JAVA 21 API NEW FEATURES

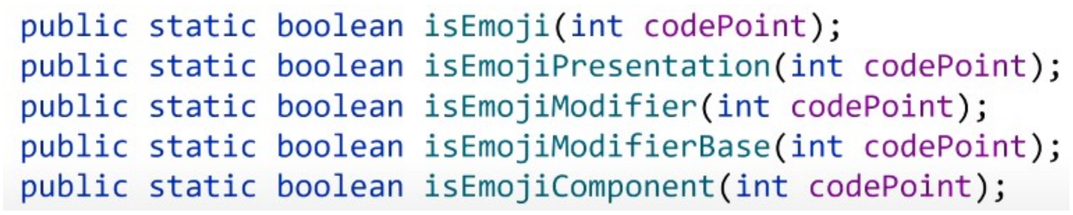
Index

- New methods on String, Character, StringBuilder and StringBuffer

- An overload of the *indexOf()* method has been added, that allows you to look for a single character or a substring between a begin index and an end index.

```java
public static void main( String[] args ) {
   String message = "apple tomato banana orange";
   var index = message.indexOf("banana", 5, 25);
   System.out.println(index);
}
```

- The Character class saw the addition of several methods to support emojis.



```java
public static boolean isEmoji(int codePoint);
public static boolean isEmojiPresentation(int codePoint);
public static boolean isEmojiModifier(int codePoint);
public static boolean isEmojiModifierBase(int codePoint);
public static boolean isEmojiComponent(int codePoint);
```

- The *StringBuffer* and *StringBuilder* classes got some attention too. You can now repeat a string of characters a number of times with the repeat() method. And bytheway there is an overload of this method, that takes a single character.

```java
public static void main( String[] args ) {
   var sb = new StringBuilder();
   sb.repeat("one ", 5);
   System.out.println(sb);
}
```

- Naming your capturing groups in regular expressions

- There is an update to the regular expression engine, that now supports named groups.

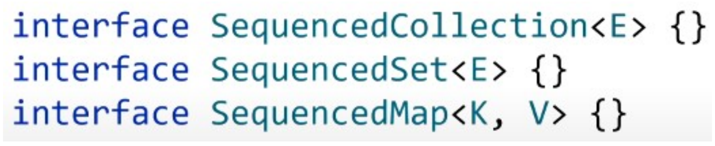
- Named groups allow you to give capturing groups a name, and get them in your code using that name.

```java
public static void main( String[] args ) {
   // Naming the capturing groups in regular expressions
   String line = "1; New York; 8 336 817";

   Pattern pattern =
      Pattern.compile("""
         (?<index>\\d+);\
         (?<city>[ a-zA-Z]+);\
         (?<population>[ \\d]+)$""");

   Matcher matcher = pattern.matcher(line);
   if (matcher.matches()) {
      var index = matcher.group("index");
      var city = matcher.group("city");
      var population = matcher.group("population");

      System.out.println(index);
      System.out.println(city);
```

```
        System.out.println(population);
    }
}
```

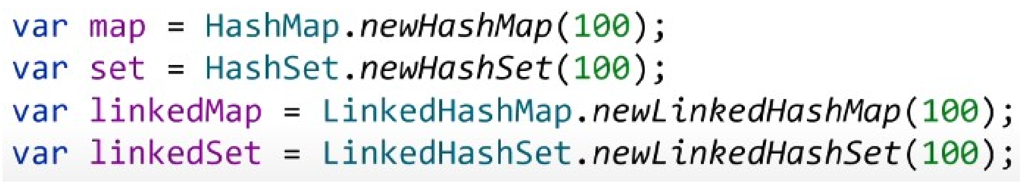
- Additions to the Collection framework

- Several elements have been added to the Collections framework, and the biggest of them is of course the two new interfaces SequencedCollection and SequencedSet.



```
interface SequencedCollection<E> {}
interface SequencedSet<E> {}
interface SequencedMap<K, V> {}
```

