Pageno:
Chapter 1. Java 8: why should you care?

1.6. Summary

Following are the key concepts you should take away from this chapter:

- Keep in mind the idea of language ecosystem and the consequent evolve-or-wither pressure on languages. Although Java may be supremely healthy at the moment, you can recall other healthy languages such as COBOL that failed to evolve.
- The core additions to Java 8 provide exciting new concepts and functionality to ease the writing of programs that are both effective and concise.
- Multicore processors aren't fully served by existing Java programming practice.
- Functions are first-class values; remember how methods can be passed as functional values and how anonymous functions (lambdas) are written.
- The Java 8 concept of Streams generalizes many aspects of Collections but both enables more readable code and allows elements of a stream to be processed in parallel.
- You can use a default method in an interface to provide a method body if an implementing class chooses not to do so.
- Other interesting ideas from functional programming include dealing with null and using pattern matching.

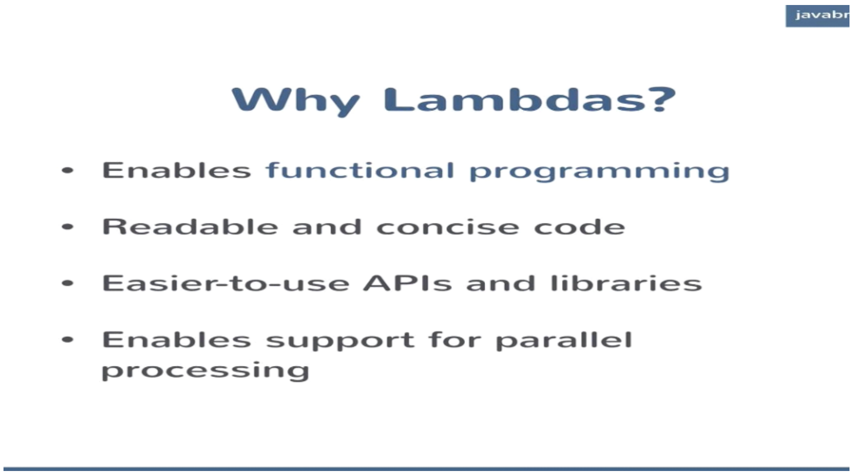
Chapter 2. Passing code with behavior Parameterization

2.5. Summary

Following are the key concepts you should take away from this chapter:

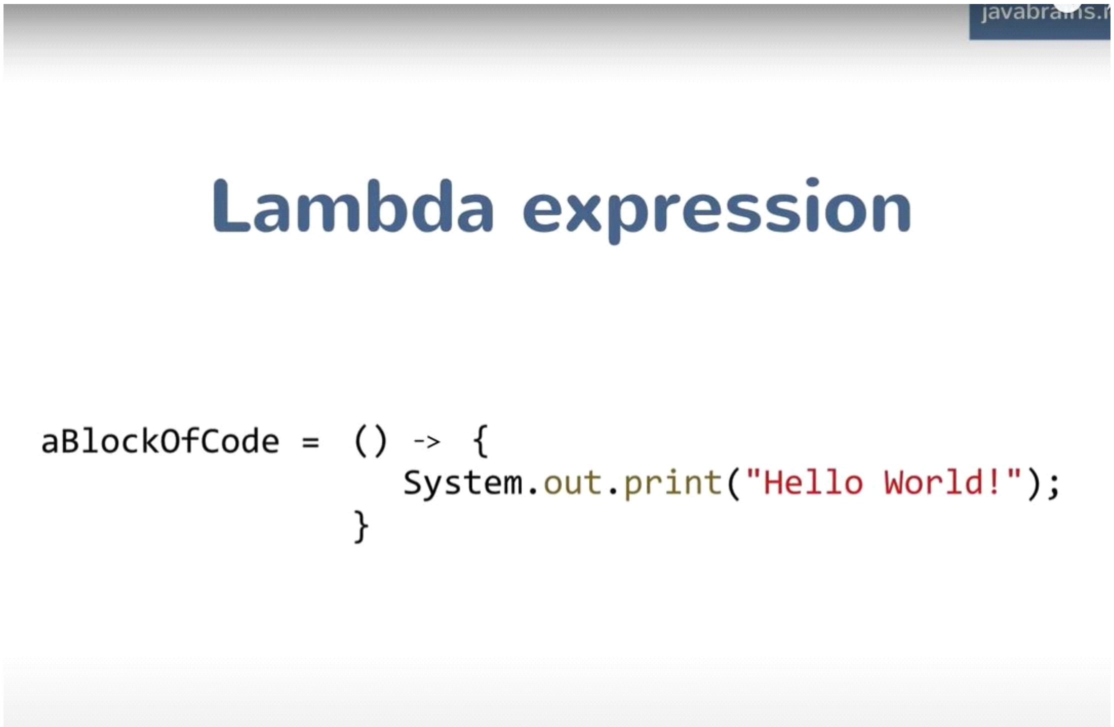
- Behavior parameterization is the ability for a method to *take* multiple different behaviors as parameters and use them internally to *accomplish* different behaviors.
- Behavior parameterization lets you make your code more adaptive to changing requirements and saves on engineering efforts in the future.
- Passing code is a way to give new behaviors as arguments to a method. But it's verbose prior to Java 8. Anonymous classes helped a bit before Java 8 to get rid of the verbosity associated with declaring multiple concrete classes for an interface that are needed only once.
- The Java API contains many methods that can be parameterized with different behaviors, which include sorting, threads, and GUI handling.

Chapter 3. Lambda expressions



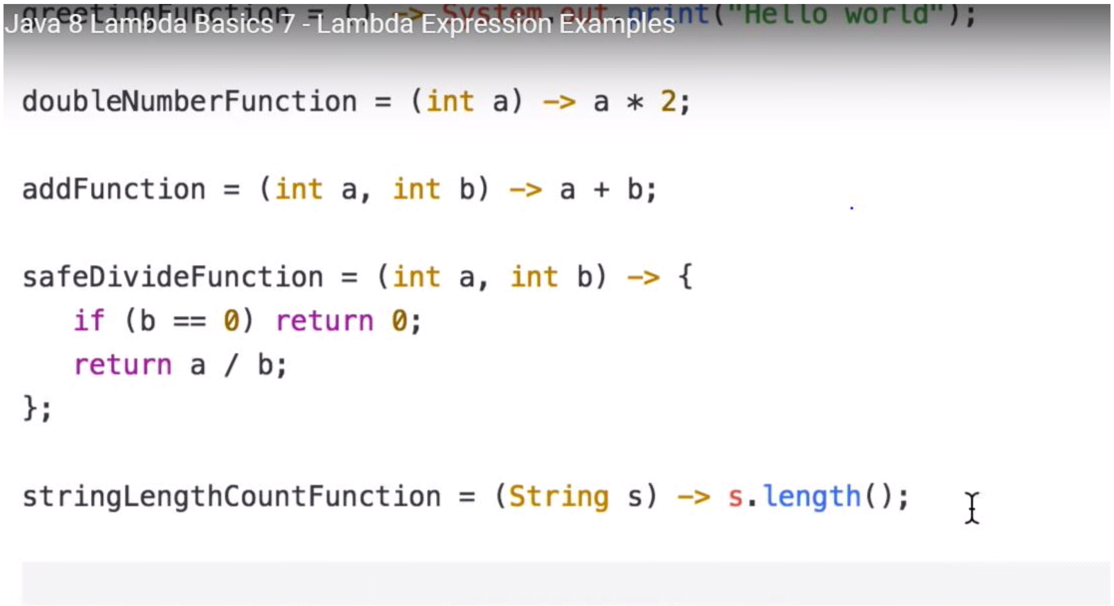
Why Lambdas?

