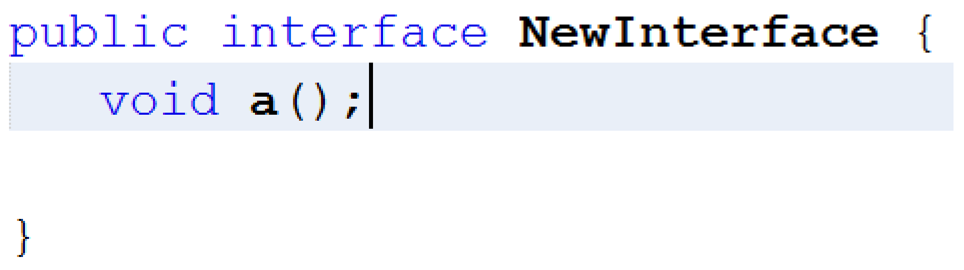
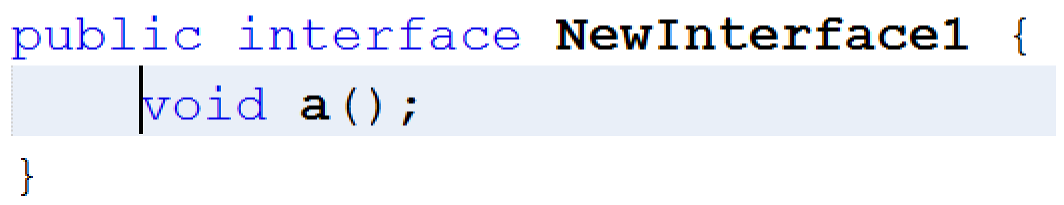
JAVA 8 versija

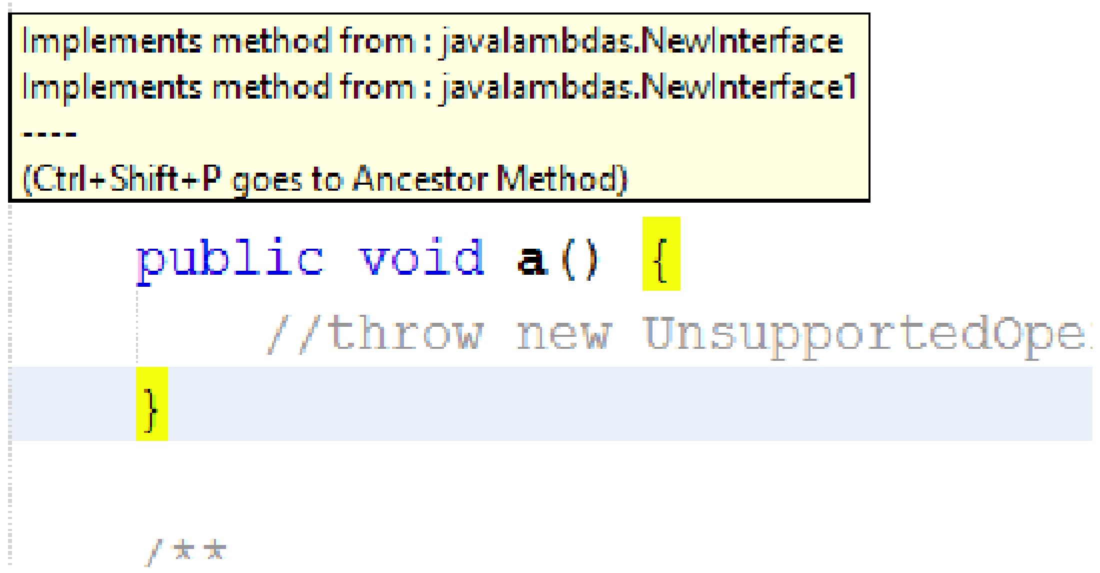
JAVA Interface pavyzdys iki 8 versijos



```java
public interface NewInterface {
    void a();

}
```



```java
public interface NewInterface1 {
     void a();
}
```

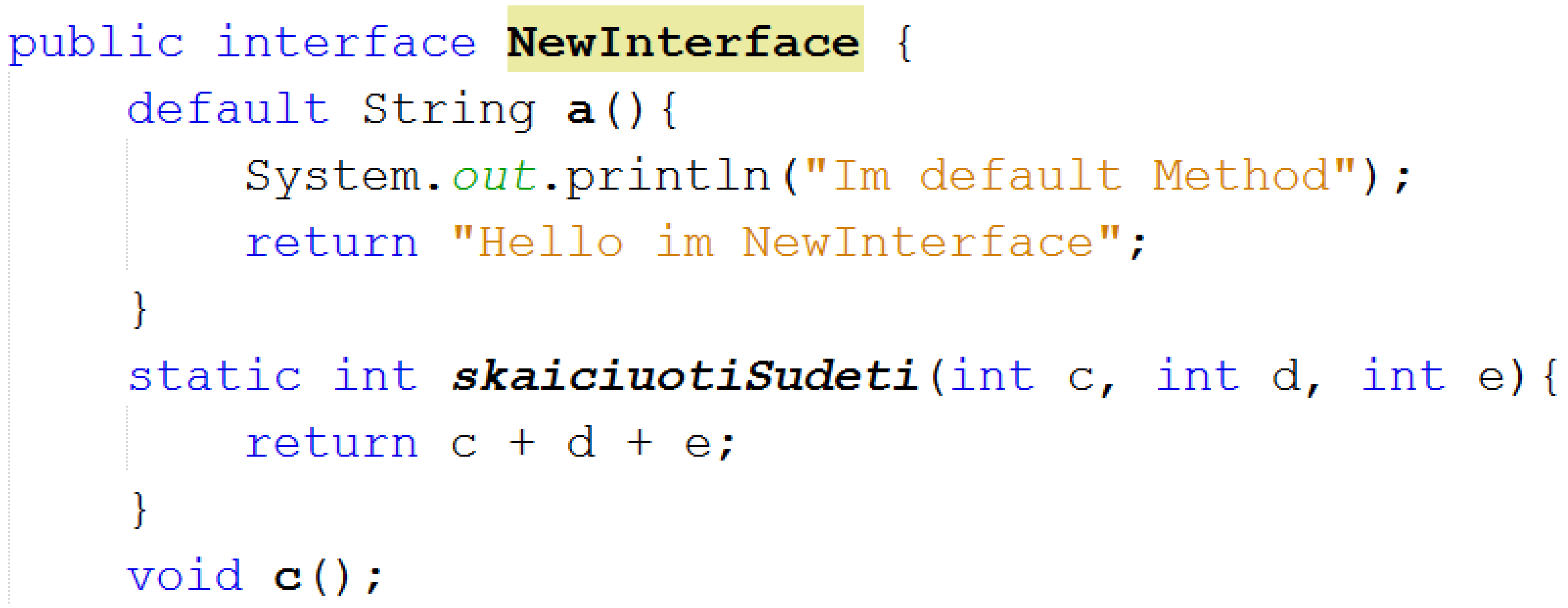


```
Implements method from : javalambdas.NewInterface
Implements method from : javalambdas.NewInterface1
----
(Ctrl+Shift+P goes to Ancestor Method)
```

```java
    public void a() {
        //throw new UnsupportedOpe
    }


    /**
```