- You have several factory methods to create different types of maps with a given initial capacity. It may look simple to do that, but it is in fact a little tricky.



```
var map = HashMap.newHashMap(100);
var set = HashSet.newHashSet(100);
var linkedMap = LinkedHashMap.newLinkedHashMap(100);
var linkedSet = LinkedHashSet.newLinkedHashSet(100);
```

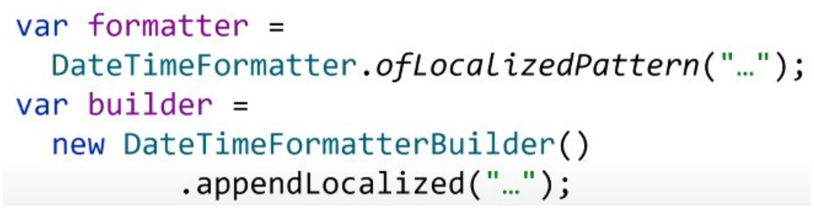
- A *HashMap* for instance, is built on an internal array, and when this array gets full, it is copied in a larger array. The problem is that it's not actually copied, **all the key value pairs of the first array need to be rehashed** in order to be placed at the right position of the new array. Because the hashing depends on the size of the array, there is little chance that the bucket they will be stored is the same from one array to the other. So you cannot actually just copy the content of the first array to the second one.

- In order to avoid collisions, an array is considered full when it reaches 75%. So to create a *HashMap* that can store let's say a hundred key value pairs, you actually need an array of size 134. Under the constraint that the size of the array should be a power of 2, you will actually create an array of size 256. This computation is error prone to say the least, so now with Java 21 you have a factory method, and actually several factory methods to do that, and you can forget about it.

- All you need to do is call newHashMap(), pass the number of entries you need, and that's it. And by the way, you have an equivalent method on HashSet, LinkedHashMap, and LinkedHashSet. Which is much simpler.

- Localization in the Date and Time API

- You now have a *ofLocalizedPattern()* factory method, that takes a format of type String, that produces a localized pattern. The corresponding builder has also been updated with an *appendLocalized()* method, that takes a localized pattern of type String.



```
var formatter =
  DateTimeFormatter.ofLocalizedPattern("…");
var builder =
  new DateTimeFormatterBuilder()
          .appendLocalized("…");
```

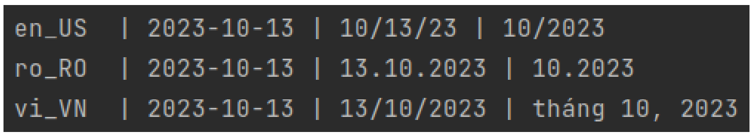
→ Extra data (from outside the video):

- To turn a date-time into a string, you need a *DateTimeFormatter*. Creating custom formatters is easy with the static factors method *ofPattern*. And creating localized formatters is easy, too, as long as you stick with the four predefined *FormatStyles SHORT*, *MEDIUM*, *LONG*, and *FULL*. But what about custom localized formatters? JDK 19 is there for you!.

- It adds the static factory method *DateTimeFormatter.**ofLocalizedPattern*** that accepts Unicode skeletons, which only define what fields you want to include but leave the formatting and order for the localization to determine. The *DateTimeFormatterBuilder* class was also extended with the methods.

```
public static void main( String[] args ) {
   var now = ZonedDateTime.now();
   Stream.of(
      Locale.of("en", "US"),
      Locale.of("ro", "RO"),
      Locale.of("vi", "VN")
   ).forEach(locale -> {
      Locale.setDefault(locale);

      var custom = DateTimeFormatter.ofPattern("y-MM-dd");
      var local = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);

      // it's now possible to create custom localized formatters for specific patterns
      var customLocal = DateTimeFormatter.ofLocalizedPattern("yMM");

      System.out.printf( "%s  | %s | %s | %s %n",
         locale, now.format(custom), now.format(local), now.format(customLocal));
   });
}
```