- Enables functional programming
- Readable and concise code
- Easier-to-use APIs and libraries
- Enables support for parallel processing

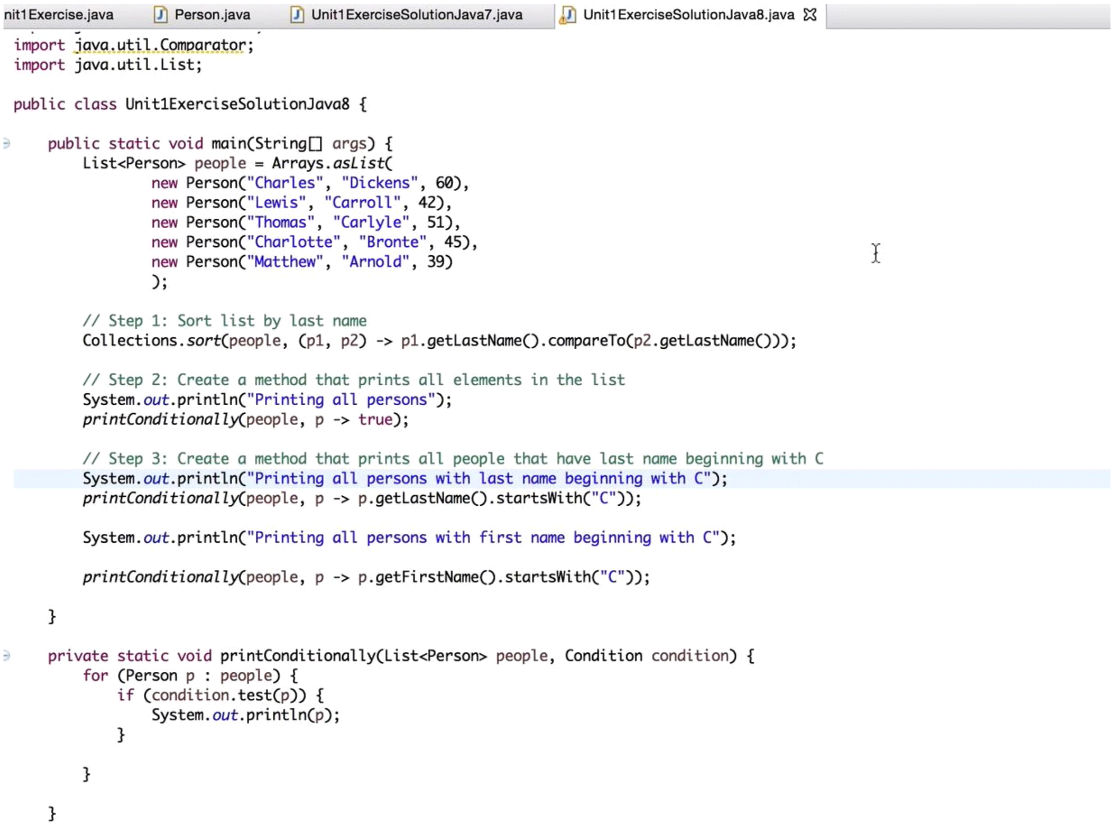


Lambda expression

```
aBlockOfCode = () -> {
                   System.out.print("Hello World!");
               }
```

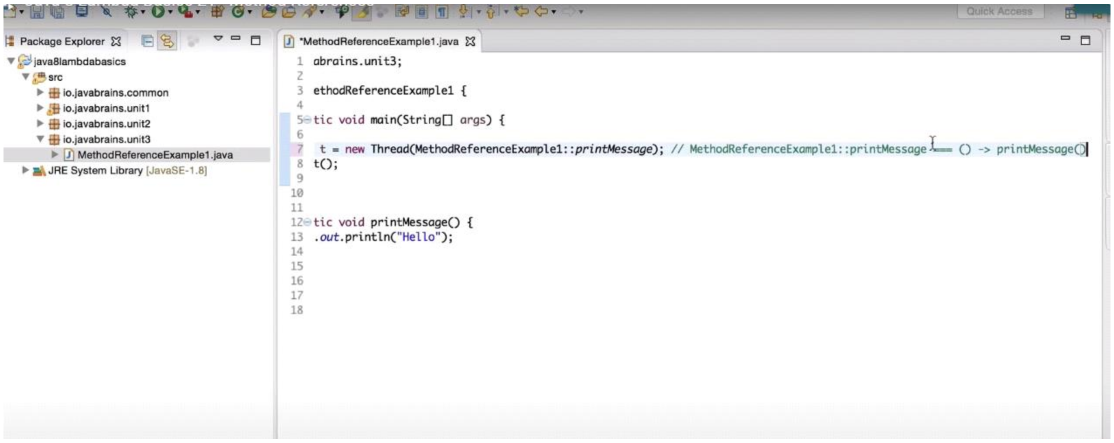


```
greetingFunction = () -> System.out.print("Hello world");

doubleNumberFunction = (int a) -> a * 2;

addFunction = (int a, int b) -> a + b;

safeDivideFunction = (int a, int b) -> {
   if (b == 0) return 0;
   return a / b;
};

stringLengthCountFunction = (String s) -> s.length();
```

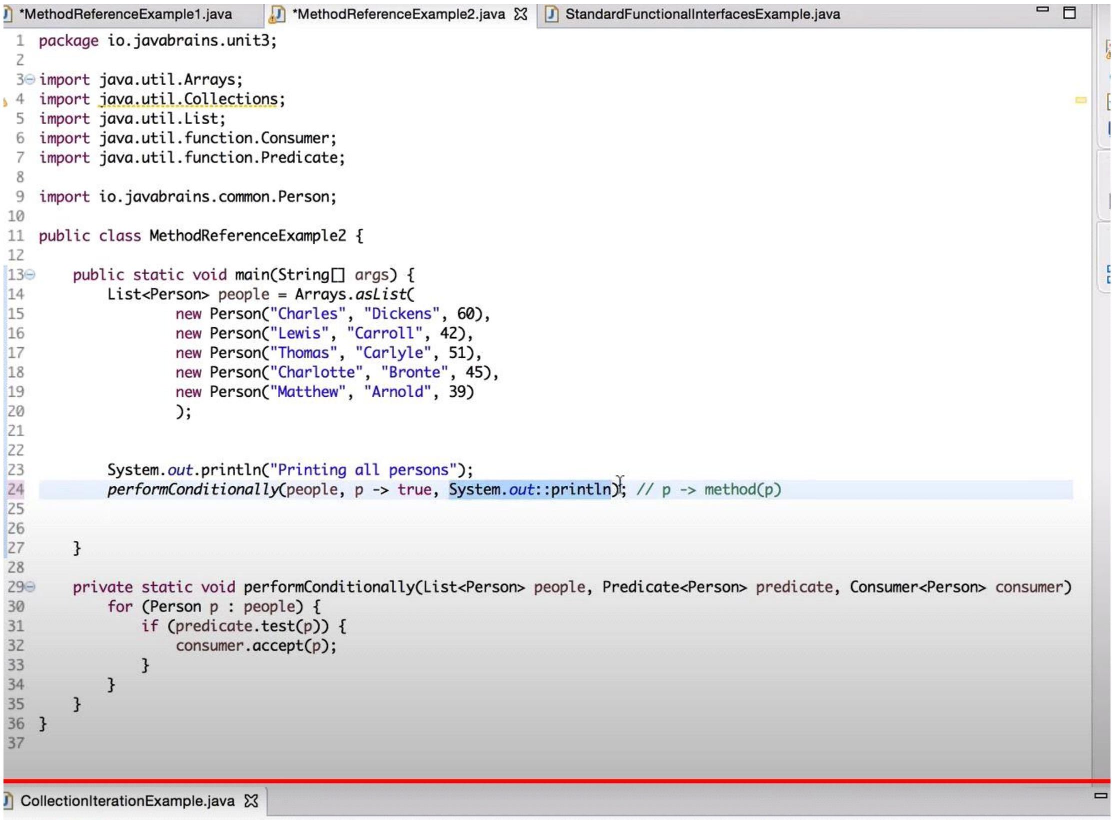


nit1Exercise.java
Person.java
Unit1ExerciseSolutionJava7.java
Unit1ExerciseSolutionJava8.java

