
        else
            System.out.println(2);
} else
      System.out.println(3);
```

Determining to which `if` an `else` belongs

If the code uses curly braces to mark the start and end of the territory of an `if` or `else` construct, it can be simple to determine which `else` goes with which `if`. When the `if` constructs don't use curly braces, don't be confused by the code indentation, which may or may not be correct.

```
if (score > 200)
if (score < 400)
if (score > 300)
    System.out.println(1);
else
    System.out.println(2);
else
    System.out.println(3);
```

Determining to which `if` an `else` belongs



```
if (score > 200)
if (score < 400)
if (score > 300)
    System.out.println(1);
else
    System.out.println(2);
else
    System.out.println(3);
```

3 This if has no corresponding else part.
1 Match inner-most if-else pair
2 Match next inner if-else pair

Correctly indented code

Correct code indentation	Correct code indentation (with braces)
see code below	see code below

Correct code indentation

```
if (score > 200)
    if (score < 400)
        if (score > 300)
            System.out println(1);
        else
            System.out println(2);
    else
        System.out println(3);
```

Correct code indentation (with braces)

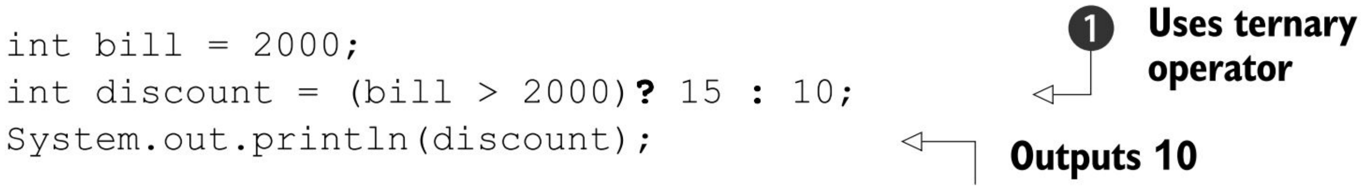
```
if (score > 200) {
    if (score < 400) {
        if (score > 300) {
            System.out println(1);
        } else {
            System.out println(2);
        }
    } else {
        System.out println(3);
    }
}
```

Ternary constructs

Ternary construct

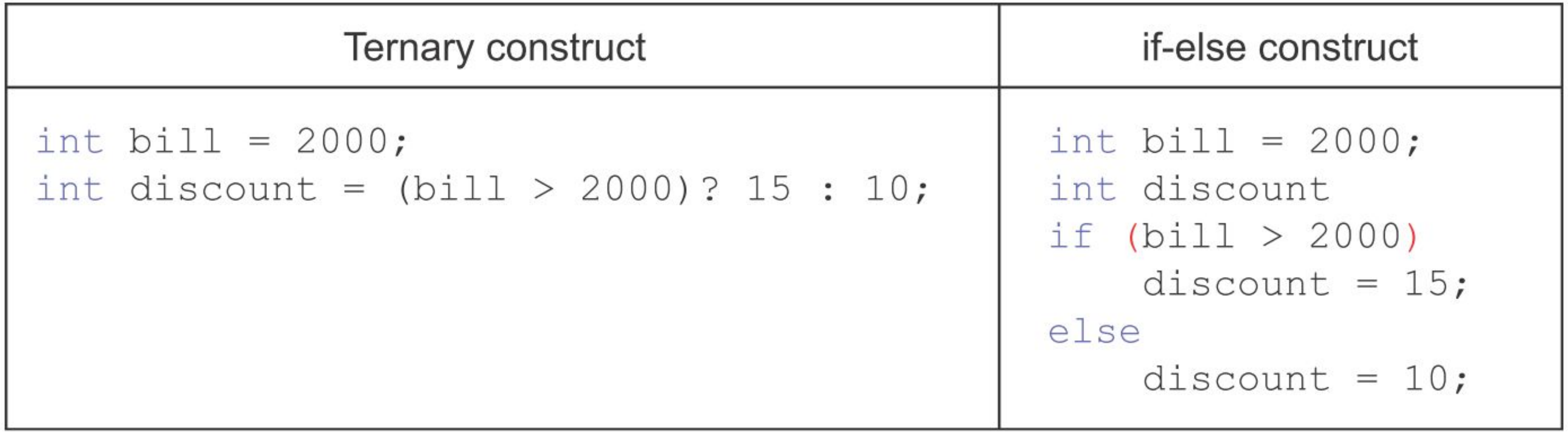
You can use a ternary operator, `?:`, to define a ternary construct.

A ternary construct can be compared to a compact `if-else` construct, used to assign a value to a variable depending on a `boolean` expression.



```
int bill = 2000;
int discount = (bill > 2000)? 15 : 10;
System.out.println(discount);
```

Ternary construct vs `if-else` construct



Ternary construct	if-else construct

`int bill = 2000;`
`int discount = (bill > 2000)? 15 : 10;`
	`int bill = 2000;`
	`int discount`
	`if (bill > 2000)`
	`    discount = 15;`
	`else`
	`    discount = 10;`

Correct usage of Ternary construct

```
int bill = 2000;
int discount = bill > 2000 ? 15 : 10;
```

OK; boolean expression not enclosed within ()

