♥♦ Regular Expressions (aka Regex) by examples for Java developers

Posted on October 11, 2014 by Arulkumaran Kumaraswamipillai — No Comments ↓

Q1 How will you go about implementing the following validation rules for a user name?

— user name must be between 2 and 17 characters long.
— valid characters are A to Z, a to z, 0 to 9, . (full-stop), _ (underscore) and – (hyphen)
— user name must begin with an alphabetic character.
— user name must not end with a . (full stop) or _ (underscore) or – (hyphen).

A1 The above rules can be implemented with a regular expression as shown below:

```
1  import java.util.regex.Matcher;
2  import java.util.regex.Pattern;
3
4  public class UserNameRegex {
5
6      public static final String PATTERN =
7                              "^[a-zA-Z][a-zA-Z0-9._-]
8      public static final Pattern p = Pattern.comp
9
10     public static boolean apply(String userName)
11         Matcher matcher = p.matcher(userName);
12         return matcher.find();
13     }
14 }
```

Not compiling the regular expression can be costly if Pattern.matches( ) is used over and over again with the same expression in a loop or frequent method calls because the matches( ) method will re-compile the expression every time it is used. You can also re-use the Matcher object for different input strings by calling the method reset( ).

What does this pattern mean?

^	[a-zA-Z]	[a-zA-Z0-9._-]{0,15}	[a-zA-Z0-9]	$
Beginning of a line	Valid start characters. Should start with an alphabet. Must occur once.	Valid characters in the middle. Minimum occurrences 0 and max occurrences 15.	Valid end characters. Should not end with "." "-" or " ". Must occur once.	End of a line.

How will you test this? Test with JUnit. The junit.jar file must be in the classpath.

```
1  import org.junit.Assert;
2  import org.junit.Test;
3
4  public class UserNameRegexTest {
5
6      @Test
7      public void testMinLength() {
8          Assert.assertFalse("can't be <2", UserNa
9      }
10
11     @Test
12     public void testMaxLength() {
13         Assert.assertFalse("Can't be >17", UserN
14                 .apply("Jonathon-Christopher"));
15     }
16
17     @Test
18     public void testCantEndWith() {
```


Other Interview Q&A t
Free Java Interviev

16 Technical Key Areas

open all | close all
- Best Practice (6)
- Coding (26)
- Concurrency (6)
- Design Concepts (7)
- Design Patterns (11)
- Exception Handling (3
- Java Debugging (21)
- Judging Experience In
- Low Latency (7)
- Memory Managemen
- Performance (13)
- QoS (8)
- Scalability (4)
- SDLC (6)
- Security (13)
- Transaction Managen

80+ step by step Java Tutorials

open all | close all
- Setting up Tutorial (6)
- Tutorial - Diagnosis (2
- Akka Tutorial (9)
- Core Java Tutorials (2
- Hadoop & Spark Tuto
- JEE Tutorials (19)
- Scala Tutorials (1)
- Spring & HIbernate Tu
- Tools Tutorials (19)
- Other Tutorials (45)

```
19          Assert.assertFalse("can't end with . - _
20                  .apply("s.g.r."));
21      }
22
23      @Test
24      public void testMustStartWith() {
25          Assert.assertFalse("Must start with an a
26                  UserNameRegex.apply("23Lucky"));
27      }
28
29      @Test
30      public void validNames() {
31          Assert.assertTrue("Min Length 2", UserNa
32          Assert.assertTrue("Max Length 17", UserN
33                  .apply("Peter-Christopher"));
34          Assert.assertTrue(". - _ allowed in the
35                  UserNameRegex.apply("s.g-h_k.r")
36          Assert.assertTrue("end with numeric", Us
37                  .apply("Lucky23"));
38
39      }
40  }
```

Q2 How would you go about validating a supplied password in your code that must adhere to the following conditions?

— Must contain a digit from 0-9.
— Must contain one uppercase character.
— Must contain one lowercase character.
— The length must be between 6 to 15 characters long.

