(/)

Java Flow Control Interview Questions (+ Answers)

Last modified: November 11, 2018

by baeldung (https://www.baeldung.com/author/baeldung/)

Java (https://www.baeldung.com/category/java/) +

Interview (https://www.baeldung.com/tag/interview/)

This article is part of a series:

1. Introduction

Control flow statements allow developers to use decision making, looping and branching to conditionally change the flow of execution of particular blocks of code.

In this article, we'll go through some flow control interview questions that might pop up during an interview and, where appropriate; we'll implement examples to understand their answers better.

2. Questions

Q1. Describe the *if-then* and *if-then-else* statements. What types of expressions can be used as conditions?

Both statements tell our program to execute the code inside of them only if a particular condition evaluates to *true*. However, the *if-then-else* statement provides a secondary path of execution in case the if clause evaluates to *false*:

```java
if (age >= 21) {
    // ...
} else {
    // ...
}
```

Unlike other programming languages, Java only supports *boolean* expressions as conditions. If we try to use a different type of expression, we'll get a compilation error.

Q2. Describe the *switch* statement. What object types can be used in the *switch* clause?

Switch allows the selection of several execution paths based on a variables' value.

Each path is labeled with *case* or *default*, the *switch* statement evaluates each *case* expression for a match and executes all statements that follow the matching label until a *break* statement is found. If it can't find a match, the *default* block will be executed instead:

```
switch (yearsOfJavaExperience) {
    case 0:
        System.out.println("Student");
        break;
    case 1:
        System.out.println("Junior");
        break;
    case 2:
        System.out.println("Middle");
        break;
    default:
        System.out.println("Senior");
}
```

We can use *byte*, *short*, *char*, *int*, their wrapped versions, *enum*s and *String*s as *switch* values.

Q3. What happens when we forget to put a *break* statement in a *case* clause of a *switch*?

The *switch* statement falls-trough. This means that it will continue the execution of all *case* labels until if finds a *break* statement, even though those labels don't match the expression's value.

Here's an example to demonstrate this:

```
int operation = 2;
int number = 10;

switch (operation) {
    case 1:
        number = number + 10;
        break;
    case 2:
        number = number - 4;
    case 3:
        number = number / 3;
    case 4:
        number = number * 10;
        break;
}
```

After running the code, *number* holds the value 20, instead of 6. This can be useful in situations when we want to associate the same action with multiple cases.

Q4. When is preferable to use a *switch* over an *if-then-else* statement and vice versa?

A *switch* statement is better suited when testing a single variable against many single values or when several values will execute the same code:

```
switch (month) {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        days = 31;
        break;
case 2:
    days = 28;
    break;
default:
    days = 30;
}
```

An *if-then-else* statement is preferable when we need to check ranges of values or multiple conditions:

```
if (aPassword == null || aPassword.isEmpty()) {
    // empty password
} else if (aPassword.length() < 8 || aPassword.equals("12345678")) {
    // weak password
} else {
    // good password
}
```

Q5. What types of loops does Java support?

Java offers three different types of loops: *for*, *while*, and *do-while*.

A *for* loop provides a way to iterate over a range of values. It's most useful when we know in advance how many times a task is going to be repeated:

```
for (int i = 0; i < 10; i++) {
     // ...
}
```

A *while* loop can execute a block of statements while a particular condition is *true*:

```
while (iterator.hasNext()) {
    // ...
}
```

A *do-while* is a variation of a *while* statement in which the evaluation of the *boolean* expression is at the bottom of the loop. This guarantees that the code will execute at least once:

```
do {
    // ...
} while (choice != -1);
```

Q6. What is an *enhanced for* loop?

Is another syntax of the *for* statement designed to iterate through all the elements of a collection, array, enum or any object implementing the *Iterable* interface:

```
for (String aString : arrayOfStrings) {
    // ...
}
```

Q7. How can you exit anticipatedly from a loop?

Using the *break* statement, we can terminate the execution of a loop immediately:

```
for (int i = 0; ; i++) {
    if (i > 10) {
        break;
    }
}
```

Q8. What is the difference between an unlabeled and a labeled *break* statement?

An unlabeled *break* statement terminates the innermost *switch*, *for*, *while* or *do-while* statement, whereas a labeled *break* ends the execution of an outer statement.

Let's create an example to demonstrate this:

```
int[][] table = { { 1, 2, 3 }, { 25, 37, 49 }, { 55, 68, 93 } };
boolean found = false;
int loopCycles = 0;

outer: for (int[] rows : table) {
    for (int row : rows) {
        loopCycles++;
        if (row == 37) {
            found = true;
            break outer;
        }
    }
}
```

When the number 37 is found, the labeled *break* statement terminates the outermost *for* loop, and no more cycles are executed. Thus, *loopCycles* ends with the value of 5.