```
int bill = 2000;
int discount;
discount = (bill > 2000) ? 15 : 10;
```

OK; variable discount isn't declared in this statement

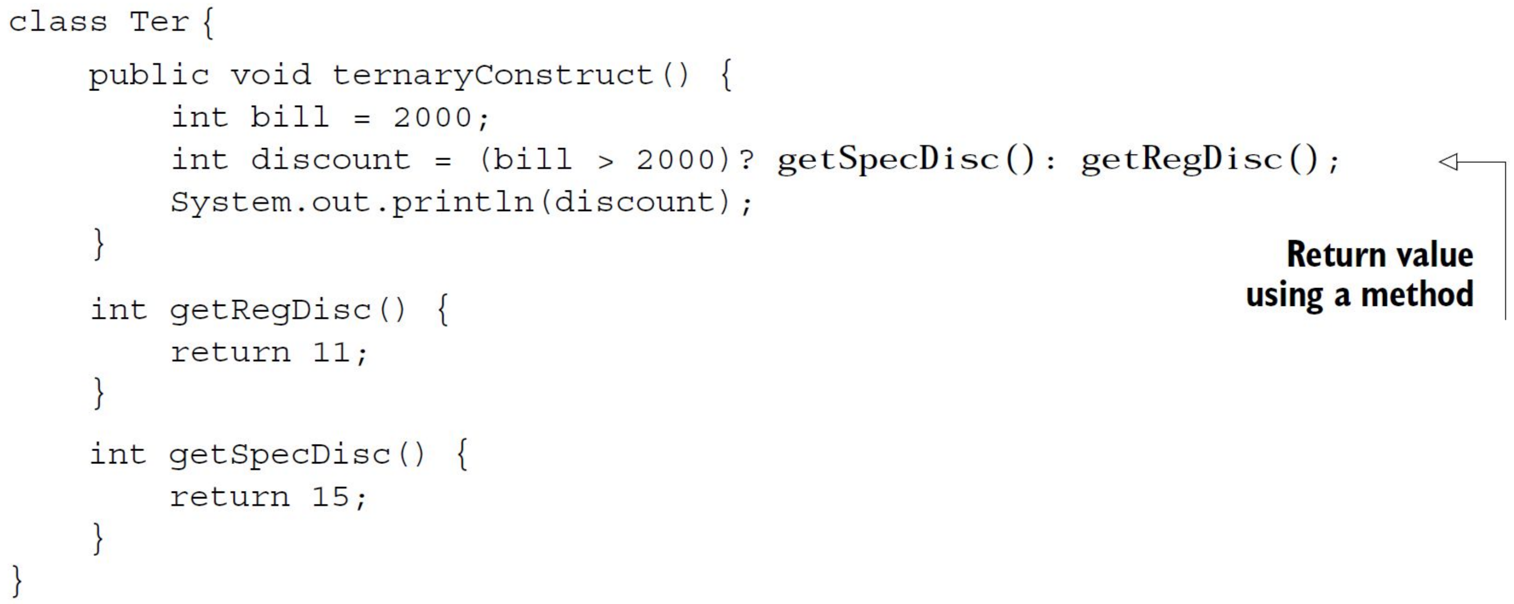
```
int bill = 2000;
int discount = (bill > 2000) ? bill-150 : bill - 100;
System.out.println(discount);
```

Assign expression to variable discount

Outputs 1900

Correct usage of Ternary construct

A method that returns a value can also be used to initialize a variable in a ternary construct:

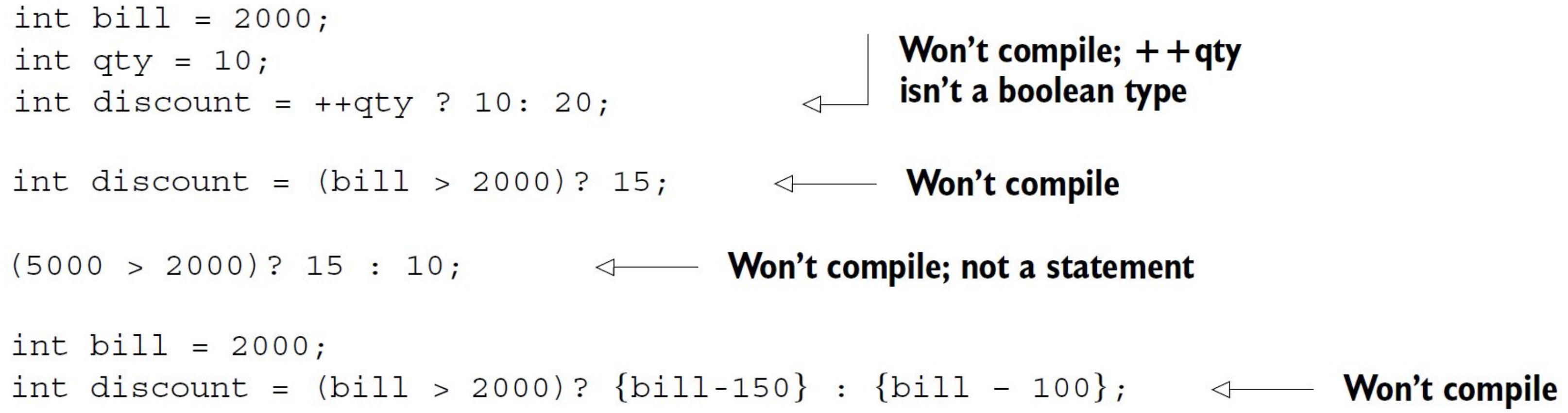


```
class Ter {
    public void ternaryConstruct() {
        int bill = 2000;
        int discount = (bill > 2000)? getSpecDisc(): getRegDisc();
        System.out.println(discount);
    }
    int getRegDisc() {
        return 11;
    }
    int getSpecDisc() {
        return 15;
    }
}
```

Return value using a method

Incorrect usage of Ternary construct

If the expression used to evaluate a ternary operator doesn't return a `boolean` or a `Boolean` value, the code won't compile.



```
int bill = 2000;
int qty = 10;
int discount = ++qty ? 10: 20;
```

Won't compile; ++qty isn't a boolean type

```
int discount = (bill > 2000)? 15;
```

Won't compile

```
(5000 > 2000)? 15 : 10;
```

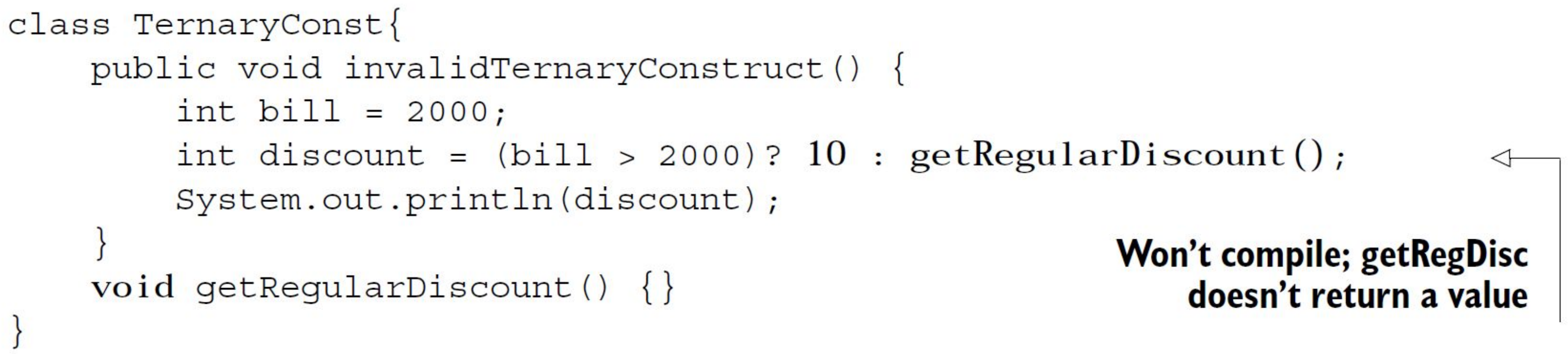
Won't compile; not a statement

```
int bill = 2000;
int discount = (bill > 2000)? {bill-150} : {bill - 100};
```

Won't compile

Incorrect usage of Ternary construct

A method that doesn’t return a value can’t be used to initialize variables in a ternary construct.



```
class TernaryConst{
    public void invalidTernaryConstruct() {
        int bill = 2000;
        int discount = (bill > 2000)? 10 : getRegularDiscount();
        System.out.println(discount);
    }
    void getRegularDiscount() {}
}
```

Won’t compile; getRegDisc doesn’t return a value

Nested ternary construct

In the following example, the `if` part of the ternary operator includes another ternary operator.