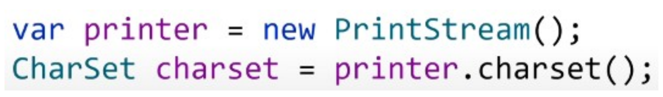
Output:



```
en_US  | 2023-10-13 | 10/13/23 | 10/2023
ro_RO  | 2023-10-13 | 13.10.2023 | 10.2023
vi_VN  | 2023-10-13 | 13/10/2023 | tháng 10, 2023
```

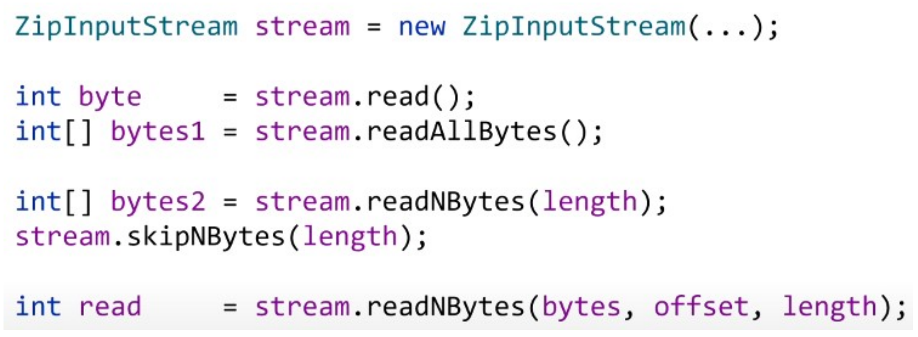
- Additions to the Java I/O API

- Several elements have been added to Java I/O. You can now get the charset that the PrintStream is using to produce its results.



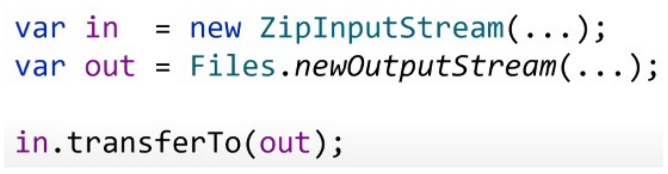
```
var printer = new PrintStream();
CharSet charset = printer.charset();
```

- And you also have several methods that have been added to the *ZipInputStream* class, to read bytes, one by one, and several at a time, or to skip them.



```
ZipInputStream stream = new ZipInputStream(...);

int byte     = stream.read();
int[] bytes1 = stream.readAllBytes();

int[] bytes2 = stream.readNBytes(length);
stream.skipNBytes(length);

int read     = stream.readNBytes(bytes, offset, length);
```

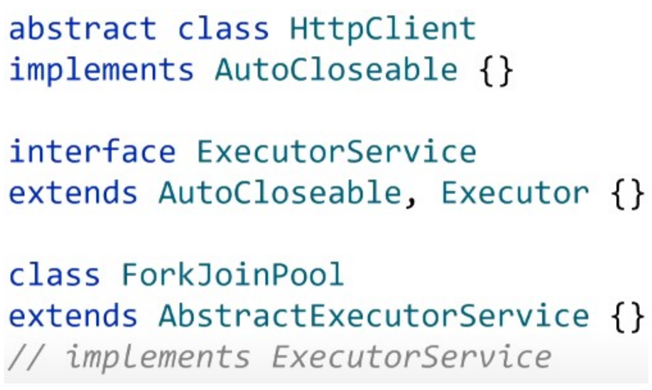
- In *ZipInputStream*, a *tranferTo()* method has been added. This method was added on *InputStream* with Java 9, and it is now also defined in *ZipInputStream*. This method takes an *OutputStream* as a parameter, and copies the elements of the input stream to the output stream. Once this method returns, there is no more element to be read from this input stream.



```
var in  = new ZipInputStream(...);
var out = Files.newOutputStream(...);

in.transferTo(out);
```

- Using AutoCloseable with HttpClient, ExecutorService, and ForkJoinPool

- Several classes from the JDK became auto closeable between 17 and 21. Among them you have HttpClient, ExecutorService, and ForkJoinPool.