However, the unlabeled *break* only ends the innermost statement, returning the flow of control to the outermost *for* that continues the loop to the next *row* in the *table* variable, making the *loopCycles* end with a value of 8.

Q9. What is the difference between an unlabeled and a labeled *continue* statement?

An unlabeled *continue* statement skips to the end of the current iteration in the innermost *for*, *while*, or *do-while* loop, whereas a labeled *continue* skips to an outer loop marked with the given label.

Here's an example that demonstrates this:

```
int[][] table = { { 1, 15, 3 }, { 25, 15, 49 }, { 15, 68, 93 } };
int loopCycles = 0;

outer: for (int[] rows : table) {
    for (int row : rows) {
        loopCycles++;
        if (row == 15) {
            continue outer;
        }
    }
}
```

The reasoning is the same as in the previous question. The labeled *continue* statement terminates the outermost *for* loop.

Thus, *loopCycles* ends holding the value 5, whereas the unlabeled version only terminates the innermost statement, making the *loopCycles* end with a value of 9.

Q10. Describe the execution flow inside a *try-catch-finally* construct.

When a program has entered the *try* block, and an exception is thrown inside it, the execution of the *try* block is interrupted, and the flow of control continues with a *catch* block that can handle the exception being thrown.

If no such block exists then the current method execution stops, and the exception is thrown to the previous method on the call stack. Alternatively, if no exception occurs, all *catch* blocks are ignored, and program execution continues normally.

A *finally* block is always executed whether an exception was thrown or not inside the body of the *try* block.

Q11. In which situations the *finally* block may not be executed?

When the JVM is terminated while executing the *try* or *catch* blocks, for instance, by calling *System.exit(),* or when the executing thread is interrupted or killed, then the finally block is not executed.

Q12. What is the result of executing the following code?

```
public static int assignment() {
    int number = 1;
    try {
        number = 3;
        if (true) {
            throw new Exception("Test Exception");
        }
        number = 2;
    } catch (Exception ex) {
        return number;
    } finally {
        number = 4;
    }
    return number;
}

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

The code outputs the number 3. Even though the *finally* block is always executed, this happens only after the *try* block exits.

In the example, the *return* statement is executed before the *try-catch* block ends. Thus, the assignment to *number* in the *finally* block makes no effect, since the variable is already returned to the calling code of the *testAssignment* method.

Q13. In which situations *try-finally* block might be used even when exceptions might not be thrown?

This block is useful when we want to ensure we don't accidentally bypass the clean up of resources used in the code by encountering a *break*, *continue* or *return* statement:

```java
HeavyProcess heavyProcess = new HeavyProcess();
try {
    // ...
    return heavyProcess.heavyTask();
} finally {
    heavyProcess.doCleanUp();
}
```

Also, we may face situations in which we can't locally handle the exception being thrown, or we want the current method to throw the exception still while allowing us to free up resources:

```java
public void doDangerousTask(Task task) throws ComplicatedException {
    try {
        // ...
        task.gatherResources();
        if (task.isComplicated()) {
            throw new ComplicatedException("Too difficult");
        }
        // ...
    } finally {
        task.freeResources();
    }
}
```

Q14. How does *try-with-resources* work?

The *try-with-resources* statement declares and initializes one or more resources before executing the *try* block and closes them automatically at the end of the statement regardless of whether the block completed normally or abruptly. Any object implementing *AutoCloseable* or *Closeable* interfaces can be used as a resource:

```
try (StringWriter writer = new StringWriter()) {
    writer.write("Hello world!");
}
```

3. Conclusion

In this article, we covered some of the most frequently asked questions appearing in technical interviews for Java developers, regarding control flow statements. This should only be treated as the start of further research and not as an exhaustive list.

Good luck in your interview.

Next »
Java Exceptions Interview Questions (+ Answers) (https://www.baeldung.com/java-exceptions-interview-questions)

« Previous
Java Generics Interview Questions (+Answers) (https://www.baeldung.com/java-generics-interview-questions)

>> CHECK OUT THE COURSE (/ls-course-end)

Learning to "Build your API
with Spring"?

Enter your email address

>> Get the eBook

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP CLIENT (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)

SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE COURSES (HTTPS://COURSES.BAELDUNG.COM)
CONSULTING WORK (/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
THE FULL ARCHIVE (/FULL_ARCHIVE)
WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORS (/EDITORS)
OUR PARTNERS (/PARTNERS)
ADVERTISE ON BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)