```
int bill = 2000;
int qty = 10;
int discount = (bill > 1000)? (qty > 11)? 10 : 9 : 5;
System.out.println(discount);
```

Outputs 9

Exercise #5.1: Using `if`, `if-else` statements

Create a program which reads from the keyboard two numbers and prints to the console the one that is bigger. Implement the solution using:

- `if-else` construct
- ternary construct

Exercise #5.2: Using `if`, `if-else` statements

Create a program which reads from the keyboard three numbers and prints to the console the one that is smaller than others. Implement the solution using:

- `if-else` construct
- ternary construct

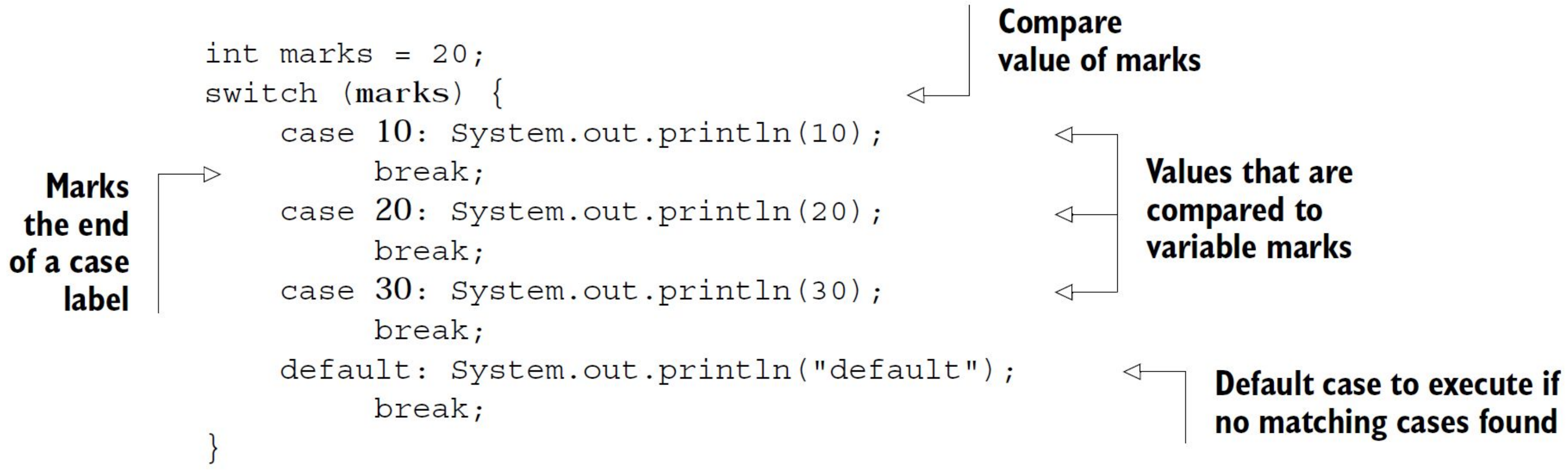
HM. Exercise #5.3: Using `if`, `if-else` statements

1. Write a Java program to get a number from the user and print whether it is positive or negative. (Input number: 35, *Expected Output* :"Number is positive")
2. Write a Java program to solve quadratic equations (Input a: 1, b: 5, c: 1; *Expected Output* : "The roots are -0.20871215252208009 and -4.7912878474779195 ")
3. Write a Java program to check whether a triangle is equilateral, scalene or isosceles. Ask the user to provide the length of each side.

The `switch` statement

Create and use a `switch` statement

You can use a `switch` statement to compare the value of a variable with multiple values. For each of these values, you can define a set of statements to execute.



```
int marks = 20;
switch (marks) {
    case 10: System.out.println(10);
         break;
    case 20: System.out.println(20);
         break;
    case 30: System.out.println(30);
         break;
    default: System.out.println("default");
         break;
}
```

Compare value of marks
Marks the end of a case label
Values that are compared to variable marks
Default case to execute if no matching cases found

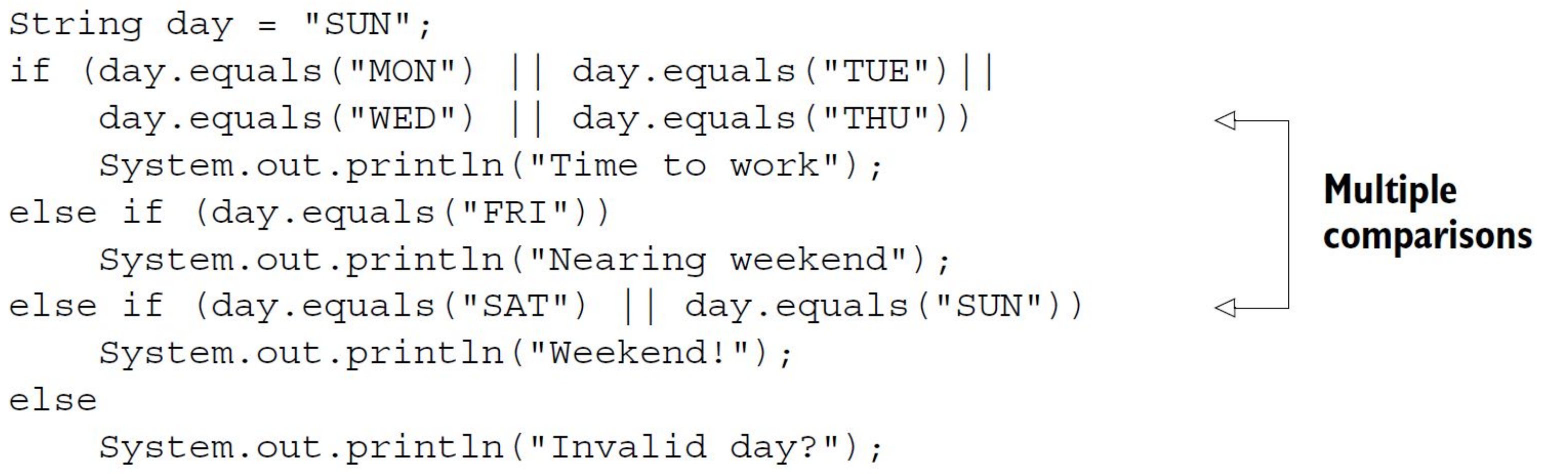
Create and use a `switch` statement

A `switch` statement can define multiple `case` labels within its `switch` block but only a single `default` label. The `default` label executes when no matching value is found in the `case` labels.

A `break` statement is used to exit a `switch` statement, after the code completes its execution for a matching `case`.

Comparing a `switch` statement with multiple `if-else` constructs

A `switch` statement can improve the readability of your code by replacing a set of related `if-else-if-else`statements with a `switch` and multiple `case` statements.