A2 It can be achieved with a regular expression. But unlike the previous question, this is a bit more complicated than it looks. You might be tempted to write something like,

```
1 [0-9A-Za-z]{6,15}
```

But the above regex is not going to meet the requirement. The above regex will only ensure that the valid characters are used and the password is of correct length (i.e. between 6 and 15 characters). But it won't ensure that there is at least "one digit from 0-9, one lower case and one uppercase". So, the above regex will return true for passwords like

```
1 Johnpwd           //no digit in the password
2 john123           //no uppercase character in the
```

This is where the regex with **LOOK AHEAD** comes to the rescue. The pattern below uses "look ahead" with the syntax **?=**. The regex pattern with look ahead is shown below:

```
1 ((?=[A-Za-z]*[0-9])(?=[0-9A-Z]*[a-z])(?=[0-9a-z]*
```

```
1  public static void correct() {
2      String PATTERN =
3          "((?=[A-Za-z]*[0-9])(?=[0-9A-Z]*[a-z])(?
4      Pattern p = Pattern.compile(PATTERN);
5      Matcher matcher = p.matcher("John123");
6      System.out.println(matcher.find( ));
7      matcher = p.matcher("john123");
8      System.out.println(matcher.find( ));
9      matcher = p.matcher("johnpwd");
10     System.out.println(matcher.find( ));
11 }
```

Don't be overwhelmed by long regex patterns as shown above. Once you divide them into **smaller chunks** as shown below, things become much clearer.

(?=[A-Za-z]*[0-9])	**?=** → means look ahead for * → means 0 to many **[A-Za-z]*** → means zero to many valid characters like A-Z and a-z. **[0-9]** → means followed by a digit. The [0-9] can also be substituted with "**\d**" meaning a digit. When used within Java, you need to use "\\d", the first back slash is to escape the "\" in "\d" since "\" is an escape character in Java.
(?=[0-9A-Z]*[a-z])	Similar.. Positively look ahead for presence of a-z followed by 0 or many valid characters like 0-9 and A-Z.
(?=[0-9a-z]*[A-Z])	Similar.. Positively look ahead for presence of A-Z followed by 0 or many valid characters like 0-9 and a-z.
((?=[A-Za-z]*[0-9])(?=[0-9A-Z]*[a-z])(?=[0-9a-z]*[A-Z])**){6,15}**	The overall length must be at least 6 characters and maximum allowed length is 15.

Note: The **look ahead** and **look behind** constructs are collectively known as "**look around**" constructs. Unlike the greedy quantifiers discussed, the "look arounds" actually match characters, but then give up the match and only return the result with "match" or "no match". That is why they are called "assertions". They do not consume characters in the string like [0-9A-Za-z]{6,15}, but only **assert** whether a match is possible or not.

The look ahead constructs are **non-capturing**. If you want to capture the values matched, for example to debug or use the captured values, the above regex can be modified with additional parentheses to capture the matched values.

```
1 (?=([A-Za-z]*)([0-9]))(?=([0-9A-Z]*)([a-z]))(?=([
```

You can display the captured values as shown below:

```
1  public static void correctWithGroup() {
2          String PATTERN =
3           "((?=([A-Za-z]*)([0-9]))(?=([0-9A-Z]*)(
4          Pattern p = Pattern.compile(PATTERN);
5          Matcher matcher = p.matcher("John123");
6          System.out.println(matcher.find());
7
8          int count = matcher.groupCount();
9          //0, and 1 are outside parentheses, whic
10         for (int i = 2; i <= count; i++) {
11             System.out.println(matcher.group(i))
12         }
13 }
```

Outputs:

```
1
2 true
3 John
4 1
5 J
6 o
7
8 J
9
```

As you can see,

(?=([A-Za-z]*)([0-9])) matches John followed by 1, i.e. presence of a digit.
(?=([0-9A-Z]*)([a-z])) matches J followed by o, i.e. presence of a lower-case alphabet.
(?=([0-9a-z]*)([A-Z])) matches nothing followed by J, i.e. presence of an uppercase letter.

Q3 What does .*, +, and ? mean with respect to regular expressions? Where do you look for the summary of Java

regular expression constructs?
A3 *, +, and ? are known as (greedy) quantifiers as they quantify the preceding character(s). For example,

. → matches any character.
.* → matches any character repeated 0 or more times.
.+ → matches any character repeated 1 or more times.
.? → matches any character repeated 0 or 1 time. This means optional.