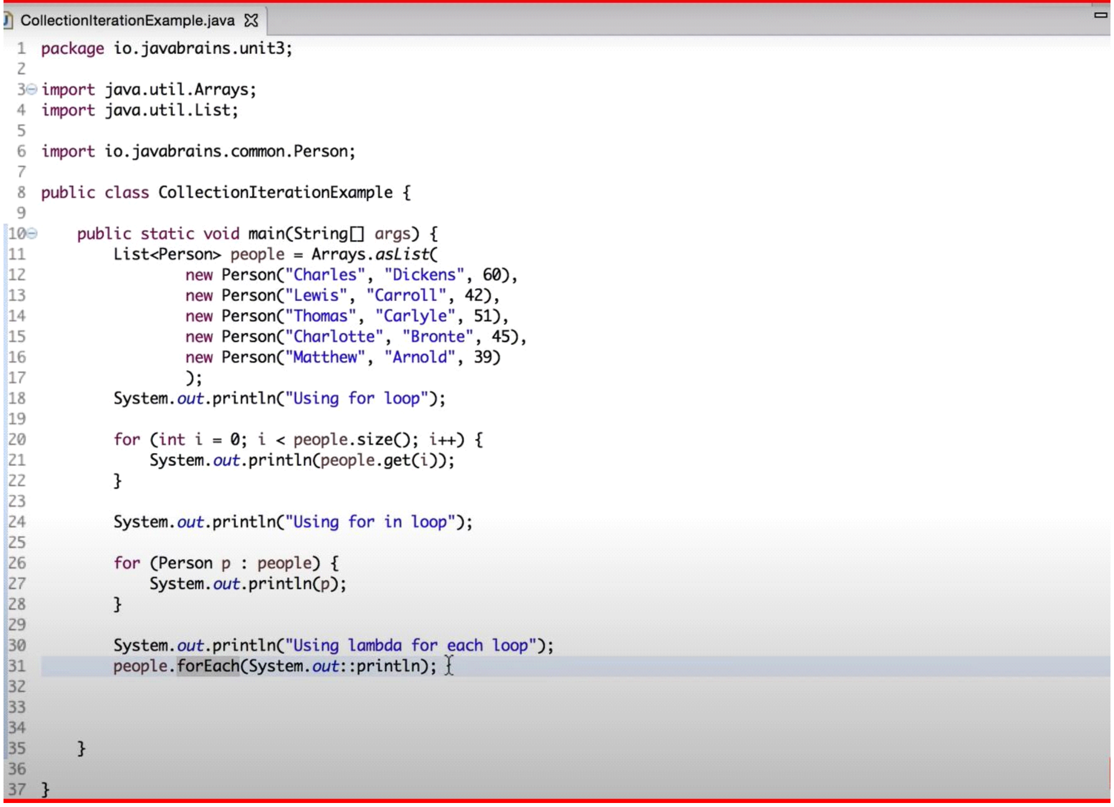
```java
import java.util.Comparator;
import java.util.List;

public class Unit1ExerciseSolutionJava8 {

    public static void main(String[] args) {
        List<Person> people = Arrays.asList(
                new Person("Charles", "Dickens", 60),
                new Person("Lewis", "Carroll", 42),
                new Person("Thomas", "Carlyle", 51),
                new Person("Charlotte", "Bronte", 45),
                new Person("Matthew", "Arnold", 39)
                );

        // Step 1: Sort list by last name
        Collections.sort(people, (p1, p2) -> p1.getLastName().compareTo(p2.getLastName()));

        // Step 2: Create a method that prints all elements in the list
        System.out.println("Printing all persons");
        printConditionally(people, p -> true);

        // Step 3: Create a method that prints all people that have last name beginning with C
        System.out.println("Printing all persons with last name beginning with C");
        printConditionally(people, p -> p.getLastName().startsWith("C"));

        System.out.println("Printing all persons with first name beginning with C");

        printConditionally(people, p -> p.getFirstName().startsWith("C"));

    }

    private static void printConditionally(List<Person> people, Condition condition) {
        for (Person p : people) {
            if (condition.test(p)) {
                System.out.println(p);
            }

        }

    }
```



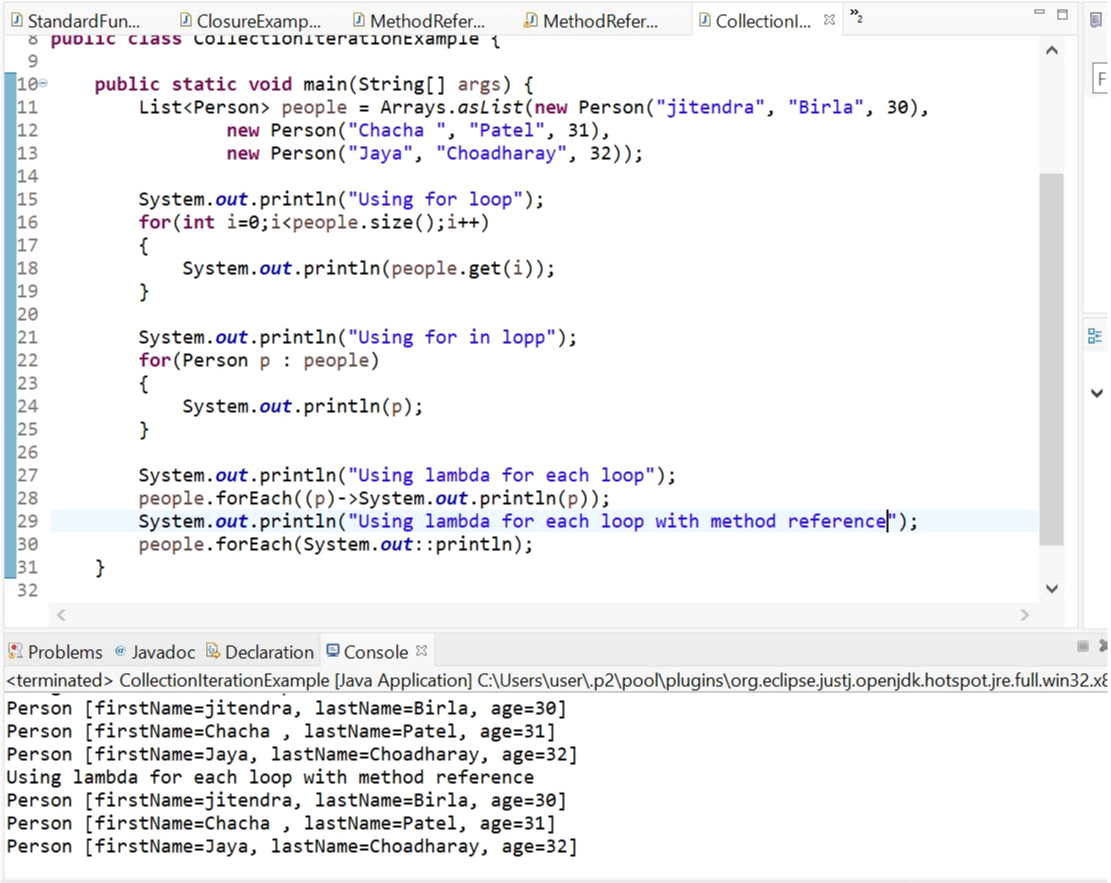
```java
abrains.unit3;

ethodReferenceExample1 {

tic void main(String[] args) {

 t = new Thread(MethodReferenceExample1::printMessage); // MethodReferenceExample1::printMessage == () -> printMessage()
t();



tic void printMessage() {
.out.println("Hello");
```


*MethodReferenceExample1.java
*MethodReferenceExample2.java
StandardFunctionalInterfacesExample.java

```java
package io.javabrains.unit3;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Predicate;

import io.javabrains.common.Person;

public class MethodReferenceExample2 {

    public static void main(String[] args) {
        List<Person> people = Arrays.asList(
                new Person("Charles", "Dickens", 60),
                new Person("Lewis", "Carroll", 42),
                new Person("Thomas", "Carlyle", 51),
                new Person("Charlotte", "Bronte", 45),
                new Person("Matthew", "Arnold", 39)
                );


        System.out.println("Printing all persons");
        performConditionally(people, p -> true, System.out::println); // p -> method(p)


    }

    private static void performConditionally(List<Person> people, Predicate<Person> predicate, Consumer<Person> consumer)
        for (Person p : people) {
            if (predicate.test(p)) {
                consumer.accept(p);
            }
        }
    }
}