```
String day = "SUN";
if (day.equals("MON") || day.equals("TUE")||
    day.equals("WED") || day.equals("THU"))
    System.out.println("Time to work");
else if (day.equals("FRI"))
    System.out.println("Nearing weekend");
else if (day.equals("SAT") || day.equals("SUN"))
    System.out.println("Weekend!");
else
    System.out.println("Invalid day?");
```

Multiple comparisons

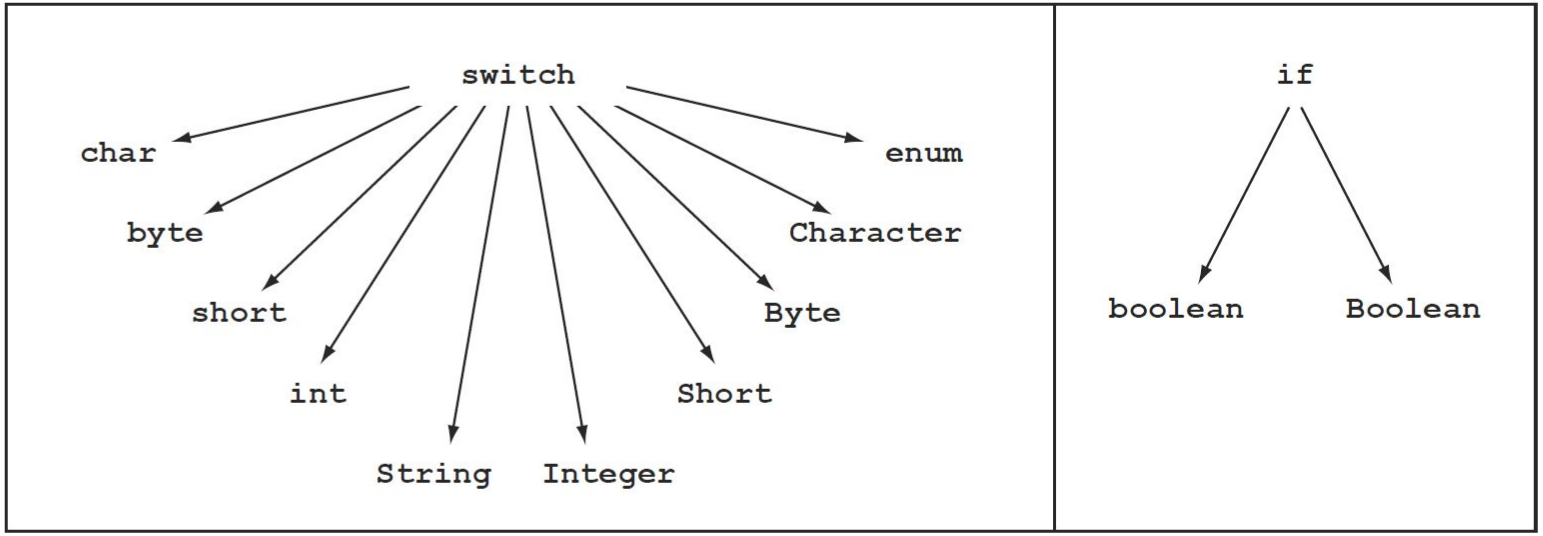
Comparing a `switch` statement with multiple `if-else` constructs

Now examine this implementation of the preceding code using the `switch` statement:

```
String day = "SUN";
switch (day) {
    case "MON":
    case "TUE":
    case "WED":
    case "THU": System.out.println("Time to work");
                break;
    case "FRI": System.out.println("Nearing weekend");
                break;
    case "SAT":
    case "SUN": System.out.println("Weekend!");
                break;
    default: System.out.println("Invalid day?");
}
```

Arguments passed to a `switch` statement

You can't use the `switch` statement to compare all types of values, such as all types of objects and primitives. There are limitations on the types of arguments that a `switch` statement can accept.



Types of arguments that can be passed to a `switch` statement and an `if` construct

Arguments passed to a `switch` statement

Apart from passing a variable to a `switch` statement, you can also pass an expression to the `switch` statement as long as it returns one of the allowed types.

```
int score = 10, num = 20;
switch (score+num) {
    // ..code
}
```

Type of score+num is int and can thus be passed as an argument to the switch statement

```
double history = 20;
switch (history) {
    // ..code
}
```

double variable can't be passed as an argument to a switch statement

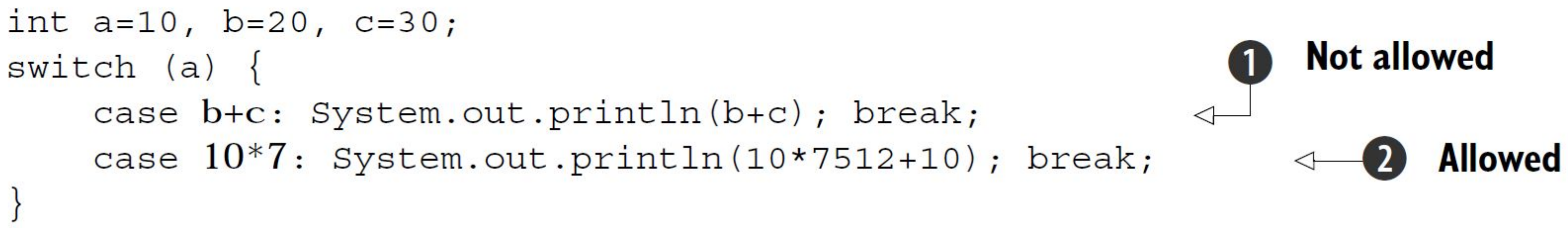
Values passed to the label case of a `switch` statement

You're constrained in a couple of ways when it comes to the value that can be passed to the `case` label in a `switch` statement:

- `case` values should be compile-time constants
- `case` values should be assignable to the argument passed to the switch statement
- `null` isn't allowed as a `case` label
- one code block can be defined for multiple cases

`case` values should be compile-time constants

The value of a `case` label must be a compile-time constant value; that is, the value should be known at the time of code compilation.



```
int a=10, b=20, c=30;
switch (a) {
    case b+c: System.out.println(b+c); break;
    case 10*7: System.out.println(10*7512+10); break;
}
```

`case` values should be compile-time constants

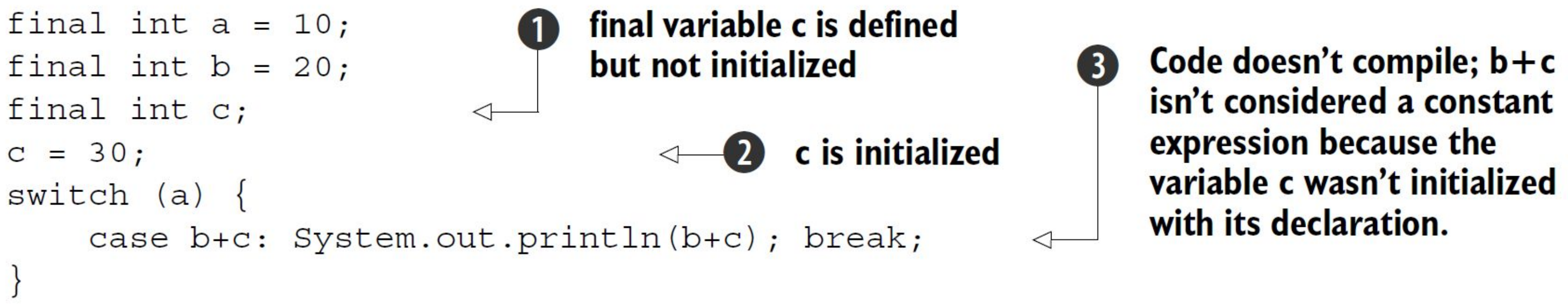
You can use variables in an expression if they're marked `final` because the value of `final` variables can't change once they're initialized.