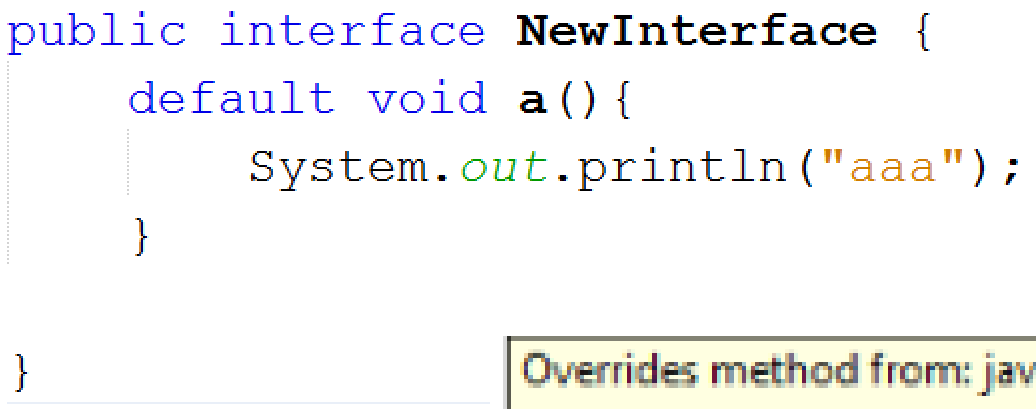
JAVA atnaujino Interface funkcionalumą nuo 8+ versijos

- Įvesti metodai **default** ir **static.**

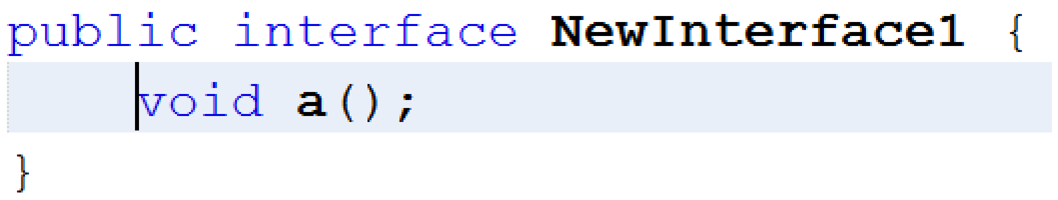


```java
public interface NewInterface {
    default String a(){
        System.out.println("Im default Method");
        return "Hello im NewInterface";
    }
    static int skaiciuotiSudeti(int c, int d, int e){
        return c + d + e;
    }
    void c();
```

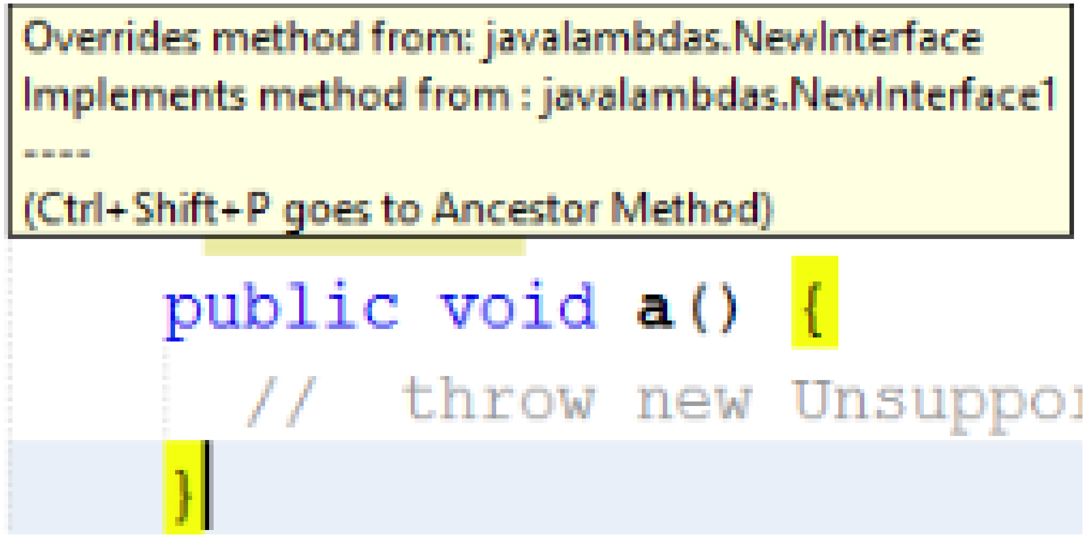
Default metodas kaip ir įprastinis metodas tiktai priekyje yra prirašomas žodis default



```
public interface NewInterface {
    default void a(){
        System.out.println("aaa");
    }

}
```



```
public interface NewInterface1 {
    void a();
}
```



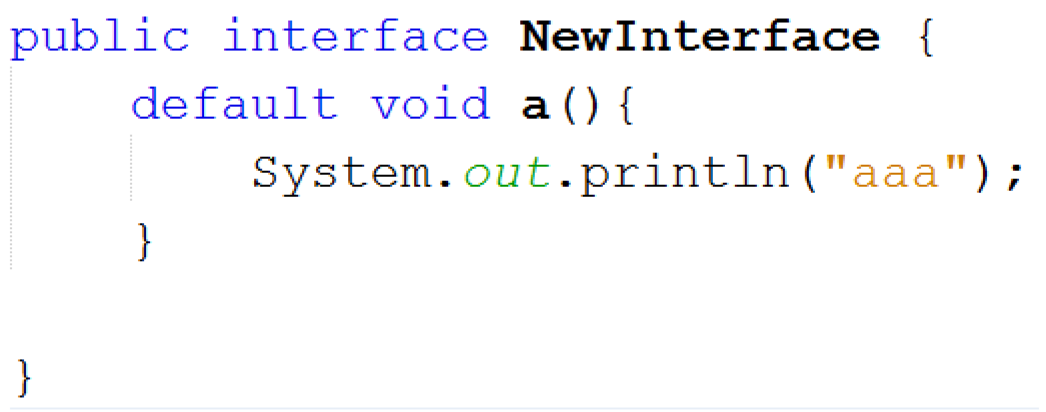
Overrides method from: javalambdas.NewInterface
Implements method from : javalambdas.NewInterface1