```

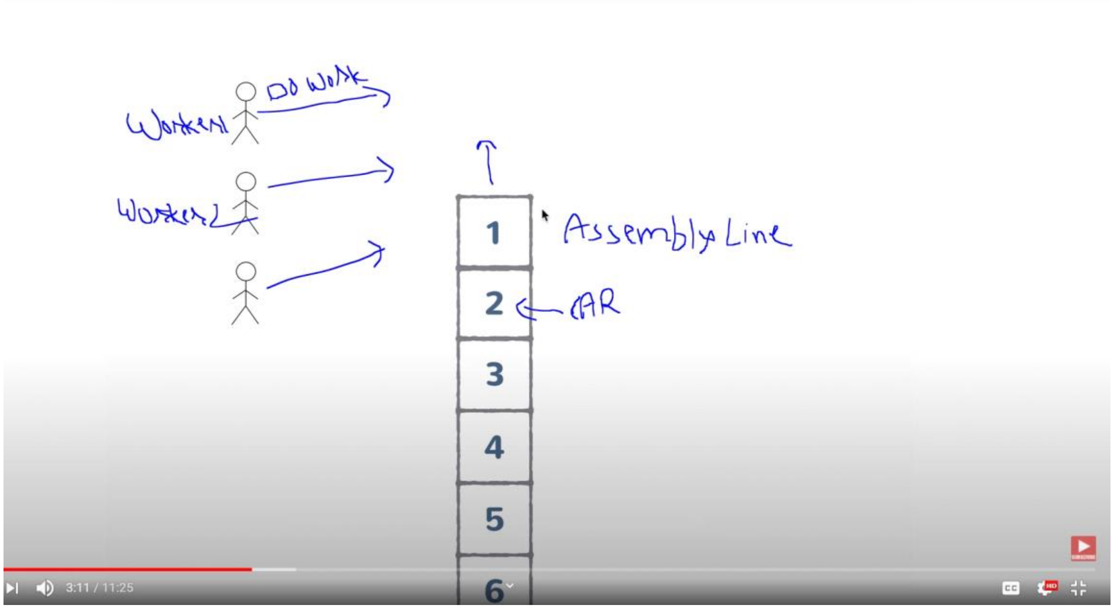


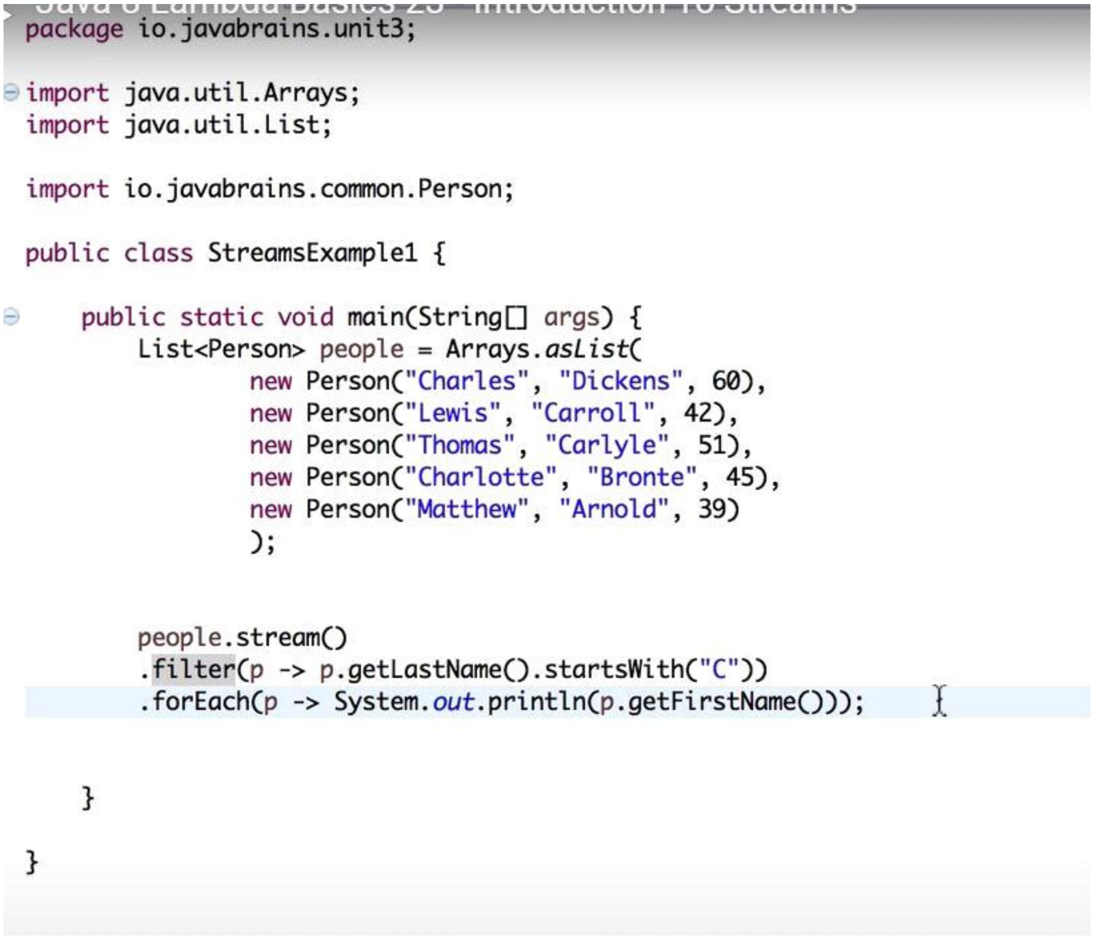
```java
package io.javabrains.unit3;

import java.util.Arrays;
import java.util.List;

import io.javabrains.common.Person;

public class CollectionIterationExample {

    public static void main(String[] args) {
        List<Person> people = Arrays.asList(
                new Person("Charles", "Dickens", 60),
                new Person("Lewis", "Carroll", 42),
                new Person("Thomas", "Carlyle", 51),
                new Person("Charlotte", "Bronte", 45),
                new Person("Matthew", "Arnold", 39)
                );
        System.out.println("Using for loop");

        for (int i = 0; i < people.size(); i++) {
            System.out.println(people.get(i));
        }

        System.out.println("Using for in loop");

        for (Person p : people) {
            System.out.println(p);
        }

        System.out.println("Using lambda for each loop");
        people.forEach(System.out::println);



    }

}
```



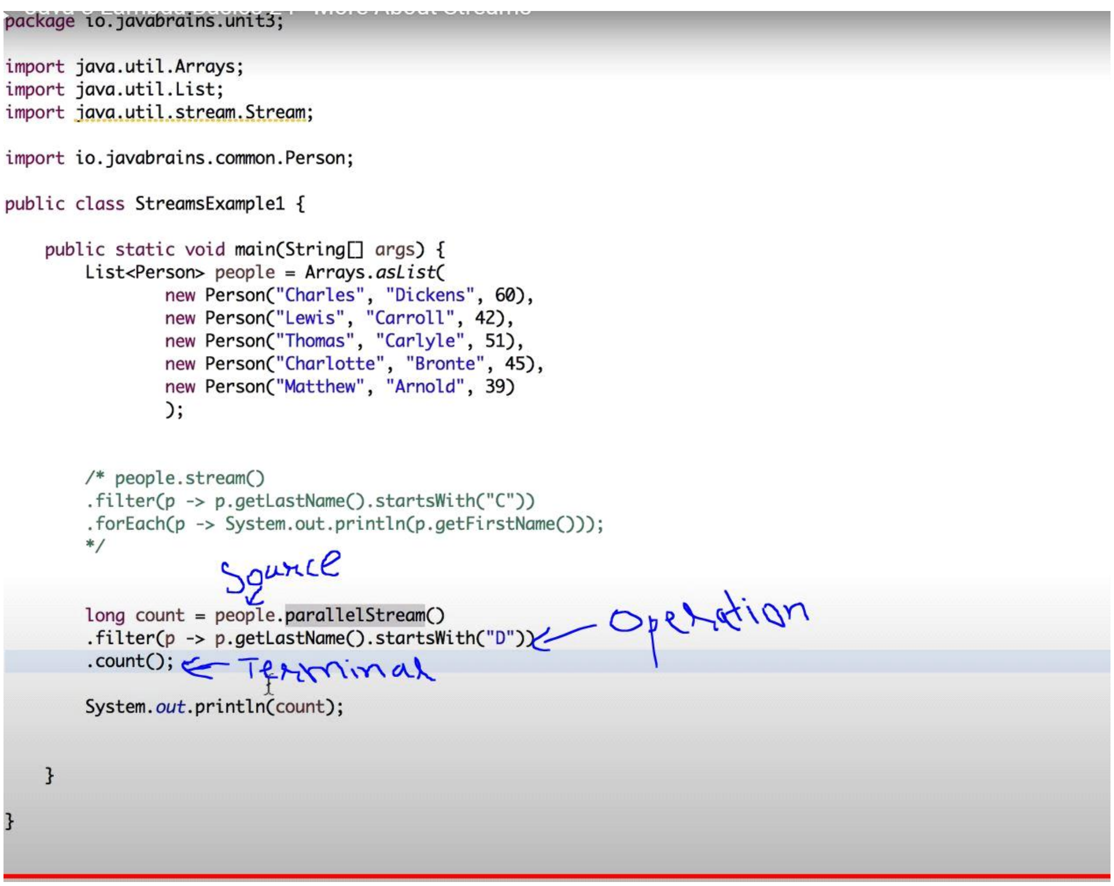
```java
public class CollectionIterationExample {

    public static void main(String[] args) {
        List<Person> people = Arrays.asList(new Person("jitendra", "Birla", 30),
                new Person("Chacha ", "Patel", 31),
                new Person("Jaya", "Choadharay", 32));

        System.out.println("Using for loop");
        for(int i=0;i<people.size();i++)
        {
            System.out.println(people.get(i));
        }

        System.out.println("Using for in lopp");
        for(Person p : people)
        {
            System.out.println(p);
        }

        System.out.println("Using lambda for each loop");
        people.forEach((p)->System.out.println(p));
        System.out.println("Using lambda for each loop with method reference");
        people.forEach(System.out::println);
    }

```

```
<terminated> CollectionIterationExample [Java Application] C:\Users\user\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x8
Person [firstName=jitendra, lastName=Birla, age=30]
Person [firstName=Chacha , lastName=Patel, age=31]
Person [firstName=Jaya, lastName=Choadharay, age=32]
Using lambda for each loop with method reference
Person [firstName=jitendra, lastName=Birla, age=30]
Person [firstName=Chacha , lastName=Patel, age=31]
Person [firstName=Jaya, lastName=Choadharay, age=32]
```





```
package io.javabrains.unit3;

import java.util.Arrays;
import java.util.List;

import io.javabrains.common.Person;

public class StreamsExample1 {

    public static void main(String[] args) {
        List<Person> people = Arrays.asList(
                new Person("Charles", "Dickens", 60),
                new Person("Lewis", "Carroll", 42),
                new Person("Thomas", "Carlyle", 51),
                new Person("Charlotte", "Bronte", 45),
                new Person("Matthew", "Arnold", 39)
                );


        people.stream()
        .filter(p -> p.getLastName().startsWith("C"))
        .forEach(p -> System.out.println(p.getFirstName()));


    }

}
```



```
package io.javabrains.unit3;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

import io.javabrains.common.Person;

public class StreamsExample1 {

    public static void main(String[] args) {
        List<Person> people = Arrays.asList(
                new Person("Charles", "Dickens", 60),
                new Person("Lewis", "Carroll", 42),
                new Person("Thomas", "Carlyle", 51),
                new Person("Charlotte", "Bronte", 45),
                new Person("Matthew", "Arnold", 39)
                );


        /* people.stream()
        .filter(p -> p.getLastName().startsWith("C"))
        .forEach(p -> System.out.println(p.getFirstName()));
        */


        long count = people.parallelStream()
        .filter(p -> p.getLastName().startsWith("D"))
        .count();

        System.out.println(count);


    }

}
```