```
final int a = 10;
final int b = 20;
final int c = 30;
switch (a) {
    case b+c: System.out.println(b+c); break;
}
```

❶ Expression b+c is compile-time constant

`case` values should be compile-time constants

You may be surprised to learn that if you don’t assign a value to a `final` variable with its declaration, it isn’t considered a compile-time constant.



```
final int a = 10;
final int b = 20;
final int c;
c = 30;
switch (a) {
    case b+c: System.out.println(b+c); break;
}
```

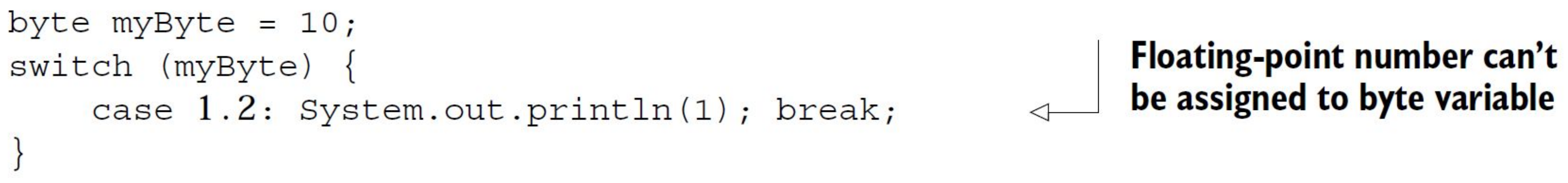
1 final variable c is defined but not initialized

2 c is initialized

3 Code doesn’t compile; b+c isn’t considered a constant expression because the variable c wasn’t initialized with its declaration.

`case` values should be assignable to the argument passed to the `switch` statement

Examine the following code, in which the type of argument passed to the `switch` statement is `byte` and the `case` label value is of the type `float`.

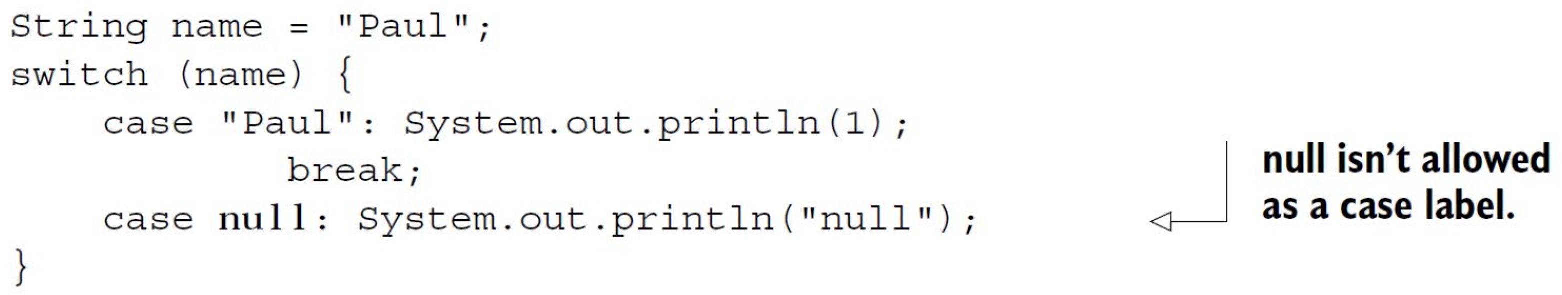


```
byte myByte = 10;
switch (myByte) {
    case 1.2: System.out.println(1); break;
}
```

Floating-point number can’t be assigned to byte variable

`null` isn't allowed as a `case` label

Code that tries to compare the variable passed to the `switch` statement with `null` won't compile.

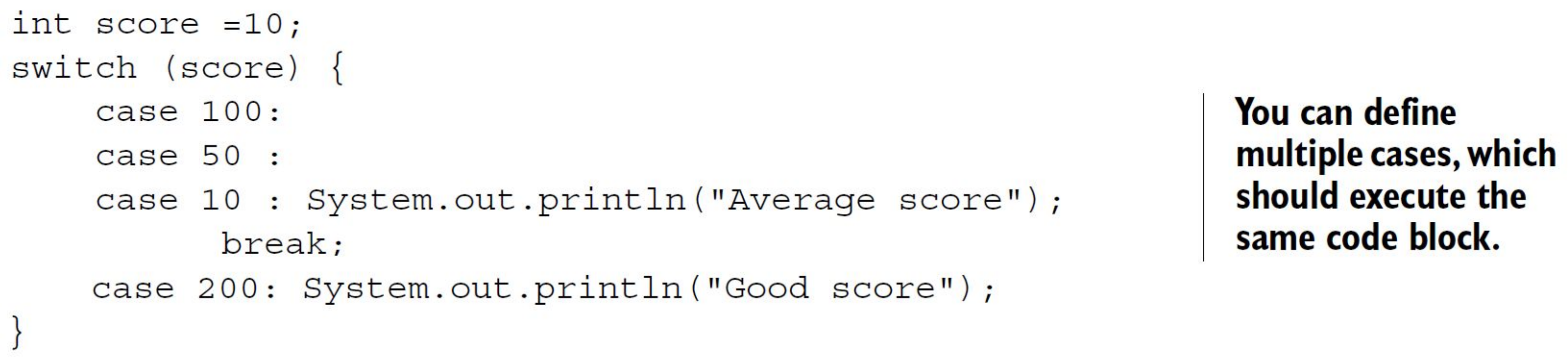