(Ctrl+Shift+P goes to Ancestor Method)

```
public void a() {
  //  throw new Unsuppo
}
```

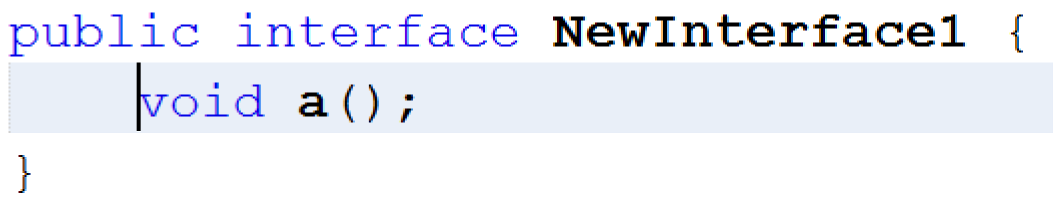
Pagal nutylėjimą(angl. default) metodas vs abstraktus metodas

Pagal nutylėjimą



```
public interface NewInterface {
    default void a(){
        System.out.println("aaa");
    }

}
```

Abstraktus metodas



```
public interface NewInterface1 {
    void a();
}
```

Visi abstraktūs metodai privalo būti implementuoti, o default metodai neprivalo

Lambdos ir funkcinis interfeisas

```
//Before Java 8:
new Thread(new Runnable() {
@Override public void run() {
System.out.println("Before Java8, too much code for too little to
do"); } }).start();
//Java 8 way: new Thread( () -> System.out.println("In Java8,
Lambda expression rocks !!") ).start();
```

Lambdos panaudojimas

- (parametrai) -> išraiška
  (parametrai) -> sakinys
  (parametrai) -> { sakiniai }
- Jeigu nėra atliekami jokie parametro pakeitimai lamba išraiška galima apibrėžti štai taip : `() -> System.out.println("Hello Lambda Expressions");`
- Jeigu metodas yra dviejų parametru: (int even, int odd) -> even + odd
- Patarimas naudoti trumpesnius vardus lambda išraiškose pvz a,b,c arba x,y, negu even arba odd taip supaprastinant kodo skaitomumą.

Event handling using Java 8 Lambda expressions

```
// Before Java 8:
JButton show = new JButton("Show");
show.addActionListener(new ActionListener() {
@Override public void actionPerformed(ActionEvent e) {
            System.out.println("Event handling without lambda
expression is boring"); } });
// Java 8 way: show.addActionListener((e) -> {
System.out.println("Light, Camera, Action !! Lambda
expressions Rocks"); });
```

```
//Prior Java 8 : List features = Arrays.asList("Lambdas", "Default Method",
"Stream API", "Date and Time API");
for (String feature : features) { System.out.println(feature); }
//In Java 8: List features = Arrays.asList("Lambdas", "Default Method",
"Stream API", "Date and Time API");
features.forEach(n -> System.out.println(n));
// Even better use Method reference feature of Java 8 // method reference
is denoted by :: (double colon) operator // looks similar to score resolution
operator of C++ features.forEach(System.out::println);
```

Using Lambda expression and Functional interface Predicate

```
public static void filter(List names, Predicate condition) {
	for(String name: names) {
		if(condition.test(name)) {
			System.out.println(name + " ");
		}
	}
}
```

Filter metodas pritaikius lambda išraiškas

```
public static void filter(List names, Predicate condition) {
names.stream().filter((name) -> (condition.test(name))).forEach((name) -> {
        System.out.println(name + " ");
});
}
```

```
public static void main(args[]){ List languages = Arrays.asList("Java", "Scala", "C++", "Haskell", "Lisp");
System.out.println("Languages which starts with J :");
filter(languages, (str)->str.startsWith("J"));
System.out.println("Languages which ends with a ");
filter(languages, (str)->str.endsWith("a"));
System.out.println("Print all languages :");
filter(languages, (str)->true);
System.out.println("Print no language : ");
filter(languages, (str)->false);
System.out.println("Print language whose length greater than 4:");
filter(languages, (str)->str.length() > 4); }
```

How to combine Predicate in Lambda Expressions

```
/ We can even combine Predicate using and(), or() And xor() logical functions // for
example to find names, which starts with J and four letters long, you // can pass
combination of two Predicate

Predicate<String> startsWithJ = (n) -> n.startsWith("J");
Predicate<String> fourLetterLong = (n) -> n.length() == 4;
names.stream() .filter(startsWithJ.and(fourLetterLong)) .forEach((n) ->
System.out.print("\nName, which starts with 'J' and four letter long is : " + n));
```

Map and Reduce example in Java 8 using lambda expressions (1 of 2)

- This example is about one of the popular functional programming concept called map. It allows you to transform your object. Like in this example we are transforming each element of costBeforeTax list to including Value added Test. We passed a lambda expression x -> x*x to map() method which applies this to all elements of the stream. After that we use forEach() to print the all elements of list. You can actually get a List of all cost with tax by using Stream API's Collectors class. It has methods liketoList() which will combine result of map or any other operation. Since Collector perform terminal operator on Stream, you can't re-use Stream after that. You can even use reduce() method from Stream API to reduce all numbers into one, which we will see in next example

Map and Reduce example in Java 8 using lambda expressions (2 of 2)

```
// applying 12% VAT on each purchase
// Without lambda expressions:
List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
for (Integer cost : costBeforeTax) {
double price = cost + .12*cost;
System.out.println(price); }
// With Lambda expression:
List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
costBeforeTax.stream().map((cost) -> cost + .12*cost).forEach(System.out::println);
```

Output

```
112.0
224.0
336.0
448.0
560.0
112.0
224.0
336.0
448.0
560.0
```

Map Reduce example using Lambda Expressions in Java 8 (1 of 2)

- In previous example, we have seen how map can transform each element of a Collection class e.g. List. There is another function called reduce() which can combine all values into one. Map and Reduce operations are core of functional programming, reduce is also known as fold operation because of its folding nature. By the way reduce is not a new operation, you might have been already using it. If you can recall SQL aggregate functions like sum(), avg() or count(), they are actually reduce operation because they accept multiple values and return a single value. Stream API defines reduce() function which can accept a lambda expression, and combine all values. Stream classes like IntStream has built-in methods like average(), count(), sum() to perform reduce operations and mapToLong(), mapToDouble() methods for transformations. It doesn't limit you, you can either use built-in reduce function or can define yours. In this Java 8 Map Reduce example, we are first applying 12% VAT on all prices and then calculating total of that by using reduce() method.