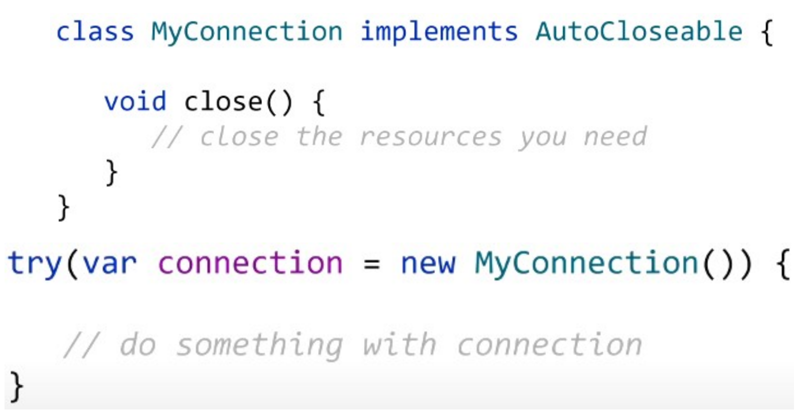
```
abstract class HttpClient
implements AutoCloseable {}

interface ExecutorService
extends AutoCloseable, Executor {}

class ForkJoinPool
extends AbstractExecutorService {}
// implements ExecutorService
```

- *AutoCloseable* is a nifty mechanism that allows you to automatically close a resource once you do not need it anymore.

- *AutoCloseable* is a very simple interface, that has only one method: *close()*.

- Once you have a class that implements this interface, you can use the *try-with-resources* statements to create instances of it. Any class will do, as long as it implements this AutoCloseable interface.

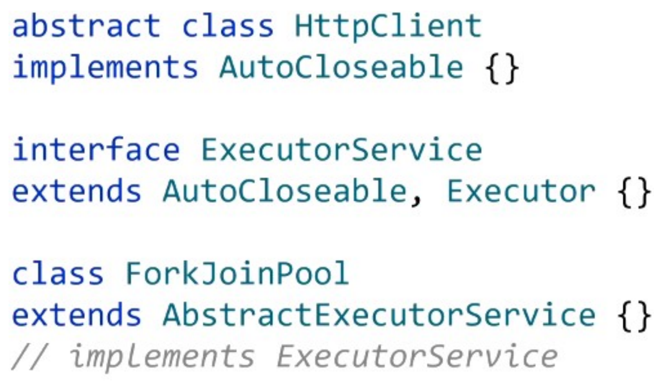
- When your code exits the *try-with-resources* block, you have the guarantee that your *close()* method will be called, no matter what.



```
   class MyConnection implements AutoCloseable {

      void close() {
         // close the resources you need
      }
   }
try(var connection = new MyConnection()) {

   // do something with connection
}
```

- Basically all the resources from Java I/O implement this interface, and the same goes for the elements of JDBC: Connection, Statement, ResultSet, and the like, they are all AutoCloseable. And there are many classes in the JDK that are also AutoCloseable. And, again, any class from your application could be AutoCloseable.

- The good news is that HttpClient, ExecutorService, and ForkJoinPool are joining the family, meaning that you can also use them with this try-with-resources pattern.



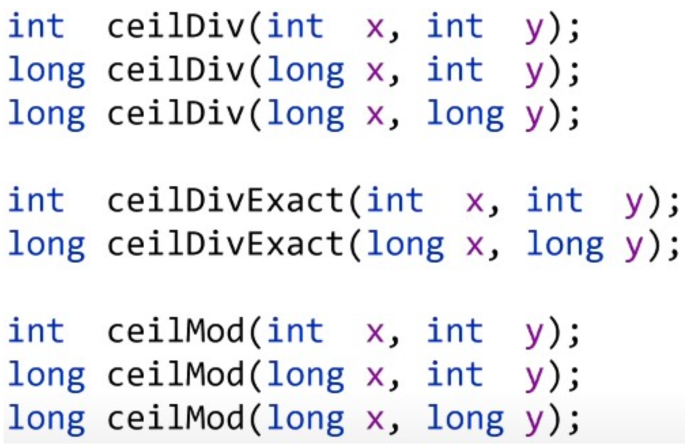
```
abstract class HttpClient
implements AutoCloseable {}

interface ExecutorService
extends AutoCloseable, Executor {}

class ForkJoinPool
extends AbstractExecutorService {}
// implements ExecutorService
```

- Additions to the Math class

- Three overloads of the ceilDiv() method have been added, along with two overloads of ceilDivExact() and two more of ceilMod().