[a-zA-Z]* → Any alphabet repeated 0 or more times.
[a-zA-Z]+ → Any alphabet repeated 1 or more times.
[a-zA-Z]? → Any alphabet repeated 0 or 1.

Note: These are not wild characters. *, +, and ? are **regex repetitions**. The {x,y} is also used for repetition. For example, [a]{3,5} means the letter "a" repeated at least 3 times or a maximum of 5 times. In fact, internally the **Pattern.java** class implements,

a* as a{0, 0x7FFFFFFF}
a+ as a{1, 0x7FFFFFFF}

The values in square brackets (i.e. [ ]) are known as **character sets**. The ., *, ?, etc are escaped and used as literal values within the square brackets. Here are some examples of the character sets.

[aeiou] → matches exactly one lowercase vowel.

[^aeiou] → matches a character that ISN'T a lowercase vowel (**^** inside [ ] means **NOT**).

^[a-z&&[^aeiou]]*$ → matches any character other than a vowel anchored between start (^) and end ($). This is a character class **subtraction** regex. The "**&&**" means **intersection**.

[a-d[m-p]] → Matches characters a to d and m to p. This is a **union** regex.

[a-z&&[d-f]] → Matches only d, e, and f. This is an **intersection** regex.

[x-z[\\p{Digit}]] → matches x-z and 0-9. Similar to [x-z0-9] or [x-z[\\d]]. The "\p" stands for POSIX character classes.

^[aeiou] – matches a lowercase vowel anchored at the **beginning of a line**

[^^] → matches a character that isn't a caret '^' .

^[^^] → matches a character that isn't a caret at the **beginning of a line**.

^[^.]. → matches anything but a literal period, followed by "any" character, at the beginning of a line

[.*]* → matches a contiguous sequence of optional dots (.) and asterisks (*)

[aeiou]{3} – matches 3 consecutive lowercase vowels (all not necessarily the same vowel)

\[aeiou\] → matches the string "[aeiou]". "\" means escape.

Q4 What does grouping mean with regards to regular expressions? Would you prefer capturing or non-capturing group?
A4 A **group** is a pair of parentheses used to group sub patterns. For example, c(a|u)t matches cat or cut. A group also captures the matching text within the parentheses. The groups are numbered from left to right, outside to inside. For example, (x(y*))+(z*) has 3 **explicit** and 1 **implicit** groups. The implicit group 0 is the entire match.

implicit group 0: (x(y*))+(z*)
explicit group 1: (x(y*))
explicit group 2: (y*)
explicit group 3: (z*)

Capturing groups are numbered by counting their opening parentheses from left to right. The captured sub sequence may be used later in the expression, via a back reference, and may also be retrieved from the matcher once the match operation is complete. Groups beginning with **?:** are pure, **non-capturing groups** that do not capture text and do not count towards the group total. For example, (?:x(?:y*))+(?:z*) will only have group 0 being the entire match.

```
1  import java.util.regex.Matcher;
2  import java.util.regex.Pattern;
3
4  public class RegexGroup {
5
6      public static void main(String[ ] args) {
7          CharSequence inputStr = "xyyxyyzz";
8          String patternCapturing = "(x(y*))+(z*)"
9          String patternNonCapturing= "(?:x(?:y*))
10         System.out.println("====Capturing=====")
11         apply(inputStr,patternCapturing);
12         System.out.println("====Non Capturing===
13         apply(inputStr,patternNonCapturing);
14     }
15
16     public static void apply(CharSequence inputS
17         // Compile and use regular expression
18         Pattern pattern = Pattern.compile(patter
19         Matcher matcher = pattern.matcher(inputS
20         boolean matchFound = matcher.find();
21
22         if (matchFound) {
23             // Get all groups for this match
24             for (int i = 0; i <= matcher.groupCo
25                 System.out.println("group " + i
26                         + matcher.group(i));
27             }
28         }
29     }
30 }
```

Output:

```
1
2  ====Capturing=====
3  group 0 --> xyyxyyzz
4  group 1 --> xyy
5  group 2 --> yy
6  group 3 --> zz
7  ====Non Capturing=====
8  group 0 --> xyyxyyzz
9
```

