Strings and Regex

Java Developer

StayAhead Training; January, 2023

Strings

Strings and Regex

- The String class (java.lang) is the most commonly used of all Java classes
- Strings objects are immutable
- Some String objects are reused to improve performance & conserve memory
- String objects, unlike any other type, can be created like primitives
- The String class has many instance methods

Immutable Objects

Strings and Regex

- An immutable object is one that cannot be modified
- An immutable object's field(s) cannot be written to
- Classes that produce immutable objects are increasingly common in Java
- Immutable objects suit applications where all data, even historical data, is of value - rather than modify, we transform (create new data from existing data)
- Strings are immutable - each String method produces new data from existing data, it does not modify the object on which it's called

Java Memory*

Strings and Regex

- When you start a Java app the JVM consumes some of the OS's memory
- JVM memory is divided into two main parts: stack and heap
- The stack stores methods whilst in execution and their local variables*
- The heap stores objects
- Recall that a reference variable stores a reference to an object
- *A local variable is one that is declared inside a method

The String Pool

Strings and Regex

- The String Pool is a part of the heap allocated specifically for String objects
- String objects placed in the Pool may be reused
- As String objects are immutable, there's no reason to duplicate them, e.g.

```
// both variables reference the same String object
var s1 = "Hello";
var s2 = "Hello";
```

- The placing of a String object in the Pool is called interning

Garbage Collection*

Strings and Regex

```
var client = new Client();
client = null;
```

- Setting the variable, client, to null means that the variable no longer references a Client object; it now references nothing
- The Client object still exists in the heap (for now); this is a memory leak - memory that is occupied but no longer in use
- Every so often the JVM halts execution of your app to reclaim memory that is occupied but no longer in use - this process is called garbage collection

The String Pool

Strings and Regex

- Before Java 7 interned String objects (those placed in the String Pool) were not garbage collected like those objects in normal heap
- This lead to the occasional out-of-memory exception
- Since Java 7 String Pool objects are garbage collected

String Creation

Strings and Regex

- String creation may take one of two forms, e.g.

```
// assign from the Pool or create new and intern
var s1 = "Hello";

// create new and do NOT intern
var s2 = new String("Hello");
```

Strings and the Equality Operator*

Strings and Regex

- When used with reference variables, the equality operator (==) tests for equality of reference (memory address of the object), not equality of content

```
// The "Hello" object is interned and reused
var s1 = "Hello";
var s2 = "Hello";
System.out.println(s1 == s2); // true

// The "Goodbye" object is duplicated in the heap
var s3 = new String("Goodbye");
var s4 = new String("Goodbye");
System.out.println(s3 == s4); // false
```

Escape Characters

Strings and Regex

- The escape character (\) is used to insert special characters into a String
- For example, how does one include double quotes in a String?

```
var quote = "He said \"I did it, it was me\"";
```

- Special characters: `\"` (dbl quote) `\\` (backslash) `\n` (newline) `\t` (tabspace)

Concatenation

Strings and Regex

- The plus (+) symbol is used to concatenate Strings with other Strings and/or values, e.g.

```
var name = "John Smith";
var age = 29;
System.out.println("Name: " + name + ", age: " + age);
```

- This form of concatenation is painful to write and hard to read;
  there are better options including String.format and StringBuilder

Text Blocks (since Java 13)

Strings and Regex

- A text block is a triple-quoted String in which whitespace and quotes are interpreted literally without having to include escape characters, e.g.

```
var json = """
  {
    "title": "Brave New World",
    "author": "Huxley, Aldous"
  }
""";
```

Format Strings

Strings and Regex

- The String class's static **format** method is similar to System.out.printf except that it returns the formatted String rather than writing it to stdout, e.g.

```
var desc = "Ball";
var price = 4.99;
var fs = String.format("|%-10s|£%9.2f|", desc, price);
// fs is: |Ball      |£     4.99|
```

- `%-10s` is a left justified (-), 10 char width any type (s)

- `%9.2f` is a right justified, 9 char width floating point number (f) with 2 digits after the decimal point (9 is the total width including the decimal point)