Table 3.2. Common functional interfaces in Java 8

Functional interface	Function descriptor	Primitive specializations
Predicate<T>	T -> boolean	IntPredicate, LongPredicate, DoublePredicate
Consumer<T>	T -> void	IntConsumer, LongConsumer, DoubleConsumer
Function<T, R>	T -> R	IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T>
Supplier<T>	() -> T	BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier
UnaryOperator<T>	T -> T	IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator
BinaryOperator<T>	(T, T) -> T	IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator
BiPredicate<L, R>	(L, R) -> boolean	
BiConsumer<T, U>	(T, U) -> void	ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>
BiFunction<T, U, R>	(T, U) -> R	ToIntBiFunction<T, U>, ToLongBiFunction<T, U>, ToDoubleBiFunction<T, U>

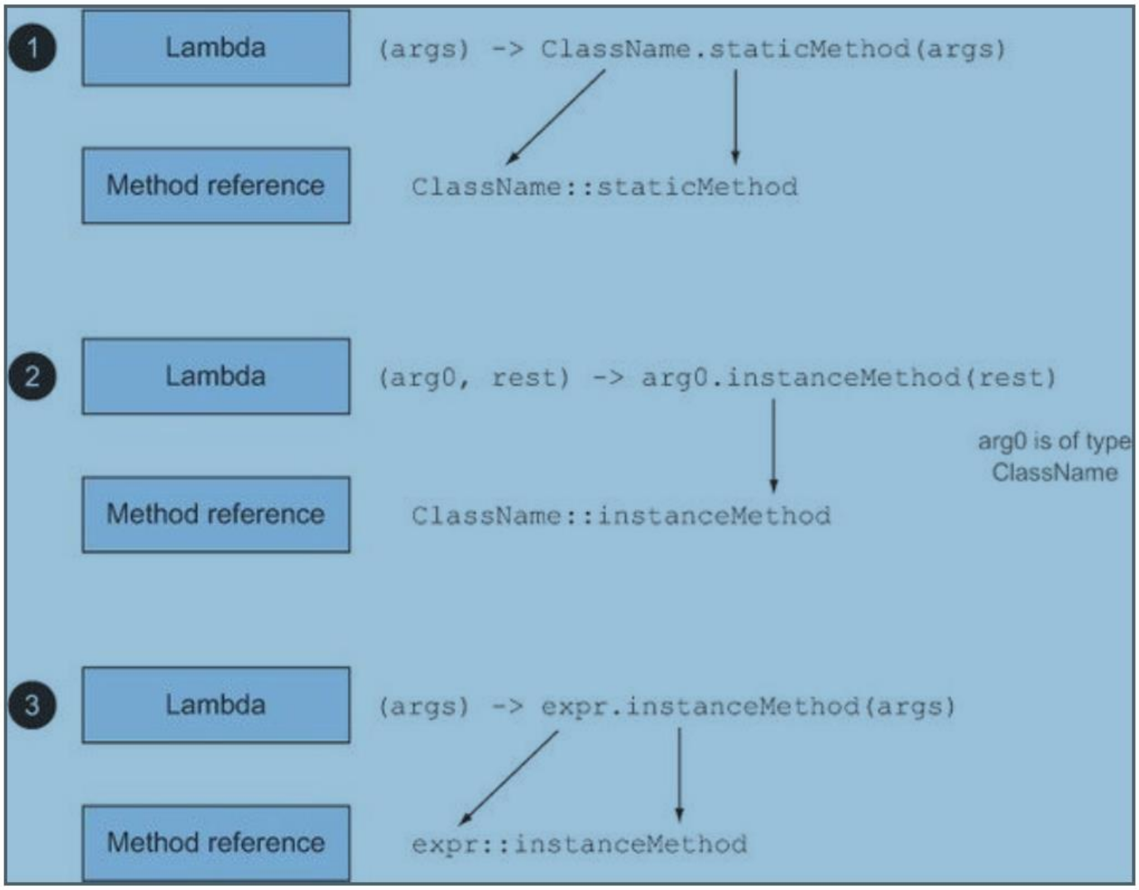
Table 3.3. Examples of lambdas with functional interfaces

Use case	Example of lambda	Matching functional interface
A boolean expression	(List<String> list) -> list.isEmpty()	Predicate<List<String>>
Creating objects	() -> new Apple(10)	Supplier<Apple>
Consuming from an object	(Apple a) -> System.out.println(a.getWeight())	Consumer<Apple>
Select/extract from an object	(String s) -> s.length()	Function<String, Integer> or ToIntFunction<String>
Combine two values	(int a, int b) -> a * b	IntBinaryOperator
Compare two objects	(Apple a1, Apple a2) -> a1.getWeight().compareTo (a2.getWeight())	Comparator<Apple> or BiFunction<Apple, Apple, Integer> or ToIntBiFunction<Apple, Apple>

Table 3.4. Examples of lambdas and method reference equivalents

Lambda	Method reference equivalent
(Apple a) -> a.getWeight()	Apple::getWeight
() -> Thread.currentThread().dumpStack()	Thread.currentThread()::dumpStack
(str, i) -> str.substring(i)	String::substring
(String s) -> System.out.println(s)	System.out::println

You can think of method references as syntactic sugar for lambdas that refer only to a single method because you write less to express the same thing.


1
Lambda
(args) -> ClassName.staticMethod(args)
Method reference
ClassName::staticMethod
2
Lambda
(arg0, rest) -> arg0.instanceMethod(rest)
arg0 is of type ClassName
Method reference
ClassName::instanceMethod
3
Lambda
(args) -> expr.instanceMethod(args)
Method reference
expr::instanceMethod

3.10. Summary

Following are the key concepts you should take away from this chapter:

- A *lambda expression* can be understood as a kind of anonymous function: it doesn't have a name, but it has a list of parameters, a body, a return type, and also possibly a list of exceptions that can be thrown.
- Lambda expressions let you pass code concisely.
- A *functional interface* is an interface that declares exactly one abstract method.
- Lambda expressions can be used only where a functional interface is expected.
- Lambda expressions let you provide the implementation of the abstract method of a functional interface directly inline and *treat the whole expression as an instance of a functional interface*.
- Java 8 comes with a list of common functional interfaces in the java.util .function package, which includes Predicate<T>, Function<T, R>, Supplier<T>, Consumer<T>, and BinaryOperator<T>, described in table 3.2.
- There are primitive specializations of common generic functional interfaces such as Predicate<T> and Function<T, R> that can be used to avoid boxing operations: IntPredicate, IntToLongFunction, and so on.
- The execute around pattern (that is, you need to execute a bit of behavior in the middle of code that's always required in a method, for example, resource allocation and cleanup) can be used with lambdas to gain additional flexibility and reusability.
- The type expected for a lambda expression is called the *target* type.
- Method references let you reuse an existing method implementation and pass it around directly.
- Functional interfaces such as Comparator, Predicate, and Function have several default methods that can be used to combine lambda expressions.