Map Reduce example using Lambda Expressions in Java 8 (2 of 2)

```
// Applying 12% VAT on each purchase
// Old way:
List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
double total = 0;
for (Integer cost : costBeforeTax) {
            double price = cost + .12*cost;
            total = total + price;
} System.out.println("Total : " + total);
// New way: List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
double bill = costBeforeTax.stream().map((cost) -> cost + .12*cost).reduce((sum, cost) ->
sum + cost).get(); System.out.println("Total : " + bill);
```

Output

```
Total :
1680.0
Total :
1680.0
```

Creating a List of String by filtering (1 of 2)

- Filtering is one of the common operation Java developers perform with large collections, and you will be surprise how much easy it is now to filter bulk data/large collection using lambda expression and stream API. Stream provides a filter() method, which accepts a Predicate object, means you can pass lambda expression to this method as filtering logic. Following examples of filtering collection in Java with lambda expression will make it easy to understand.

Creating a List of String by filtering (2 of 2)

```
// Create a List with String more than 2 characters

List<String> filtered = strList.stream().filter(x -> x.length()> 2).collect(Collectors.toList());

System.out.printf("Original List : %s, filtered list : %s %n", strList, filtered);

Output : Original List : [abc, , bcd, , defg, jk], filtered list : [abc, bcd, defg]
```

Applying function on Each element of List (1 of 2)

- We often need to apply certain function to each element of List e.g. multiplying each element by certain number or dividing it, or doing anything with that. Those operations are perfectly suited for map() method, you can supply your transformation logic to map()method as lambda expression and it will transform each element of that collection, as shown in below example.

Applying function on Each element of List (2 of 2)

```
// Convert String to Uppercase and join them using coma List<String> G7
= Arrays.asList("USA", "Japan", "France", "Germany", "Italy",
"U.K.","Canada"); String G7Countries = G7.stream().map(x ->
x.toUpperCase()).collect(Collectors.joining(", "));
System.out.println(G7Countries); Output : USA, JAPAN, FRANCE,
GERMANY, ITALY, U.K., CANADA
```

Creating a Sub List by Copying distinct values

```
// Create List of square of all distinct numbers
List<Integer> numbers = Arrays.asList(9, 10, 3, 4, 7, 3, 4);
List<Integer> distinct = numbers.stream().map( i ->
i*i).distinct().collect(Collectors.toList());
System.out.printf("Original List : %s, Square Without duplicates : %s %n",
numbers, distinct);
Output : Original List : [9, 10, 3, 4, 7, 3, 4],
Square Without duplicates : [81, 100, 9, 16, 49]
```

Calculating Maximum, Minimum, Sum and Average of List elements

- There is a very useful method called summaryStattics() in stream classes like IntStream, LongStream and DoubleStream. Which returns returns an IntSummaryStatistics, LongSummaryStatistics or DoubleSummaryStatistics describing various summary data about the elements of this stream. In following example, we have used this method to calculate maximum and minimum number in a List. It also has getSum() and getAverage() which can give sum and average of all numbers from List.

IntSummaryStatistics

```
//Get count, min, max, sum, and average for numbers
List<Integer> primes = Arrays.asList(2, 3, 5, 7, 11, 13, 17, 19, 23, 29);
IntSummaryStatistics stats = primes.stream().mapToInt((x) -> x).summaryStatistics();
System.out.println("Highest prime number in List : " + stats.getMax());
System.out.println("Lowest prime number in List : " + stats.getMin());
System.out.println("Sum of all prime numbers : " + stats.getSum());
System.out.println("Average of all prime numbers : " + stats.getAverage());
```

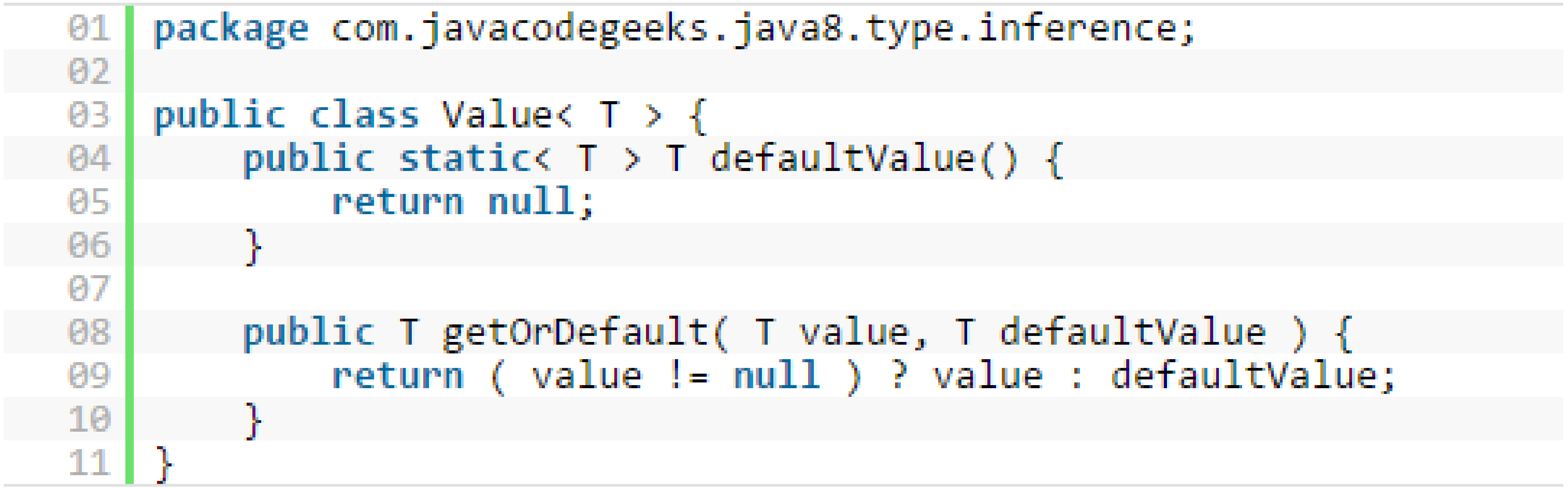
Output :

```
Highest prime number in List : 29 Lowest prime number in List : 2 Sum of all
prime numbers : 129 Average of all prime numbers : 12.9
```

Lambda Expression vs Anonymous class

- Since lambda expression is effectively going to replace Anonymous inner class in new Java code, its important to do a comparative analysis of both of them. One key difference between using Anonymous class and Lambda expression is the use of this keyword. For anonymous class 'this' keyword resolves to *anonymous class*, whereas for lambda expression 'this' keyword resolves to*enclosing class* where lambda is written. Another difference between lambda expression and anonymous class is in the way these two are compiled. Java compiler compiles lambda expressions and convert them into private method of the class. It uses invokedynamic byte code instruction from Java 7 to bind this method dynamically.