Format Strings

Strings and Regex

- Since Java 13 the String class has an instance method named **formatted** that behaves like the static format method, e.g.

```
var desc = "Ball";
var price = 4.99;
var fs = "|%-10s|£%9.2f|".formatted(desc, price);
// fs is: |Ball      |£     4.99|
```

- The formatted method may be applied to text blocks too

Length

Strings and Regex

- A String object is effectively an array of characters (`char[]`)
- Every String class has a **length** method which returns the length of the underlying array, e.g.

```
var str = "Hello world";
var strLength = str.length(); // 11
```

- NB: methods can be called on String literals too, e.g.

```
var strLength = "Hello world".length(); // 11
```

Transformative Methods

Strings and Regex

- The **concat** method is used to concatenate two Strings, e.g.

```
var s1 = "Hello ";
var s2 = "world";
var s3 = s1.concat(s2);
assert s3.equals("Hello world");
```

Transformative Methods

Strings and Regex

- The **indent** method (since Java 12) is used to indent each line by the given number of spaces e.g.

```
var s1 = "Hello world";

var s2 = s1.indent(5);

assert s2.equals("     Hello world");
```

Transformative Methods

Strings and Regex

- The **repeat** method is used to repeat the String n times, e.g.

```
var s1 = "ha";
var s2 = s1.repeat(3);
assert s2.equals("hahaha");
```

Transformative Methods

Strings and Regex

- The **replace** method is used to replace one part of a String with another, e.g.

```
var s1 = "Hello world";

var s2 = s1.replace("world", "Java");

assert s2.equals("Hello Java");
```

Transformative Methods

Strings and Regex

- The **split** method is used to split a String using a given delimiter, e.g.

```
var s1 = "Hello world";
var s2 = s1.split(" ");
assert s2[0].equals("Hello");
assert s2[1].equals("world");
```

Transformative Methods

Strings and Regex

- The **strip** method is used to remove leading and trailing whitespace, e.g.

```
var s1 = "   Hello   ";
var s2 = s1.strip();
assert s2.equals("Hello");
```

Transformative Methods

Strings and Regex

- The **substring** method is used to extract a part of the String, e.g.

```
var s1 = "Hello world";

var s2 = s1.substring(6);

assert s2.equals("world");
```

- The substring method is overloaded such that you can specify both start index (inclusive) and end index (exclusive)

Transformative Methods

Strings and Regex

- The **toLowerCase** method is used to convert the chars to lower case, e.g.

```
var s1 = "Hello world";

var s2 = s1.toLowerCase();

assert s2.equals("hello world");
```

Transformative Methods

Strings and Regex

- The **toUpperCase** method is used to convert the chars to upper case, e.g.

```
var s1 = "Hello world";

var s2 = s1.toUpperCase();

assert s2.equals("HELLO WORLD");
```

Search Methods

Strings and Regex

- The **contains** method is used to determine if the String contains a given sequence of characters (substring), e.g.

```
var s1 = "Hello world";

var result = s1.contains("world");

assert result;
```

Search Methods

Strings and Regex

- The **indexOf** method is used to get the first index of the given character, e.g.

```
var s1 = "Hello world";

var result = s1.indexOf('o');

assert result == 4;
```

- The String class has a lastIndexOf method also

Comparative Methods

Strings and Regex

- The **compareTo** method is used to compare the String with another lexicographically (e.g. a < b etc.), e.g.

```
var s1 = "abc";

var s2 = "ace";

var result = s1.compareTo(s2);

assert result < 1;
```

- The return value is: negative if s1 is less than s2; zero if s1 is equal to s2; positive if s1 is greater than s2

Comparative Methods

Strings and Regex

- The **endsWith** method is used to test if the String ends with the given String e.g.

```
var s1 = "Hello world";

var result = s1.endsWith("world");

assert result;
```

Comparative Methods

Strings and Regex

- The **equals** method is used to test if the String equals the given String e.g.

```
var s1 = "Hello world";

var s2 = "hello world";

var result = s1.equals(s2);

assert !result;
```