The following sample code illustrates substitution, where the captured group numbers can be used for the subsequent

matches. For example, the term “marketing” and “sales” can be captured and reused as these terms appear more than once. Once within the URL and once outside the URL as per the code snippet shown below.

```
1  import java.util.regex.Matcher;
2  import java.util.regex.Pattern;
3
4  public class CapturingParenthesis {
5
6      /**
7       * To match a URI that starts with one of th
8       * http://www.site.com/marketing/ or http://
9       */
10     public static void main(String[] args) {
11         final String input = "You can try either
12                 + "http://www.site.com/marketing
13                 + "queries or http://www.site.co
14                 + "related queries and further i
15
16         System.out.println("========Capturing===
17         String capPat1 = "http://www.site.com/(m
18         execute(input, capPat1);
19
20         System.out.println("=====Capturing with
21         String capPat2 = "http://www.site.com/(m
22         execute(input, capPat2);
23
24         System.out.println("========Non capturin
25         // can be more efficient
26         String nonCapPat1 = "http://www.site.com
27         execute(input, nonCapPat1);
28
29         System.out.println("===Non capturing no
30         // can't use it for substituting the pre
31         String nonCapPat2 = "http://www.site.com
32         execute(input, nonCapPat2);
33     }
34
35     public static void execute(String input, Str
36         Pattern p = Pattern.compile(pattern);
37         Matcher matcher = p.matcher(input);
38
39         while (matcher.find()) {
40             System.out.print("Start index: " + m
41             System.out.print(" End index: " + ma
42             System.out.println("group= " + match
43         }
44     }
45 }
```

Output:

```
1
2  ========Capturing=========
3  Start index: 19 End index: 49 group= http://www.
4  Start index: 83 End index: 109 group= http://www
5  =====Capturing with substitution====
```

```
6  Start index: 19 End index: 63 group= http://www.
7  Start index: 83 End index: 119 group= http://www
8  ========Non capturing=========
9  Start index: 19 End index: 49 group= http://www.
10 Start index: 83 End index: 109 group= http://www
11 ===Non capturing no substitution==
12
```

Note: The "\1" (group 1) captures either "marketing" or "sales" when the capturing pattern is used. Since "\" has to be escaped in Java, it is represented as "\\1". The .* matches the word "for" as it means match any character 0 or more times.

Group capturing incur a small-time penalty each time you use them. If you don't really need to capture the text inside a group, prefer using non-capturing groups.

Q5 What do you understand by **greedy, reluctant, and possessive quantifiers**? What do you understand by the term **"backtracking"**?

A5

Greedy	Reluctant	Possessive
[a-z][A-Z]* [a-z][A-Z]+ [a-z][A-Z]? [a-z][A-Z]{1,2} [a-z][A-Z]{2,} [a-z][A-Z]{,6}	[a-z][A-Z]*? [a-z][A-Z]+? [a-z][A-Z]?? [a-z][A-Z]{1,2}? [a-z][A-Z]{2,}? [a-z][A-Z]{,6}?	[a-z][A-Z]*+ [a-z][A-Z]++ [a-z][A-Z]?+ [a-z][A-Z]{1,2}+ [a-z][A-Z]{2,}+ [a-z][A-Z]{,6}+
As the name implies a greedy quantifier will first try to **match as many characters as possible** from an input string, even if this means that the input string will not have sufficient characters left in it to match the rest of the regular expression. If this happens, the greedy quantifier will **backtrack**, returning characters until an overall match is found or until there are no more characters.	As the name implies a reluctant (or lazy) quantifier, on the other hand, will first try to **match as few characters in the input string as possible**.	As the name implies a possessive quantifier always match the entire input string, trying once (and only once) for a match. **Unlike the greedy quantifier, the possessive quantifiers never backtrack**, even if doing so would allow the overall match to succeed.

```
1  import java.util.regex.Matcher;
2  import java.util.regex.Pattern;
3
4  public class GreedyReluctantPossessive {
5
6      public static void main(String[ ] args) {
```

```
7           String input = "xfooxxfooxxxxfooxxxxxxxxx
8
9           // this is the default for *, +, and ?.
10          // fetches the whole String eagerly and
11          String eagerPattern = ".*foo";
12          System.out.println("=======Greedy=======
13          execute(input, eagerPattern);
14
15          // Backtrack to the beginning and start
16          String reluctantPattern = ".*?foo";
17          System.out.println("=======Reluctant====
18          execute(input, reluctantPattern);
19
20          // fetches the whole String eagerly, but
21          String possessivePattern = ".*+foo";
22          System.out.println("=======Possessive===
23          execute(input, possessivePattern);
24
25          // possessive pattern is more suited for
26          String input2 = "a=\"xfoo\" some text b=
27                                              + "som
28          possessivePattern = "\"xfoo\"*+";
29          System.out.println("=====Possessive matc
30          execute(input2, possessivePattern);
31      }
32
33      private static void execute(String input, St
34          Pattern p = Pattern.compile(pattern);
35          Matcher matcher = p.matcher(input);
36          while (matcher.find( )) {
37              System.out.print("Start index: " + m
38              System.out.print(" End index: " + ma
39              System.out.println("group= " + match
40          }
41      }
42  }
```