```
String name = "Paul";
switch (name) {
    case "Paul": System.out.println(1);
            break;
    case null: System.out.println("null");
}
```

null isn't allowed as a case label.

One code block can be defined for multiple cases

It’s acceptable to define a single code block for multiple `case` labels in a `switch` statement.

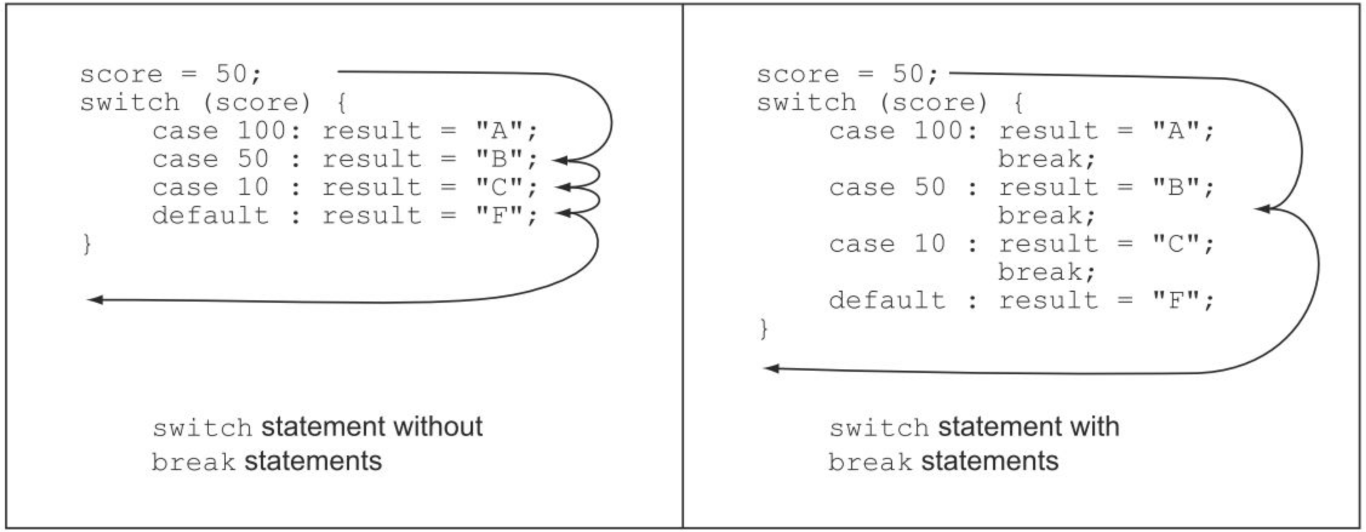


```
int score =10;
switch (score) {
    case 100:
    case 50 :
    case 10 : System.out.println("Average score");
              break;
    case 200: System.out.println("Good score");
}
```

You can define multiple cases, which should execute the same code block.

Use of `break` statements within a switch

In the absence of the `break` statement, control will *fall through* the remaining code and execute the code corresponding to all the *remaining* cases that *follow* that matching case.



```
score = 50;
switch (score) {
    case 100: result = "A";
    case 50 : result = "B";
    case 10 : result = "C";
    default : result = "F";
}
```

`switch` statement without `break` statements

```
score = 50;
switch (score) {
    case 100: result = "A";
              break;
    case 50 : result = "B";
              break;
    case 10 : result = "C";
              break;
    default : result = "F";
}
```

`switch` statement with `break` statements

Exercise #5.4: Using `switch` statements

Create a program which reads from the keyboard a number and displays the corresponding month of the year. If the number is lower than 1 or bigger than 12, display an error message.

Exercise #5.5: Using `switch` statements

Create a program which reads from the console a `String` (“MON”, “TUE”, “WED”, “THU”, “FRI”, ”SAT”, “SUN”) and displays the remaining days of the week when we we’ll have our classes (including current day). In case the user inserted a different `String`, display a corresponding message.

Example:

    Input: “MON”
    Expected result: “MON, WED, FRI”.

Example:

    Input: “TUE”
    Expected result: “WED, FRI”.

HM. Exercise #5.6: Using `switch` statements

Write a Java program that reads a number from the console and displays the name of the weekday.

HM. Exercise #5.7: Using `switch` statements

Create a program which reads from the console a `String` (“MON”, “TUE”, “WED”, “THU”, “FRI”, ”SAT”, “SUN”) and displays the days of the week when we had our lectures (including current day). In case the user inserted a different `String`, display a corresponding message.

Example:

Input: “TUE”

Expected result: “MON”.

Example:

Input: “FRI”

Expected result: “MON, WED, FRI”.

HM. Exercise #5.8: Using `switch` statements

Write a Java program that would take three inputs from the user: operator and 2 numbers. It will then perform calculation based on numbers and the entered operator. Then the result is displayed on the screen.

Example:

```
    Enter operator (either +, -, * or /): *
    Enter number1 and number2 respectively: 2 3

    2*3.5 = 7
```

HM. Exercise #5.9: Using `switch` statements

Create a Java program to display the "text mark" corresponding to a certain "numerical mark", using the following equivalence:

9,10 = I'm proud of you!
7,8 = Very good!
6 = Good.
5 = Approved.
0-4 = Fail!

Your program must ask the user for a numerical mark and display the corresponding text mark.

Implement the solution for this exercise first by using the "`if`" construct then using the "`switch`" statement.

The `for` loop

The `for` loop

A `for` loop is usually used to execute a set of statements a fixed number of times. It takes the following form:

```
for (initialization; condition; update) {
    statements;
}
```

The `for` loop. Example

Here’s a simple example:

```
int tableOf = 25;
for (int ctr = 1; ctr <= 5; ctr++) {
    System.out.println(tableOf * ctr);
}
```

1 Executes multiple times

Output:

```
25
50
75
100
125
```

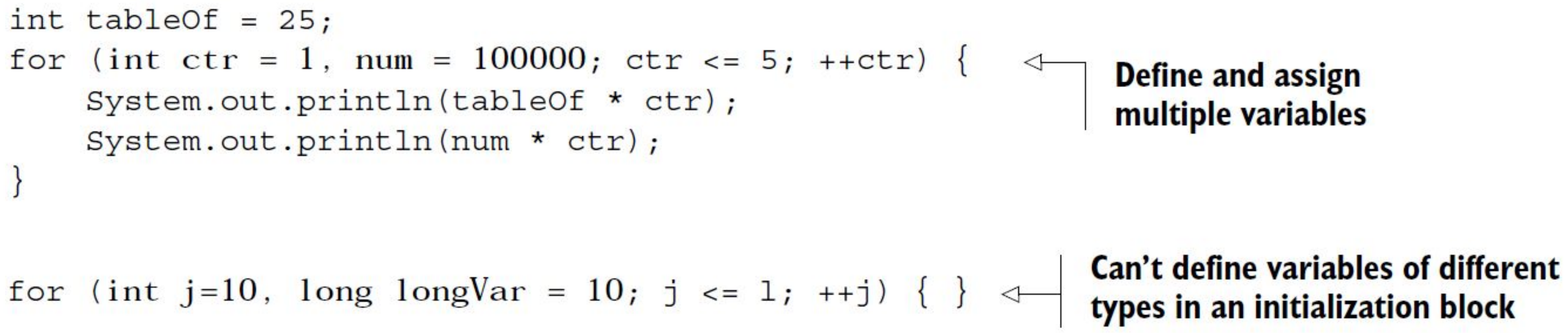
The `for` loop components

The `for` loop defines three types of statements separated with semicolons (`;`), as follows:

- Initialization statements
- Termination condition
- Update clause (executable statement)

Initialization block

An initialization block executes only once. A `for` loop can declare and initialize multiple variables in its initialization block, but the variables it declares should be of the same type.