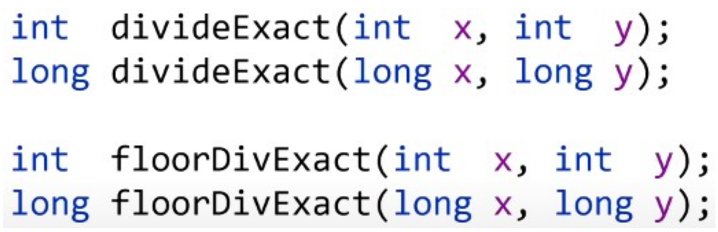
- Dividing integers may sound simple, but it's actually quite complex. You have several ways to handle rounding, made even more complex when you need to support negative numbers, without forgetting about overflow.

- Remember that you have Integer.MAX_VALUE and Integer.MIN_VALUE, which can generate some unexpected overflows. That makes many corner cases that you need to handle properly, thus the amount of overloads for these methods.



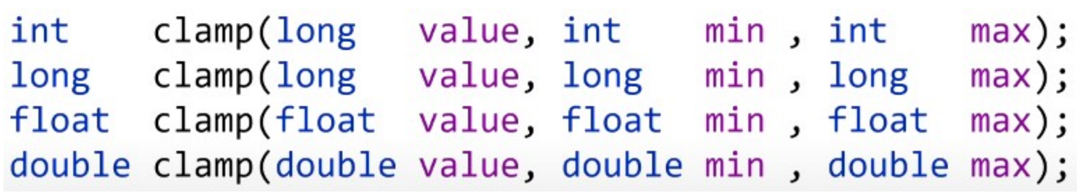
```
int  ceilDiv(int  x, int  y);
long ceilDiv(long x, int  y);
long ceilDiv(long x, long y);

int  ceilDivExact(int  x, int  y);
long ceilDivExact(long x, long y);

int  ceilMod(int  x, int  y);
long ceilMod(long x, int  y);
long ceilMod(long x, long y);
```

- Another four methods added: divideExact() and floorDivideExact(). These four actually throw an ArithmeticException in case of an overflow, instead of silently returning Integer.MIN_VALUE or Long.MIN_VALUE.



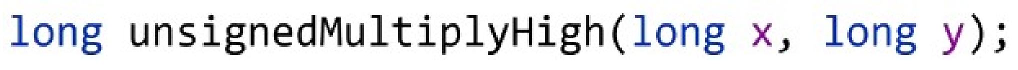
```
int  divideExact(int  x, int  y);
long divideExact(long x, long y);

int  floorDivExact(int  x, int  y);
long floorDivExact(long x, long y);
```

- Another method has been added with several overloads, to handle the int, long, float, and double primitive types: this method is clamp(). This method clamps the provided value between the min and the max you give.



```
int    clamp(long   value, int    min , int    max);
long   clamp(long   value, long   min , long   max);
float  clamp(float  value, float  min , float  max);
double clamp(double value, double min , double max);
```

- As a bonus, you get a last method: unsignedMultiplyHigh(), that multiplies two longs as unsigned numbers, and returns the 64 higher bits as a long.



```
long unsignedMultiplyHigh(long x, long y);
```

- Multiplying BigIntegers in parallel

- The capability to perform multiplications in parallel. This implementation is quite complex and uses the Fork / Join framework.

- As usual when using this kind of things, be careful, measure your performance gains precisely, and in a production environment, to make sure that it fits your exact use case, and that you are actually improving your performances.

```
BigInteger parallelMultiply(BigInteger value);
```

- Additions to the Thread class

- I just would like to cover two elements here. First, you now have a *join()* method and a *sleep()* method on the *Thread* class, that both take a *Duration* as a parameter.