Comparative Methods

Strings and Regex

- The **equalsIgnoreCase** method is used to test if the String equals the given String ignoring case differences e.g.

```
var s1 = "Hello world";

var s2 = "hello world";

var result = s1.equalsIgnoreCase(s2);

assert result;
```

Comparative Methods

Strings and Regex

- The **isBlank** method returns true if the String is empty or contains whitespace only (spaces, tabspaces, newline characters etc.), e.g.

```
var s1 = "\n\t ";

var result = s1.isBlank();

assert result;
```

Comparative Methods

Strings and Regex

- The **isEmpty** method returns true if the String's length is zero, e.g.

```
var s1 = "";

var result = s1.isEmpty();

assert result;
```

Comparative Methods

Strings and Regex

- The **startsWith** method is used to test if the String starts with the given String e.g.

```
var s1 = "Hello world";

var result = s1.startsWith("Hello");

assert result;
```

Method Chaining*

Strings and Regex

- Those String methods that return a new String can be chained together to perform a sequence of operations on one line, e.g.

```
var s1 = "   Hello world   ";

var s2 = s1.strip().substring(0, 5).toLowerCase();

assert s2.equals("hello");
```

StringBuilder

Strings and Regex

- The StringBuilder class provide yet another alternative to concatenation
- The class's append method is overloaded to accept any type and returns a reference to itself to support method chaining, e.g.

```
var name = "John Smith";
var age = 29;
var builder = new StringBuilder();
var sentence = builder.append("Name:").append(name)
  .append(", age:").append(age).toString();
// sentence is "Name: John Smith, age: 29"
```

Pattern Matching

Strings and Regex

- Pattern matching is the testing of a String to see if it matches a pattern
- Pattern matching is commonly used to perform validation and to extract information from structured data
- For example, email addresses could be said to follow a pattern:
  - some characters (excluding whitespace)
  - the at (@) symbol
  - some more characters
  - the dot
  - yet more characters

Regular Expressions

Strings and Regex

- A regular expression (regex) is a sequence of special characters that forms a pattern for pattern matching

- There are various types of regex special characters:
  - characters
  - character classes (groups)
  - boundary matchers
  - quantifiers
  - logical operators
  - and more…

Regular Expressions

Strings and Regex

- Regex characters (a selection):
  - `\\` backslash
  - `\n` newline
  - `\t` tabspace
- Other characters, e.g. a, b, c, etc. are interpreted literally
- The String "abc" matches the pattern "abc"

Regular Expressions

Strings and Regex

- Regex character classes (a selection):
  - `.` any character
  - `[abc]` a, b, or c
  - `[^abc]` any character except a, b, or c (negation)
  - `[a-z]` a thru z (range)
  - `\d` a digit
  - `\D` a non-digit
  - `\s` a whitespace character
  - `\S` a non-whitespace character
  - `\w` a word character
  - `\W` a non-word character

Regular Expressions

Strings and Regex

- Regex boundary matchers (a selection):
  - `^` the beginning of a line
  - `$` the end of a line
  - `\b` a word boundary
  - `\B` a non-word boundary

- Regex quantifiers (a selection):
  - `X?` X, once or not at all
  - `X*` X, zero or more times
  - `X+` X, one or more times
  - `X{5,}` X, five or more times

Regular Expressions

Strings and Regex

- Some simple regular expressions, e.g.

Regex	Matches
`Error`	"Error"
`\d+`	One or more digits
`[A-z]{3}`	3 characters in the range A-z
`Error.*`	"Error" followed by zero or more characters

The matches Method*

Strings and Regex

- The String class has a **matches** method that accepts a String regex and returns true if the String matches the given pattern, e.g.

```
var s1 = "BAR123";

var pattern = "^[A-Z]{3}\\d{3}$";

var result = s1.matches(pattern);

assert result;
```

- NB: \\d instead \d because the backslash is the Java escape character

The Pattern and Matcher Classes

Strings and Regex

- The Pattern and Matcher classes enable the matching of patterns in a way that provides more information than does String's matches method, e.g.

```
var s1 = "abc123def";

var pattern = Pattern.compile("\\d{3}");

var matcher = pattern.matcher(s1);

matcher.find();

var startingIndex = matcher.start(); // 3
```