```
int tableOf = 25;
for (int ctr = 1, num = 100000; ctr <= 5; ++ctr) {
    System.out.println(tableOf * ctr);
    System.out.println(num * ctr);
}
```

Define and assign multiple variables

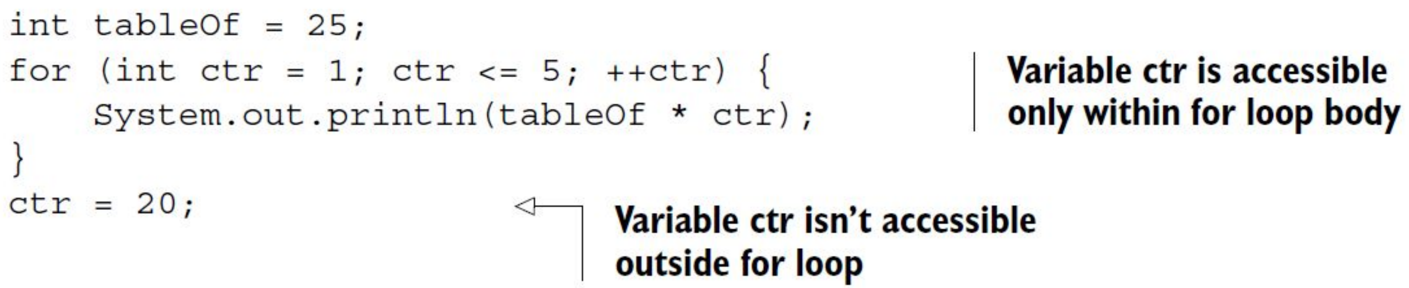
```
for (int j=10, long longVar = 10; j <= 1; ++j) { }
```

Can't define variables of different types in an initialization block

Initialization block

It’s a common programming mistake to try to use the variables defined in a `for`’s initialization block outside the `for` block.

The scope of the variables declared in the initialization block is limited to the `for` block.



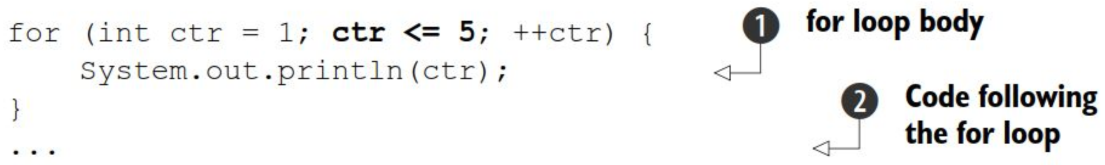
```
int tableOf = 25;
for (int ctr = 1; ctr <= 5; ++ctr) {
    System.out.println(tableOf * ctr);
}
ctr = 20;
```

Variable ctr is accessible only within for loop body

Variable ctr isn’t accessible outside for loop

Termination condition

The termination condition is evaluated once for each iteration before executing the statements defined within the body of the loop. The `for` loop terminates when the termination condition evaluates to `false`:

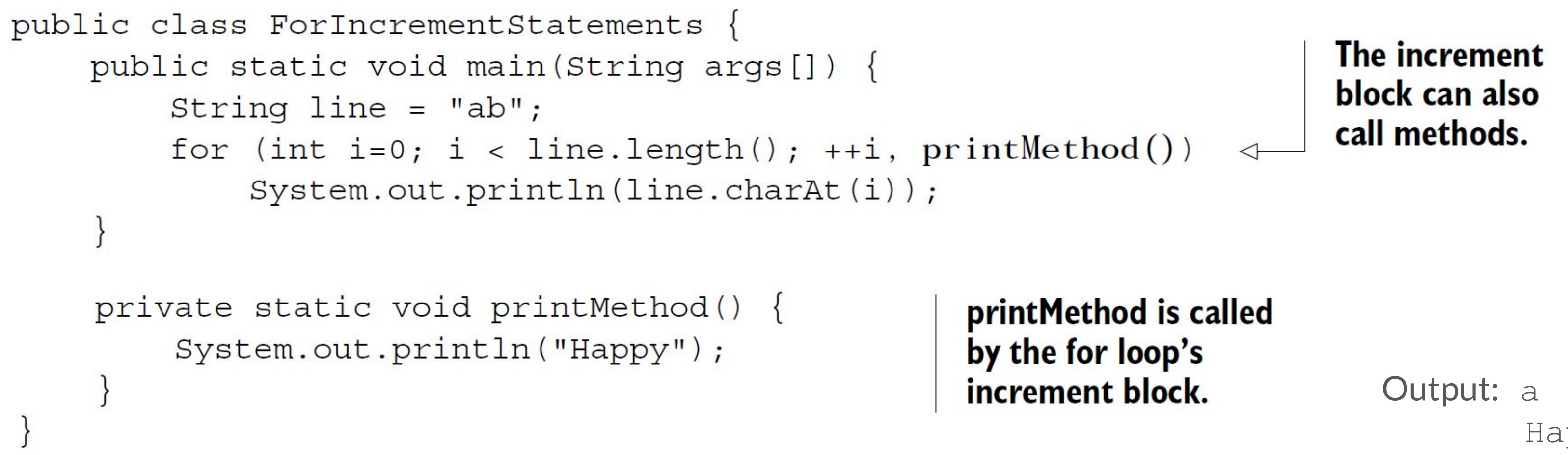


```
for (int ctr = 1; ctr <= 5; ++ctr) {
    System.out.println(ctr);
}
...
```

The update clause

The code defined in the update block executes after all the code defined in the body of the `for` loop.

You can define multiple statements in the update clause, including calls to other methods.



```
public class ForIncrementStatements {
    public static void main(String args[]) {
        String line = "ab";
        for (int i=0; i < line.length(); ++i, printMethod())
            System.out.println(line.charAt(i));
    }