Better Type Inference



```java
package com.javacodegeeks.java8.type.inference;

public class Value< T > {
    public static< T > T defaultValue() {
        return null;
    }

    public T getOrDefault( T value, T defaultValue ) {
        return ( value != null ) ? value : defaultValue;
    }
}
```

And here is the usage of **Value< String >** type.

```java
package com.javacodegeeks.java8.type.inference;

public class TypeInference {
    public static void main(String[] args) {
        final Value< String > value = new Value<>();
        value.getOrDefault( "22", Value.defaultValue() );
    }
}
```

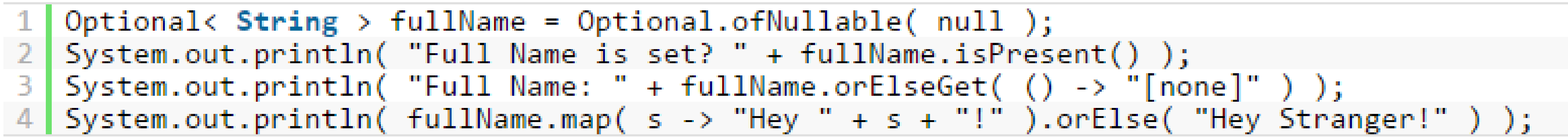
The type parameter of **Value.*defaultValue*()**is inferred and is not required to be provided. In Java 7, the same example will not compile and should be rewritten to **Value.< String >*defaultValue*()**.

JAVA 8 naujos bibliotekos ir atnaujinimai

- **Optional**
- **Streams**
- **Date/Time API (JSR 310)**
- **Nashorn JavaScript engine**
- **Base64**
- **Parallel Arrays**
- **Concurrency**

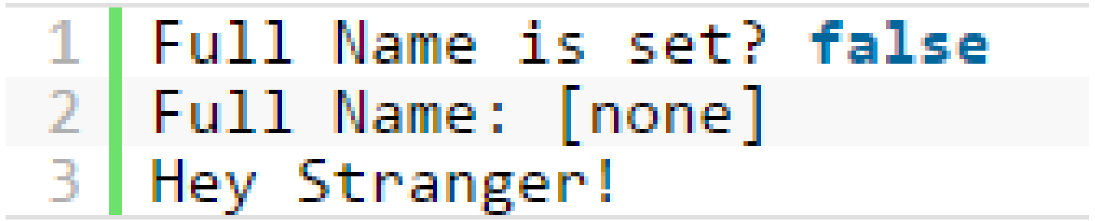
Optional

- The famous **NullPointerException** is by far the most popular cause of Java application failures. Long time ago the great Google Guava project introduced the **Optional**s as a solution to **NullPointerException**s, discouraging codebase pollution with **null** checks and encouraging developers to write cleaner code. Inspired by Google Guava, the **Optional** is now a part of Java 8 library.
- **Optional** is just a container: it can hold a **value** of some type **T** or just be **null**. It provides a lot of useful methods so the explicit **null** checks have no excuse anymore. Please refer to official Java 8 documentation for more details.
- We are going to take a look on two small examples of **Optional** usages: with the **nullable** value and with the value which does not allow **null**s.



```java
Optional< String > fullName = Optional.ofNullable( null );
System.out.println( "Full Name is set? " + fullName.isPresent() );
System.out.println( "Full Name: " + fullName.orElseGet( () -> "[none]" ) );
System.out.println( fullName.map( s -> "Hey " + s + "!" ).orElse( "Hey Stranger!" ) );
```

- The **isPresent()** method returns **true** if this instance of **Optional** has non-null value and **false** otherwise. The **orElseGet()** method provides the fallback mechanism in case **Optional** has **null** value by accepting the function to generate the default one. The **map()** method transforms the current **Optional**'s value and returns the new **Optional** instance. The **orElse()** method is similar to **orElseGet()** but instead of function it accepts the default value. Here is the output of this program:



```
Full Name is set? false
Full Name: [none]
Hey Stranger!
```

- Let us briefly look on another example:

```java
Optional< String > firstName = Optional.of( "Tom" );
System.out.println( "First Name is set? " + firstName.isPresent() );
System.out.println( "First Name: " + firstName.orElseGet( () -> "[none]" ) );
System.out.println( firstName.map( s -> "Hey " + s + "!" ).orElse( "Hey Stranger!" ) );
System.out.println();
```

And here is the output:

```
First Name is set? true
First Name: Tom
Hey Tom!
```

Streams

- The newly added Stream API (**java.util.stream**) introduces real-world functional-style programming into the Java. This is by far the most comprehensive addition to Java library intended to make Java developers significantly more productive by allowing them to write effective, clean, and concise code.
- Stream API makes collections processing greatly simplified (but it is not limited to Java collections only as we will see later). Let us take start off with simple class called Task.

```java
public class Streams  {
    private enum Status {
        OPEN, CLOSED
    };

    private static final class Task {
        private final Status status;
        private final Integer points;

        Task( final Status status, final Integer points ) {
            this.status = status;
            this.points = points;
        }

        public Integer getPoints() {
            return points;
        }

        public Status getStatus() {
            return status;
        }

        @Override
        public String toString() {
            return String.format( "[%s, %d]", status, points );
        }
    }
}
```

Task has some notion of points (or pseudo-complexity) and can be either OPEN or CLOSED. And then let us introduce a small collection of tasks to play with.