Output:

```
1
2  =======Greedy========
3  Start index: 0 End index: 27 group= xfooxxfooxxx
4  =======Reluctant========
5  Start index: 0 End index: 4 group= xfoo
6  Start index: 4 End index: 9 group= xxfoo
7  Start index: 9 End index: 16 group= xxxxfoo
8  Start index: 16 End index: 27 group= xxxxxxxxfoo
9  =======Possessive============Possessive match f
10 Start index: 2 End index: 8 group= "xfoo"
11 Start index: 21 End index: 26 group= "xfoo
12
```

Q6 Why did the string “xfooxxfooxxxxfooxxxxxxxxxfoofo” return no match when used with a possessive quantifier?

A6 Even if you try to match “xfooxxfooxxxxfooxxxxxxxxxfoo” without the last “fo”, the possessive quantifier would match nothing because the **possessive quantifiers will not give up the matches on a backtrack**, and the .*+ matches your

entire string including the last “foo” and then there’s nothing for “foo” to match in the pattern “.*+foo”. So use possessive quantifiers only when you know that what you’ve matched should never be backtracked. For example, “[^f]*+.*foo” or more importantly matching text in quotes as demonstrated above with “\”xfoo\”*+”.

Performance testing

Always test your regular expressions with negative and positive test cases. Generally the **possessive quantifiers are more efficient than the reluctant quantifiers, and the reluctant quantifiers are more efficient than the greedy quantifiers**. When a regex is used in a performance critical area of your application, it would be wise to test it first. Write a benchmark application that tests it against different inputs. Remember to test different lengths of inputs, and also inputs that almost match your pattern.

What is “backtracking”?

Like most regex engines, Java uses NFA (Non deterministic Finite Automaton) approach. This means the engine scans the regex components one by one, and progresses on the input string accordingly. However, it will go back in order to explore matching alternatives when a “dead end” is found. Alternatives result from regex structures such as quantifiers (*, +, ?) and alternation (e.g. a|b|c|d). This exploration technique is called **backtracking**. In a more simpler term, backtracking means return to attempt an untried option.

Optimization tips

The .* and alternation (e.g. a|b|c) regexes must be used sparingly. For example, if you want to retrieve everything between two “@” characters in an input string, instead of using “@(.*)@”, it’s much better to use “@([^@]*)@”. This regex can be further optimized by minimizing the backtracking with the help of a possessive quantifier “*+” instead of a greedy quantifier “*”.

Say you are using the pattern "@([^@]*)@" with a long input string that does not have any "@". Because "*" is a greedy quantifier, it will grab all the characters until the end of the string, and then it will backtrack, giving back one character at a time. The expression will fail only when it can't backtrack anymore. If you change the pattern to use "@([^@]*+)@" with a possessive quantifier "*+", the new pattern fails faster because once it has tried to match all the characters that are not a "@", it does not backtrack; Instead it fails right there. Take care not to compromise correctness for a small optimization.

Regular expressions like "(COLON|COMMA|COLUMN)" have a reputation for being slow, so watch out for them. Firstly, the order of alternation counts. So, place the more common options in the front to be matched faster. Secondly, extract out the more common ones. For example, CO(MMA|L(ON|UMN)). Since COMMA is more common, it is used first. The "CO" and "L" are extracted out as they are more common.