- You also have a *isVirtual()* method, that tells you if this thread is virtual of not. And don't forget that you have a new way to create threads from the *Thread* class. You have two polymorphic builders that you can use to create threads, both platform threads and virtual threads.

```
void sleep(Duration duration);
void join(Duration duration);

boolean isVirtual();
```

- If you want to use virtual threads, you also have a new factory method in the *Executors* class, to create an executor service that does not pool anything, that just creates virtual threads on demand. That's the *Executors.newVirtualThreadPerTaskExecutor()* factory method.

```
Thread.ofPlatform();
Thread.ofVirtual();
```

```
var executor =
   Executors.newVirtualThreadPerTaskExecutor();
```

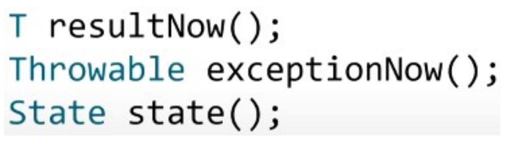
- Additions to the *Future* interface

- About the new methods added on the Future interface, there are actually three of them, resultNow() and exceptionNow(). These two methods do not declare any checked exceptions, so no more try catch in your code, which is a relief. Just be aware that if you call them and the future you are using is not done, then you will get an IllegalStateException.

- You are really supposed to call these methods when you know that your future produced a result, or an exception.

- There is a third one, state(), that you can call to check the current state of your future object. The return type is a new enumeration: Future.State. It has four instances: RUNNING, meaning that your task is not done yet, SUCCESS, meaning that your task completed successfully, FAILED: meaning that your task completed but with an exception, and CANCELLED: meaning that your task was, well, cancelled.

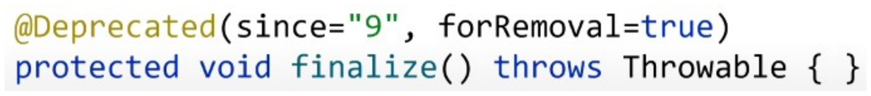
- You can now monitor what is happenning with your tasks more precisely, thanks to this API.



```
T resultNow();
Throwable exceptionNow();
State state();
```

- Deprecation of *finalize()*

- If you have some code in your current finalize() methods in your application, move it somewhere else, because at some point this code will not be executed anymore.

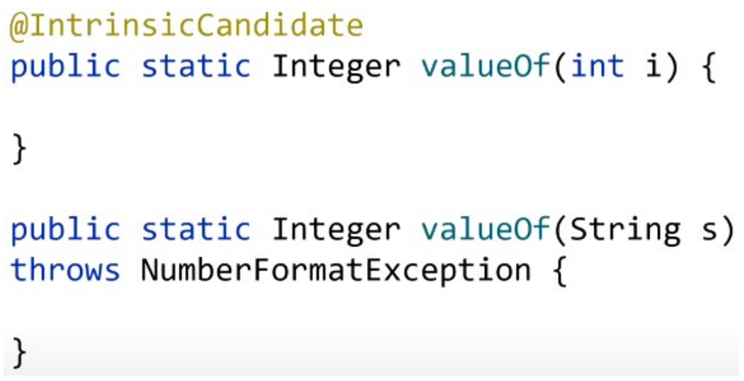


```
@Deprecated(since="9", forRemoval=true)
protected void finalize() throws Throwable { }
```

- Deprecation of the constructors of the wrapper classes

- The second element that is deprecated are the constructors of the wrapper classes: Integer, Long, Float, Double, and the like. This is in preparation of the release of Value Types, an element delivered by the Valhalla project.

- These constructors have been deprecated for removal, and they will be removed in the future. So make sure you are not calling these methods anymore, and if you are, you can just replace these calls with a call to the corresponding valueOf() factory methods, that is basically doing the same thing.



```
@IntrinsicCandidate
public static Integer valueOf(int i) {

}

public static Integer valueOf(String s)
throws NumberFormatException {

}
```