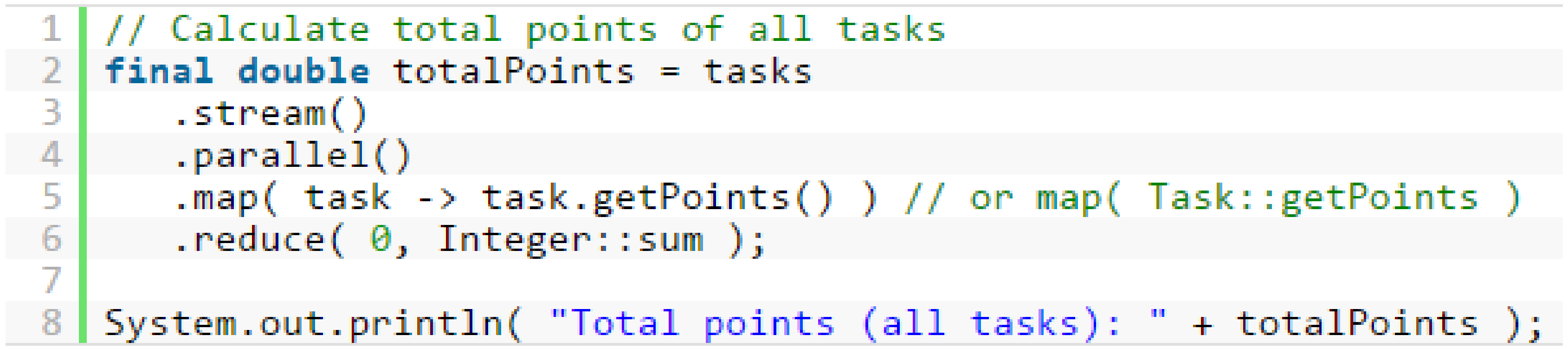
```java
final Collection< Task > tasks = Arrays.asList(
    new Task( Status.OPEN, 5 ),
    new Task( Status.OPEN, 13 ),
    new Task( Status.CLOSED, 8 )
);
```

The first question we are going to address is how many points in total all OPEN tasks have? Up to Java 8, the usual solution for it would be some sort of foreach iteration. But in Java 8 the answers is streams: a sequence of elements supporting sequential and parallel aggregate operations.

```java
// Calculate total points of all active tasks using sum()
final long totalPointsOfOpenTasks = tasks
    .stream()
    .filter( task -> task.getStatus() == Status.OPEN )
    .mapToInt( Task::getPoints )
    .sum();

System.out.println( "Total points: " + totalPointsOfOpenTasks );
```

And the output on the console looks like that:
Total points: **18**

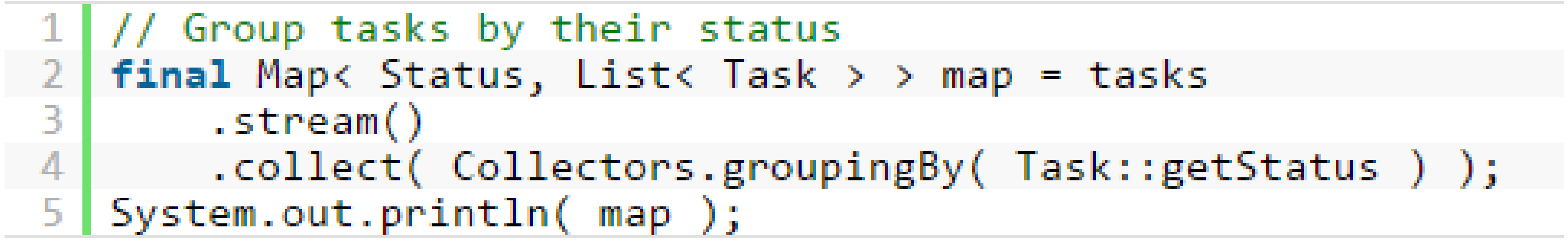
- There are a couple of things going on here. Firstly, the tasks collection is converted to its stream representation. Then, the **filter**operation on stream filters out all ***CLOSED*** tasks. On next step, the **mapToInt** operation converts the stream of **Task**s to the stream of**Integer**s using **Task::getPoints** method of the each task instance. And lastly, all points are summed up using **sum** method, producing the final result.
- Before moving on to the next examples, there are some notes to keep in mind about streams (more details here). Stream operations are divided into intermediate and terminal operations.
- Intermediate operations return a new stream. They are always lazy, executing an intermediate operation such as **filter** does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate
- Terminal operations, such as **forEach** or **sum**, may traverse the stream to produce a result or a side-effect. After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used. In almost all cases, terminal operations are eager, completing their traversal of the underlying data source.
- Yet another value proposition of the streams is out-of-the box support of parallel processing. Let us take a look on this example, which does sums the points of all the tasks.



```java
// Calculate total points of all tasks
final double totalPoints = tasks
    .stream()
    .parallel()
    .map( task -> task.getPoints() ) // or map( Task::getPoints )
    .reduce( 0, Integer::sum );

System.out.println( "Total points (all tasks): " + totalPoints );
```

It is very similar to the first example except the fact that we try to process all the tasks in parallel and calculate the final result using reduce method.
Here is the console output: **Total points (all tasks):** 26.0

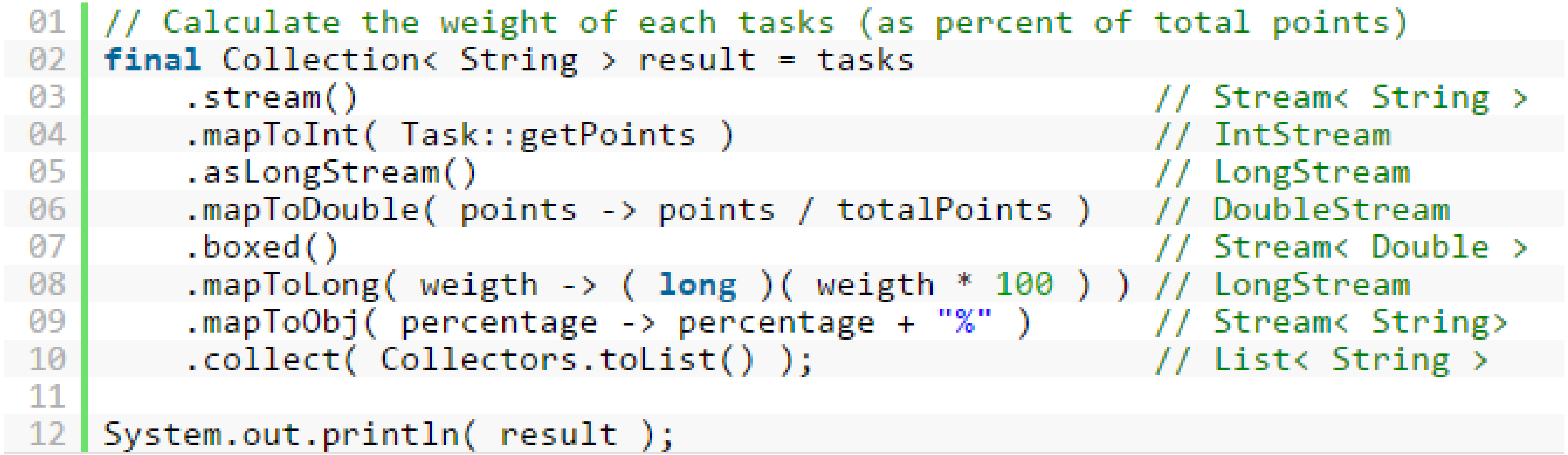
Often, there is a need to performing a grouping of the collection elements by some criteria. Streams can help with that as well as an example below demonstrates.



```java
// Group tasks by their status
final Map< Status, List< Task > > map = tasks
    .stream()
    .collect( Collectors.groupingBy( Task::getStatus ) );
System.out.println( map );
```

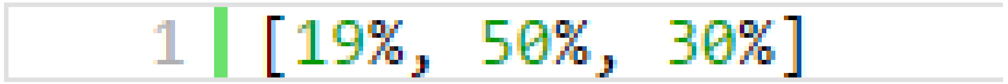
The console output of this example looks like that:
{CLOSED=[[CLOSED, 8]], OPEN=[[OPEN, 5], [OPEN, 13]]}

- To finish up with the tasks example, let us calculate the overall percentage (or weight) of each task across the whole collection, based on its points.