Chapter 4. Introducing streams

4.5. Summary

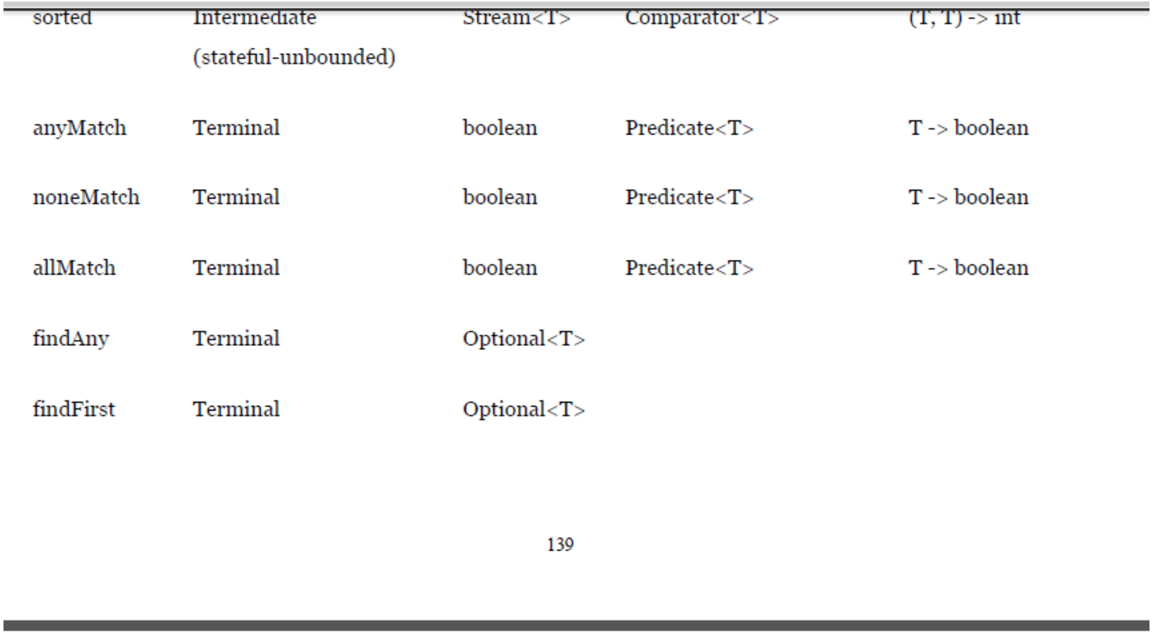
Here are some key concepts to take away from this chapter:

- A stream is a sequence of elements from a source that supports data processing operations.
- Streams make use of internal iteration: the iteration is abstracted away through operations such as filter, map, and sorted.
- There are two types of stream operations: intermediate and terminal operations.
- Intermediate operations such as filter and map return a stream and can be chained together. They're used to set up a pipeline of operations but don't produce any result.
- Terminal operations such as forEach and count return a nonstream value and process a stream pipeline to return a result.
- The elements of a stream are computed on demand.

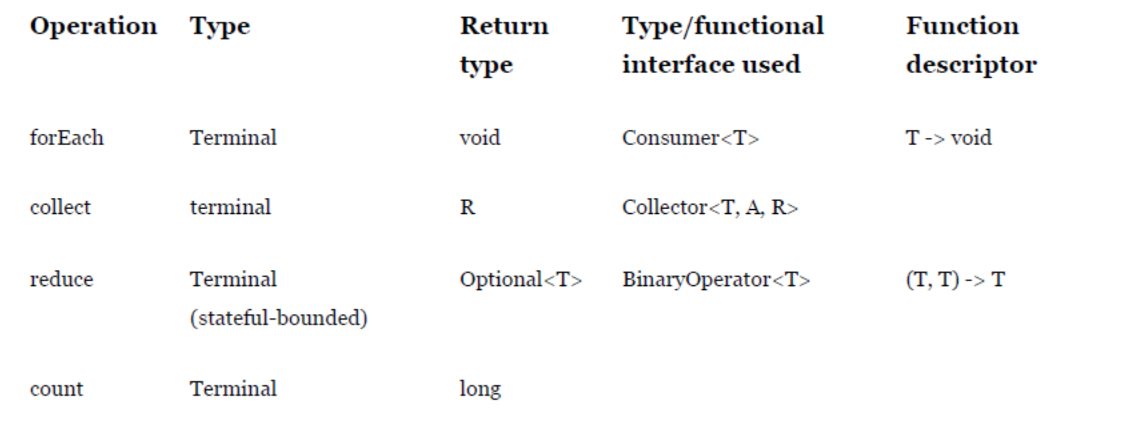
Chapter 5. Working with streams

Table 5.1. Intermediate and terminal operations

Operation	Type	Return type	Type/functional interface used	Function descriptor
filter	Intermediate	Stream<T>	Predicate<T>	T -> boolean
distinct	Intermediate (stateful-unbounded)	Stream<T>		
skip	Intermediate (stateful-bounded)	Stream<T>	long	
limit	Intermediate (stateful-bounded)	Stream<T>	long	
map	Intermediate	Stream<R>	Function<T, R>	T -> R
flatMap	Intermediate	Stream<R>	Function<T, Stream<R>>	T -> Stream<R>
sorted	Intermediate (stateful-unbounded)	Stream<T>	Comparator<T>	(T, T) -> int



sorted	Intermediate (stateful-unbounded)	Stream<T>	Comparator<T>	(T, T) -> int
anyMatch	Terminal	boolean	Predicate<T>	T -> boolean
noneMatch	Terminal	boolean	Predicate<T>	T -> boolean
allMatch	Terminal	boolean	Predicate<T>	T -> boolean
findAny	Terminal	Optional<T>		
findFirst	Terminal	Optional<T>		



Operation	Type	Return type	Type/functional interface used	Function descriptor
forEach	Terminal	void	Consumer<T>	T -> void
collect	terminal	R	Collector<T, A, R>	
reduce	Terminal (stateful-bounded)	Optional<T>	BinaryOperator<T>	(T, T) -> T
count	Terminal	long		

```
package streams.world;

import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

public class JavaInActionExcercise1 {

public static void main(String[] args) {
List<Transaction> transactions = dataPopulate();
```

```
System.out.println("transactions : " + transactions);

System.out
.println("*****//Q1: Find all transactions in the year 2011 and sort them by value(small to
high)***");

List<Transaction> transaction2011 = transactions.stream().filter(t -> t.getYear() == 2011)
.sorted(Comparator.comparing(Transaction::getValue)).collect(Collectors.toList());
System.out.println("transaction2011 : " + transaction2011);
System.out.println("*****************************************************************
*****************");

System.out.println("*****//Q2: What are all the unique cities where the traders work***");

List<String> uniqueCities =
transactions.stream().map(Transaction::getTrader).map(Trader::getCity).distinct()
.collect(Collectors.toList());

System.out.println("uniqueCities : " + uniqueCities);

System.out.println("*****//Q3 :  Find all traders from camnridge and sort them by name***");
List<Trader> listTraders = transactions.stream().map(Transaction::getTrader)
.filter(t -> t.getCity() == "Cambridge").sorted(Comparator.comparing(Trader::getName))
.collect(Collectors.toList());
System.out.println("listTraders : " + listTraders);

System.out.println("*****//Q4 :  Return a string of all traders names sorted alphabetically***");

String sortedNames = transactions.stream().map(t ->
t.getTrader().getName()).distinct().sorted().reduce("",
(n1, n2) -> n1 + " " + n2);
System.out.println("SortedNames : " + sortedNames);

System.out.println("*****//Q5 : Are any traders based in Milan***");
System.out.println(transactions.stream().map(Transaction::getTrader).anyMatch(t -> t.getCity()
== "Milan"));

System.out.println(
transactions.stream().map(Transaction::getTrader).filter(t -> t.getCity() == "Milan").findAny());

System.out.println("*****//Q6 :Print all transactions values from the traders living in cambridge
***");
List<Integer> listCambridgeTrx = transactions.stream().filter(t -> t.getTrader().getCity() ==
"Cambridge")
```

```
.map(Transaction::getValue).collect(Collectors.toList());
System.out.println("listCambridgeTrx : " + listCambridgeTrx);

System.out.println("*****//Q7 :What's the highest value of all the transaction?***");
Optional<Integer> highestValue =
transactions.stream().map(Transaction::getValue).reduce(Integer::max);
System.out.println("highestValue : " + highestValue.get());

System.out.println("*****//Q8 :Find the transaction with the smalles value***");
Optional<Transaction> smallesValue = transactions.stream()
.reduce((t1, t2) -> t1.getValue() < t2.getValue() ? t1 : t2);
System.out.println("smallesValue : " + smallesValue.get());

}

private static List<Transaction> dataPopulate() {
Trader raoul = new Trader("Raoul", "Cambridge");
Trader mario = new Trader("Mario", "Milan");
Trader alan = new Trader("Alan", "Cambridge");
Trader brian = new Trader("Brian", "Cambridge");

List<Transaction> transactions = Arrays.asList(new Transaction(brian, 2011, 300),
new Transaction(raoul, 2012, 1000), new Transaction(raoul, 2011, 400),
new Transaction(mario, 2012, 710), new Transaction(mario, 2012, 700), new Transaction(alan,
2012, 950));

return transactions;
}
}
```

5.8. Summary

It's been a long but rewarding chapter! You can now process collections more effectively. Indeed, streams let you express sophisticated data processing queries concisely. In addition, streams can be parallelized transparently. Here are some key concepts to take away from this chapter:

- The Streams API lets you express complex data processing queries. Common stream operations are summarized in table 5.1.
- You can filter and slice a stream using the filter, distinct, skip, and limit methods.
- You can extract or transform elements of a stream using the map and flatMap methods.
- You can find elements in a stream using the findFirst and findAny methods. You can match a given predicate in a stream using the allMatch, noneMatch, and anyMatch methods.
- These methods make use of short-circuiting: a computation stops as soon as a result is found; there's no need to process the whole stream.
- You can combine all elements of a stream iteratively to produce a result using the reduce method, for example, to calculate the sum or find the maximum of a stream.
- Some operations such as filter and map are stateless; they don't store any state. Some operations such as reduce store state to calculate a value. Some operations such as sorted and distinct also store state because they need to buffer all the elements of a stream before returning a new stream. Such operations are called *stateful operations*.
- There are three primitive specializations of streams: IntStream, DoubleStream, and LongStream. Their operations are also specialized accordingly.
- Streams can be created not only from a collection but also from values, arrays, files, and specific methods such as iterate and generate.
- An infinite stream is a stream that has no fixed size.