Q7 How will you use "**negative look ahead**" to split a string of digits say 230_000L to individual array of digits using a regex?

A7

```
1
2  import java.util.Arrays;
3
4  public class Test {
5
6      public static void main(String[] args) {
7          long input = 230_000L;
8          String inputStr = Long.valueOf(input).to
9          String[] split = inputStr.split("(?!^)")
10         System.out.println(Arrays.asList(split))
11     }
12 }
13
```

(?!X) –> is NEGATIVE LOOK AHEAD syntax. X is the "^". The **^** matches the beginning of a string, so the regex matches at every position that is **not the beginning of the string**, and inserts a split there.

Output:

```
1
2 [2, 3, 0, 0, 0, 0]
3
```

Q8 How will you split the following CSV values on "," ?

```
1
2 String input = "John, Peter, Samuel , Erik";
3
```

A8 Using a regex in the String.split(….) method.

```
1
2 String input = "John, Peter, Samuel , Erik";
3 String[] splits = input.split("\\s*,\\s*");
4
```

\s -> space
\\s* –> first "\" is an escape character, and "*" means zero or more spaces.

Q9 What if you have "," as a value within the quotes as in "Peter, Smith", how will you split the CSV values with regex?

```
1
2 String input = "John, \"Peter, Smith\", Samuel ,
3
```

A9

```
1
2  package com.mytutorial;
3
4  public class CsvSplitter {
5      public static void main(String[] args) {
6          String input = "John, \"Peter, Smith\",
7          String[] split = input.split(",(?=([^\"]
8
9          for (String s : split) {
10             System.out.println(s.trim());
11         }
12     }
13 }
14
```

Output:

```
1
2 John
3 "Peter, Smith"
4 Samuel
5 Erik
6
```

(?=) –> Positive Look Ahead

[^\"]*\" –> NO quotes 0 or more times, and followed by a quote.

,([^\"]*\"[^\"]*\")* –> , followed by zero or more of anything within quotes.

[^\"]*$ –> ends with anything other than a quote 0 or more times.

Popular Posts

♦ 11 Spring boot interview questions & answers

828 views

♦ Q11-Q23: Top 50+ Core on Java OOP Interview Questions & Answers

768 views

18 Java scenarios based interview Questions and Answers

400 views

001A: ♦ 7+ Java integration styles & patterns interview questions & answers

389 views

01b: ♦ 13 Spring basics Q8 – Q13 interview questions & answers

296 views

♦ 7 Java debugging interview questions & answers

293 views

01: ♦ 15 Ice breaker questions asked 90% of the time in Java job interviews with hints

286 views

♦ 10 ERD (Entity-Relationship Diagrams) Interview Questions and Answers

280 views

♦ Q24-Q36: Top 50+ Core on Java classes, interfaces and generics interview questions & answers

240 views

001B: ♦ Java architecture & design concepts interview questions & answers

202 views

Bio | Latest Posts

Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.**945+** paid members. join my LinkedIn Group. **Reviews**

About Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.**945+** paid members. join my LinkedIn Group. **Reviews**

‹ Regular Expressions interview Q&A for Java developers

If Java did not have a stack or map, how would you …. ›

Posted in Regex

Leave a Reply

Logged in as geethika. Log out?

Comment

Post Comment

Empowers you to open more doors, and fast-track

Technical Know Hows

☀ Java generics in no time ☀ Top 6 tips to transforming your thinking from OOP to FP ☀ How does a HashMap internally work? What is a hashing function?
☀ 10+ Java String class interview Q&As ☀ Java auto un/boxing benefits & caveats ☀ Top 11 slacknesses that can come back and bite you as an experienced Java developer or architect

Non-Technical Know Hows

☀ 6 Aspects that can motivate you to fast-track your career & go places ☀ Are you reinventing yourself as a Java developer? ☀ 8 tips to safeguard your Java career against offshoring ☀ My top 5 career mistakes

Prepare to succeed

☀ Turn readers of your Java CV go from “Blah blah” to “Wow”? ☀ How to prepare for Java job interviews? ☀ 16 Technical Key Areas ☀ How to choose from multiple Java job offers?

Select Category

© Disclaimer

 ↑