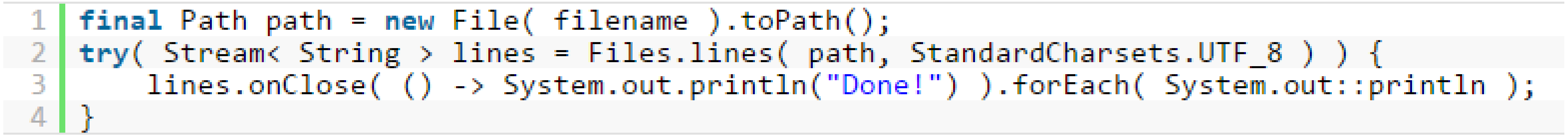
```java
// Calculate the weight of each tasks (as percent of total points)
final Collection< String > result = tasks
    .stream()                                        // Stream< String >
    .mapToInt( Task::getPoints )                     // IntStream
    .asLongStream()                                  // LongStream
    .mapToDouble( points -> points / totalPoints )   // DoubleStream
    .boxed()                                         // Stream< Double >
    .mapToLong( weigth -> ( long )( weigth * 100 ) ) // LongStream
    .mapToObj( percentage -> percentage + "%" )      // Stream< String>
    .collect( Collectors.toList() );                 // List< String >

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

The console output is just here:



```
[19%, 50%, 30%]
```

- And lastly, as we mentioned before, the Stream API is not only about Java collections. The typical I/O operations like reading the text file line by line is a very good candidate to benefit from stream processing. Here is a small example to confirm that.



```java
final Path path = new File( filename ).toPath();
try( Stream< String > lines = Files.lines( path, StandardCharsets.UTF_8 ) ) {
    lines.onClose( () -> System.out.println("Done!") ).forEach( System.out::println );
}
```

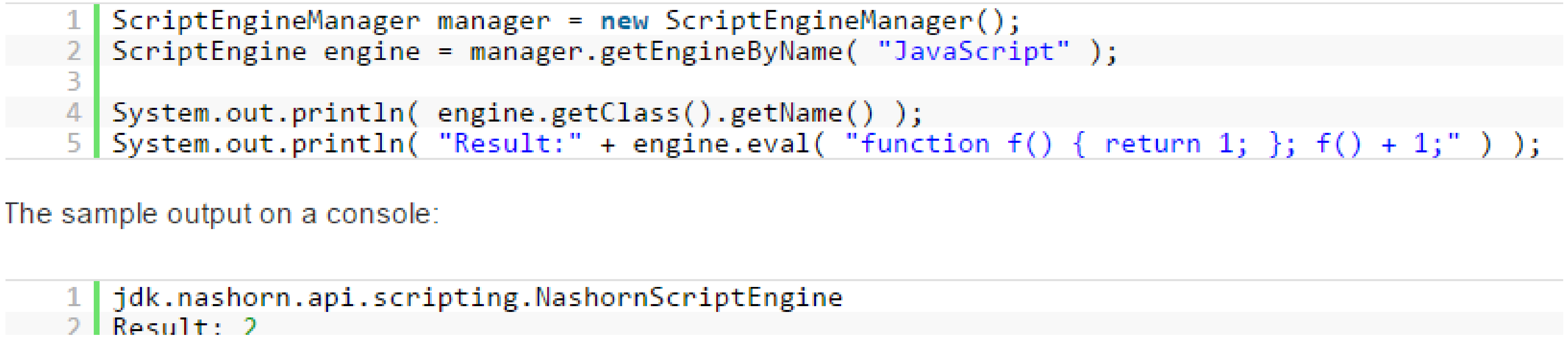
The **onClose** method called on the stream returns an equivalent stream with an additional close handler. Close handlers are run when the **close()** method is called on the stream.
Stream API together with Lambdas and Method References baked by Interface's Default and Static Methods is the Java 8 response to the modern paradigms in software development. For more details, please refer to official documentation.

Date/Time API (JSR 310)

- Java 8 makes one more take on date and time management by delivering New Date-Time API (JSR 310). Date and time manipulation is being one of the worst pain points for Java developers. The standard **java.util.Date** followed by **java.util.Calendar** hasn’t improved the situation at all (arguably, made it even more confusing).
- That is how Joda-Time was born: the great alternative date/time API for Java. The Java 8’s New Date-Time API (JSR 310) was heavily influenced by Joda-Time and took the best of it. The new **java.time** package contains all the classes for date, time, date/time, time zones, instants, duration, and clocks manipulation. In the design of the API the immutability has been taken into account very seriously: no change allowed (the tough lesson learnt from **java.util.Calendar**). If the modification is required, the new instance of respective class will be returned.
- Let us take a look on key classes and examples of their usages. The first class is **Clock** which provides access to the current instant, date and time using a time-zone. **Clock** can be used instead of **System.currentTimeMillis()** and **TimeZone.getDefault()**.

Nashorn JavaScript engine

- Java 8 comes with new Nashorn JavaScript engine which allows developing and running certain kinds of JavaScript applications on JVM. Nashorn JavaScript engine is just another implementation of javax.script.ScriptEngine and follows the same set of rules, permitting Java and JavaScript interoperability. Here is a small example.