Chapter 6. Collecting data with streams

Table 6.1. The static factory methods of the Collectors class

Factory method	Returned type	Used to	Example use
toList	List<T>	Gather all the stream's items in a List.	Example use: List<Dish> dishes = menuStream.collect(toList());
toSet	Set<T>	Gather all the stream's items in a Set, eliminating duplicates.	Example use: Set<Dish> dishes = menuStream.collect(toSet());
toCollection	Collection<T>	Gather all the stream's items in the collection created by the provided supplier.	Example use: Collection<Dish> dishes = menuStream.collect(toCollection(), ArrayList::new);
counting	Long	Count the number of items in the stream.	Example use: long howManyDishes = menuStream.collect(counting());
summingInt	Integer	Sum the values of an Integer property of the items in the stream.	Example use: int totalCalories = menuStream.collect(summingInt(Dish::getCalories));
averagingInt	Double	Calculate the average value of an Integer property of the items in the stream.	Example use: double avgCalories = menuStream.collect(averagingInt(Dish::getCalories));
summarizingInt	IntSummary-Statistics	Collect statistics regarding an Integer property of the items in the stream, such as the maximum, minimum, total, and average.	Example use: IntSummaryStatistics menuStatistics = menuStream.collect(summarizingInt(Dish::getCalories));
joining	String	Concatenate the strings resulting from the invocation of the toString method on each item of the stream	Example use: String shortMenu = menuStream.map(Dish::getName).collect(joining(", "));

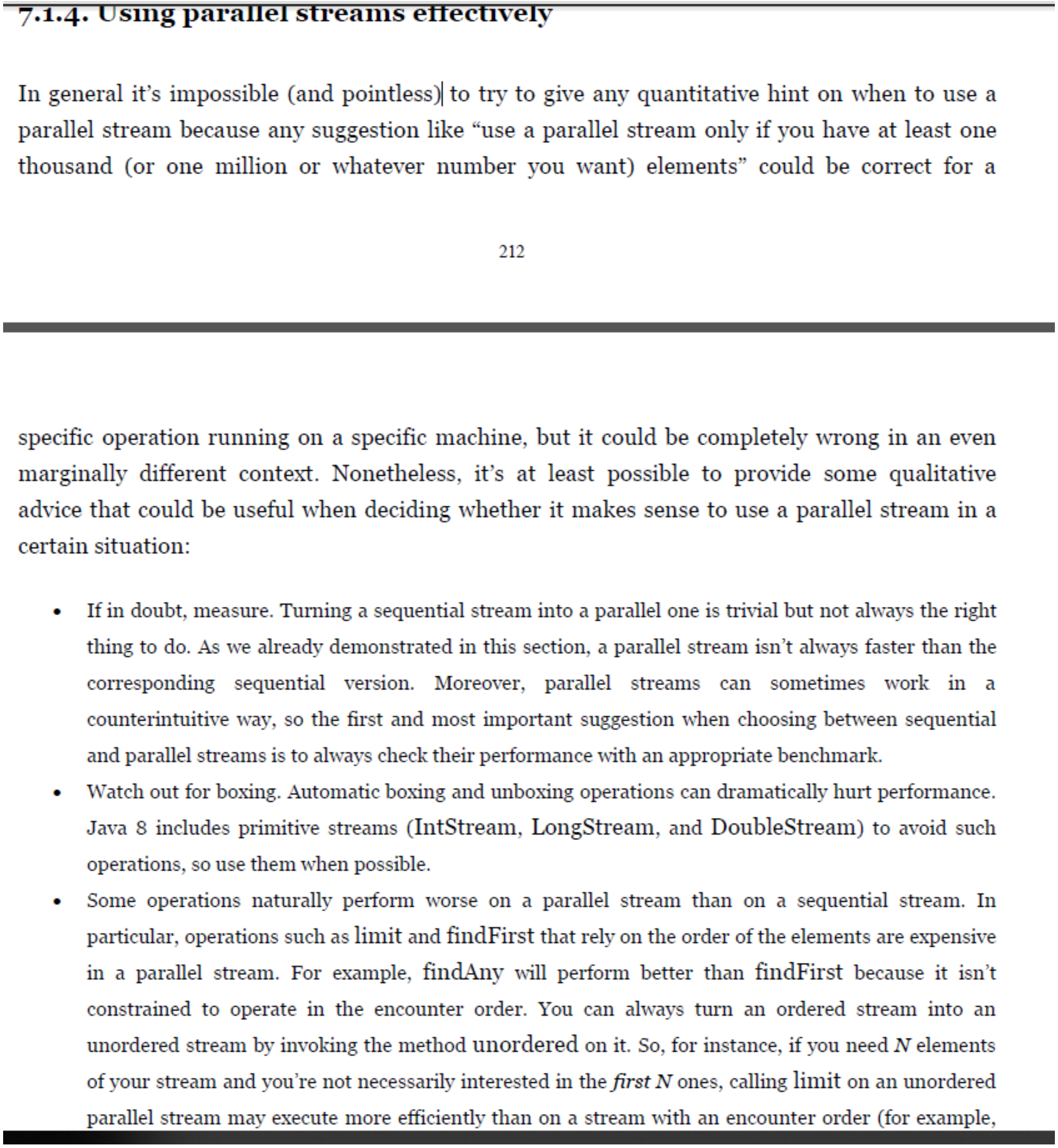
maxBy	Optional<T>	An Optional wrapping the maximal element in this stream according to the given comparator or Optional.empty() if the stream is empty.	Example use: Optional<Dish> fattest = menuStream.collect(maxBy(comparingInt(Dish::getCalories)));
minBy	Optional<T>	An Optional wrapping the minimal element in this stream according to the given comparator or Optional.empty() if the stream is empty.	Example use: Optional<Dish> lightest = menuStream.collect(minBy(comparingInt(Dish::getCalories)));
reducing	The type produced by the reduction operation	Reduce the stream to a single value starting from an initial value used as accumulator and iteratively combining it with each item of the stream using a BinaryOperator.	Example use: int totalCalories = menuStream.collect(reducing(0, Dish::getCalories, Integer::sum));
collectingAndThen	The type returned by the transforming function	Wrap another collector and apply a transformation function to its result	Example use: int howManyDishes = menuStream.collect(collectingAndThen(toList(), List::size));
groupingBy	Map<K, List<T>>	Group the items in the stream based on the value of one of their properties and use those values as keys in the resulting Map.	Example use: Map<Dish.Type, List<Dish>> dishesByType = menuStream.collect(groupingBy(Dish::getType));
partitioningBy	Map<Boolean, List<T>>	Partition the items in the stream based on the result of the application of a predicate to each of them.	Example use: Map<Boolean, List<Dish>> vegetarianDishes = menuStream.collect(partitioningBy(Dish::isVegetarian));

6.7. Summary

Following are the key concepts you should take away from this chapter:

- collect is a terminal operation that takes as argument various recipes (called collectors) for accumulating the elements of a stream into a summary result.
- Predefined collectors include reducing and summarizing stream elements into a single value, such as calculating the minimum, maximum, or average. Those collectors are summarized in table 6.1.
- Predefined collectors let you group elements of a stream with groupingBy and partition elements of a stream with partitioningBy.
- Collectors compose effectively to create multilevel groupings, partitions, and reductions.
- You can develop your own collectors by implementing the methods defined in the Collector interface.

Chapter 7. Parallel data processing and performance



7.1.4. Using parallel streams effectively

In general it's impossible (and pointless) to try to give any quantitative hint on when to use a parallel stream because any suggestion like "use a parallel stream only if you have at least one thousand (or one million or whatever number you want) elements" could be correct for a specific operation running on a specific machine, but it could be completely wrong in an even marginally different context. Nonetheless, it's at least possible to provide some qualitative advice that could be useful when deciding whether it makes sense to use a parallel stream in a certain situation:

- If in doubt, measure. Turning a sequential stream into a parallel one is trivial but not always the right thing to do. As we already demonstrated in this section, a parallel stream isn't always faster than the corresponding sequential version. Moreover, parallel streams can sometimes work in a counterintuitive way, so the first and most important suggestion when choosing between sequential and parallel streams is to always check their performance with an appropriate benchmark.
- Watch out for boxing. Automatic boxing and unboxing operations can dramatically hurt performance. Java 8 includes primitive streams (IntStream, LongStream, and DoubleStream) to avoid such operations, so use them when possible.
- Some operations naturally perform worse on a parallel stream than on a sequential stream. In particular, operations such as limit and findFirst that rely on the order of the elements are expensive in a parallel stream. For example, findAny will perform better than findFirst because it isn't constrained to operate in the encounter order. You can always turn an ordered stream into an unordered stream by invoking the method unordered on it. So, for instance, if you need *N* elements of your stream and you're not necessarily interested in the ***first*** *N* ones, calling limit on an unordered parallel stream may execute more efficiently than on a stream with an encounter order (for example,

of your stream and you're not necessarily interested in the *first N* ones, calling limit on an unordered parallel stream may execute more efficiently than on a stream with an encounter order (for example, when the source is a List).