    private static void printMethod() {
        System.out.println("Happy");
    }
}
```

The increment block can also call methods.

printMethod is called by the for loop's increment block.

Output:
```
a
Happy
b
Happy
```

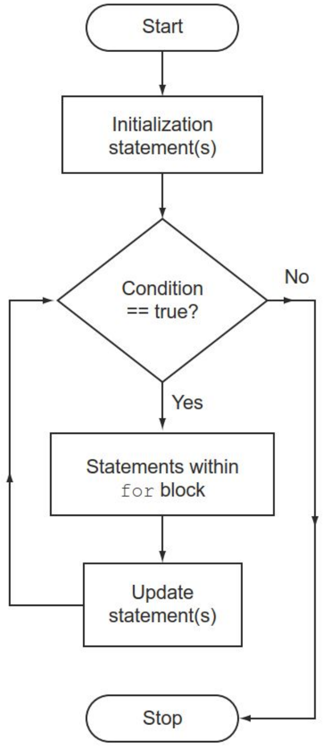
The `for` loop components

The *initialization section* may define multiple initialization statements.

The *update clause* may define multiple statements.

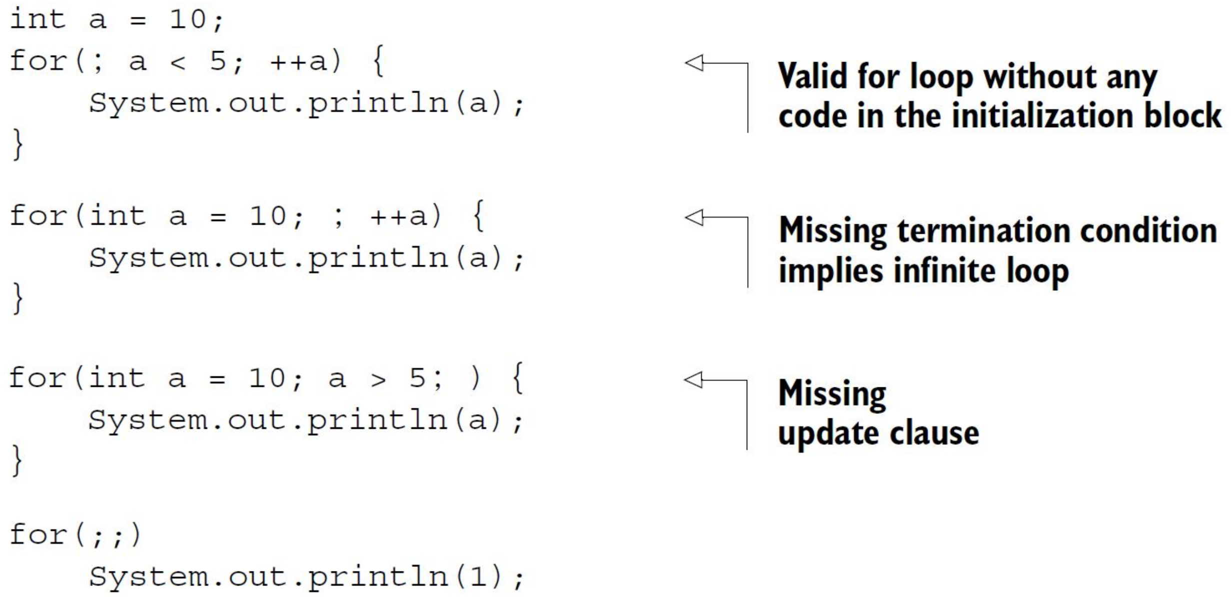
There can be only one *termination condition* for a `for` loop.

The flow of control in a `for` loop



Optional parts of a `for` statement

All three parts of a for statement - that is, *initialization block*, *termination condition*, and *update clause* - are optional. But you must specify that you aren't including a section by just including a semicolon.



```
int a = 10;
for(; a < 5; ++a) {
    System.out.println(a);
}
```

Valid for loop without any code in the initialization block

```
for(int a = 10; ; ++a) {
    System.out.println(a);
}
```

Missing termination condition implies infinite loop

```
for(int a = 10; a > 5; ) {
    System.out.println(a);
}
```

Missing update clause

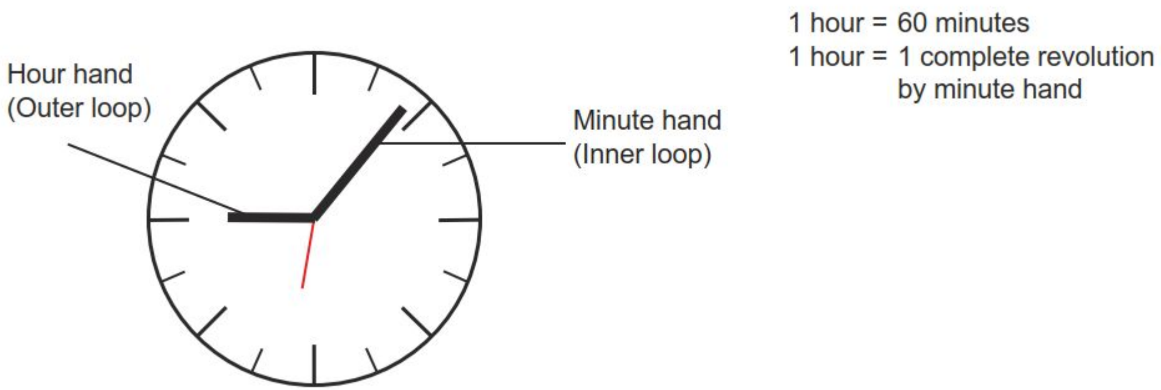
```
for(;;)
    System.out.println(1);
```

Nested `for` loop

If a loop encloses another loop, they are called ***nested loops***.

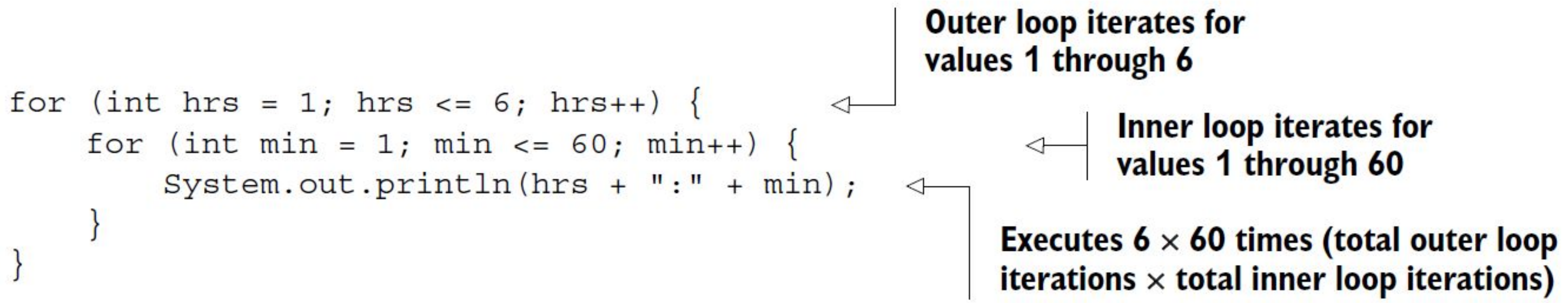
The loop that encloses another loop is called the ***outer loop***, and the enclosed loop is called the ***inner loop***.

Theoretically, there are no limits on the levels of nesting for loops.



Nested `for` loop. Example

You can use the following nested `for` loops to print out each minute (1 to 60) for hours from 1 to 6:



```
for (int hrs = 1; hrs <= 6; hrs++) {
    for (int min = 1; min <= 60; min++) {
        System.out.println(hrs + ":" + min);
    }
}
```

Outer loop iterates for values 1 through 6

Inner loop iterates for values 1 through 60

Executes 6 × 60 times (total outer loop iterations × total inner loop iterations)

Exercise #5.10: Using the `for` loop

Create a program which reads from the keyboard a positive number and displays on the screen all the numbers that are smaller than the one you inserted, but bigger than 0 (zero).

Display the numbers in the ascending order, then in descendent order.

HM. Exercise #5.11: Using the `for` loop

Create a program which reads from the keyboard a positive number and displays on the screen all the numbers that are smaller than the one you inserted, but bigger than 0 (zero) and divisible by 2.

Resources

if and else in Java
(https://www.codesdope.com/java-decide-if-or-else/)

Switch Case in Java
(https://syntaxdb.com/ref/java/switch)

Loops in Java
(https://www.javatpoint.com/java-for-loop)

Java Flow Control Interview Questions (+ Answers)
(https://www.baeldung.com/java-flow-control-interview-questions)

Java Fundamentals

Lesson 5: Flow Control

End.

Speaker: Nicolae Sîrbu
Alexandru Umaneț