```java
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName( "JavaScript" );

System.out.println( engine.getClass().getName() );
System.out.println( "Result:" + engine.eval( "function f() { return 1; }; f() + 1;" ) );
```

The sample output on a console:

```
jdk.nashorn.api.scripting.NashornScriptEngine
Result: 2
```

Base64

```java
import java.nio.charset.StandardCharsets;
import java.util.Base64;

public class Base64s {
    public static void main(String[] args) {
        final String text = "Base64 finally in Java 8!";

        final String encoded = Base64
            .getEncoder()
            .encodeToString( text.getBytes( StandardCharsets.UTF_8 ) );
        System.out.println( encoded );

        final String decoded = new String(
            Base64.getDecoder().decode( encoded ),
            StandardCharsets.UTF_8 );
        System.out.println( decoded );
    }
}
```

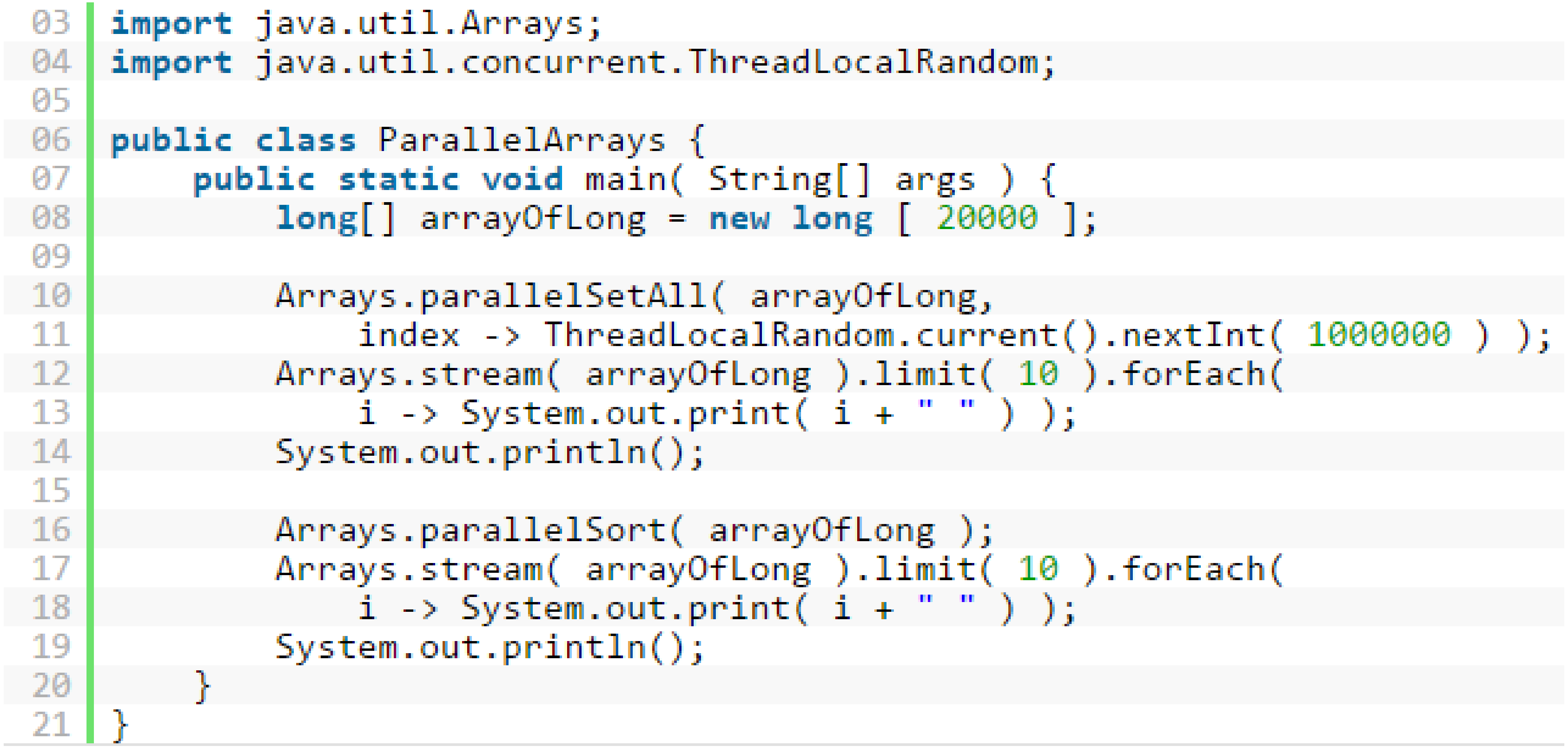
The console output from program run shows both encoded and decoded text:

```
QmFzZTY0IGZpbmFsbHkgaW4gSmF2YSA4IQ==
Base64 finally in Java 8!
```

Parallel Arrays

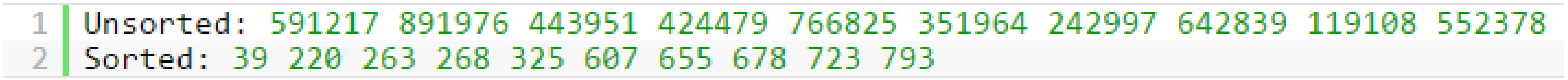
- Java 8 release adds a lot of new methods to allow parallel arrays processing. Arguably, the most important one is **parallelSort()** which may significantly speedup the sorting on multicore machines. The following small example demonstrates this new method family (**parallelXxx**) in action.

Parallel Arrays



```java
import java.util.Arrays;
import java.util.concurrent.ThreadLocalRandom;

public class ParallelArrays {
    public static void main( String[] args ) {
        long[] arrayOfLong = new long [ 20000 ];

        Arrays.parallelSetAll( arrayOfLong,
            index -> ThreadLocalRandom.current().nextInt( 1000000 ) );
        Arrays.stream( arrayOfLong ).limit( 10 ).forEach(
            i -> System.out.print( i + " " ) );
        System.out.println();

        Arrays.parallelSort( arrayOfLong );
        Arrays.stream( arrayOfLong ).limit( 10 ).forEach(
            i -> System.out.print( i + " " ) );
        System.out.println();
    }
}
```

- This small code snippet uses method ***parallelSetAll()*** to fill up arrays with 20000 random values. After that, the ***parallelSort()*** is being applied. The program outputs first 10 elements before and after sorting so to ensure the array is really ordered. The sample program output may look like that (please notice that array elements are randomly generated):



```
Unsorted: 591217 891976 443951 424479 766825 351964 242997 642839 119108 552378
Sorted: 39 220 263 268 325 607 655 678 723 793
```

Concurrency

- New methods have been added to the **java.util.concurrent.ConcurrentHashMap** class to support aggregate operations based on the newly added streams facility and lambda expressions. Also, new methods have been added to the **java.util.concurrent.ForkJoinPool**class to support a common pool (check also our free course on Java concurrency).
- The new **java.util.concurrent.locks.StampedLock** class has been added to provide a capability-based lock with three modes for controlling read/write access (it might be considered as better alternative for infamous **java.util.concurrent.locks.ReadWriteLock**).
- New classes have been added to the **java.util.concurrent.atomic** package:

DoubleAccumulator, **DoubleAdder**, **LongAccumulator**, **LongAdder.**