- Consider the total computational cost of the pipeline of operations performed by the stream. With ***N*** being the number of elements to be processed and ***Q*** the approximate cost of processing one of these elements through the stream pipeline, the product of ***N*Q*** gives a rough qualitative estimation of this cost. A higher value for ***Q*** implies a better chance of good performance when using a parallel stream.
- For a small amount of data, choosing a parallel stream is almost never a winning decision. The advantages of processing in parallel only a few elements aren't enough to compensate for the additional cost introduced by the parallelization process.
- Take into account how well the data structure underlying the stream decomposes. For instance, an ArrayList can be split much more efficiently than a LinkedList, because the first can be evenly divided without traversing it, as it's necessary to do with the second. Also, the primitive streams created with the range factory method can be decomposed quickly. Finally, as you'll learn in section 7.3, you can get full control of this decomposition process by implementing your own Spliterator.
- The characteristics of a stream, and how the intermediate operations through the pipeline modify them, can change the performance of the decomposition process. For example, a SIZED stream can be divided into two equal parts, and then each part can be processed in parallel more effectively, but a filter operation can throw away an unpredictable number of elements, making the size of the stream itself unknown.

- Consider whether a terminal operation has a cheap or expensive merge step (for example, the combiner method in a Collector). If this is expensive, then the cost caused by the combination of the partial results generated by each substream can outweigh the performance benefits of a parallel stream.

Table 7.1. Stream sources and decomposability

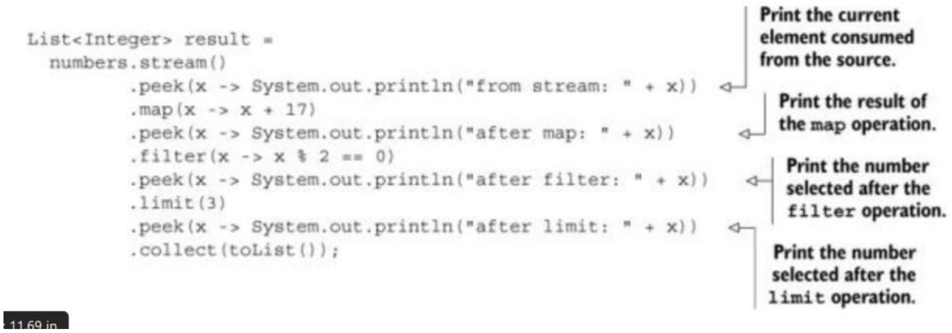
Source	Decomposability
ArrayList	Excellent
LinkedList	Poor
IntStream.range	Excellent
Stream.iterate	Poor
Stream.iterate	Poor
HashSet	Good
TreeSet	Good

7.4. Summary

In this chapter, you've learned the following:

- Internal iteration allows you to process a stream in parallel without the need to explicitly use and coordinate different threads in your code.
- Even if processing a stream in parallel is so easy, there's no guarantee that doing so will make your programs run faster under all circumstances. Behavior and performance of parallel software can sometimes be counterintuitive, and for this reason it's always necessary to measure them and be sure that you're not actually slowing your programs down.
- Parallel execution of an operation on a set of data, as done by a parallel stream, can provide a performance boost, especially when the number of elements to be processed is huge or the processing of each single element is particularly time consuming.
- From a performance point of view, using the right data structure, for instance, employing primitive streams instead of nonspecialized ones whenever possible, is almost always more important than trying to parallelize some operations.
- The fork/join framework lets you recursively split a parallelizable task into smaller tasks, execute them on different threads, and then combine the results of each subtask in order to produce the overall result.
- Spliterators define how a parallel stream can split the data it traverses.

Chapter 8. Refactoring, testing, and debugging



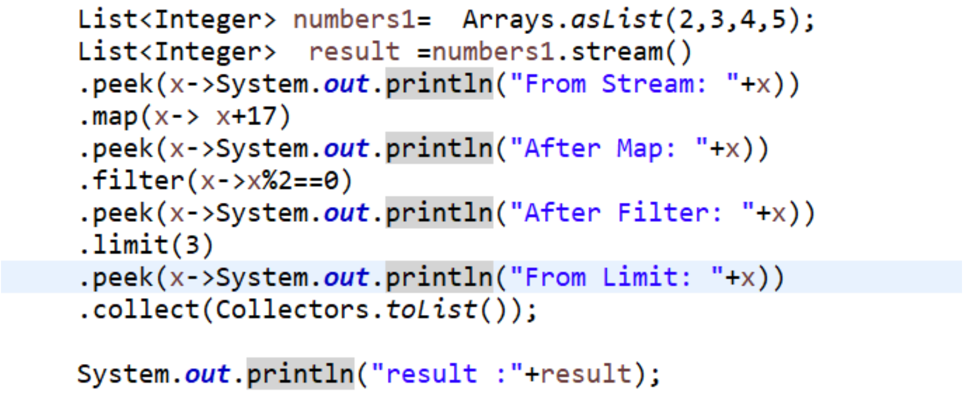
```
List<Integer> result =
  numbers.stream()
         .peek(x -> System.out.println("from stream: " + x))
         .map(x -> x + 17)
         .peek(x -> System.out.println("after map: " + x))
         .filter(x -> x % 2 == 0)
         .peek(x -> System.out.println("after filter: " + x))
         .limit(3)
         .peek(x -> System.out.println("after limit: " + x))
         .collect(toList());
```

Print the current element consumed from the source.

Print the result of the map operation.

Print the number selected after the filter operation.

Print the number selected after the limit operation.



```
List<Integer> numbers1=  Arrays.asList(2,3,4,5);
List<Integer>  result =numbers1.stream()
.peek(x->System.out.println("From Stream: "+x))
.map(x-> x+17)
.peek(x->System.out.println("After Map: "+x))
.filter(x->x%2==0)
.peek(x->System.out.println("After Filter: "+x))
.limit(3)
.peek(x->System.out.println("From Limit: "+x))
.collect(Collectors.toList());

System.out.println("result :"+result);
```

This will produce a useful output at each step of the pipeline:

```
from stream: 2
after map: 19
from stream: 3
after map: 20
after filter: 20
after limit: 20
from stream: 4
after map: 21
from stream: 5
after map: 22
after filter: 22
after limit: 22
```

8.5. Summary

Following are the key concepts you should take away from this chapter:

- Lambda expressions can make your code more readable and flexible.
- Consider converting anonymous classes to lambda expressions, but be wary of subtle semantic differences such as the meaning of the keyword this and shadowing of variables.
- Method references can make your code more readable compared to lambda expressions.
- Consider converting iterative collection processing to use the Streams API.
- Lambda expressions can help remove boilerplate code associated with several object-oriented design patterns such as strategy, template method, observer, chain of responsibility, and factory.
- Lambda expressions can be unit tested, but in general you should focus on testing the behavior of the methods where the lambda expressions appear.
- Consider extracting complex lambda expressions into regular methods.
- Lambda expressions can make stack traces less readable.
- The peek method of a stream is useful to log intermediate values as they flow past at certain points in a stream pipeline.

Chapter 9. Default methods

9.4.1. Three resolution rules to know

There are three rules to follow when a class inherits a method with the same signature from multiple places (such as another class or interface):

1. Classes always win. A method declaration in the class or a superclass takes priority over any default method declaration.

2. Otherwise, sub-interfaces win: the method with the same signature in the most specific default-providing interface is selected. (If B extends A, B is more specific than A).

3. Finally, if the choice is still ambiguous, the class inheriting from multiple interfaces has to explicitly select which default method implementation to use by overriding it and calling the desired method explicitly.

9.5. Summary

Following are the key concepts you should take away from this chapter:

- Interfaces in Java 8 can have implementation code through default methods and static methods.
- Default methods start with a default keyword and contain a body like class methods do.
- Adding an abstract method to a published interface is a source incompatibility.
- Default methods help library designers evolve APIs in a backward-compatible way.
- Default methods can be used for creating optional methods and multiple inheritance of behavior.
- There are resolution rules to resolve conflicts when a class inherits from several default methods with the same signature.
- A method declaration in the class or a superclass takes priority over any default method declaration. Otherwise, the method with the same signature in the most specific default-providing interface is selected.
- When two methods are equally specific, a class can explicitly override a method and select which one to call.

Chapter 10. Using Optional as a better alternative to Null

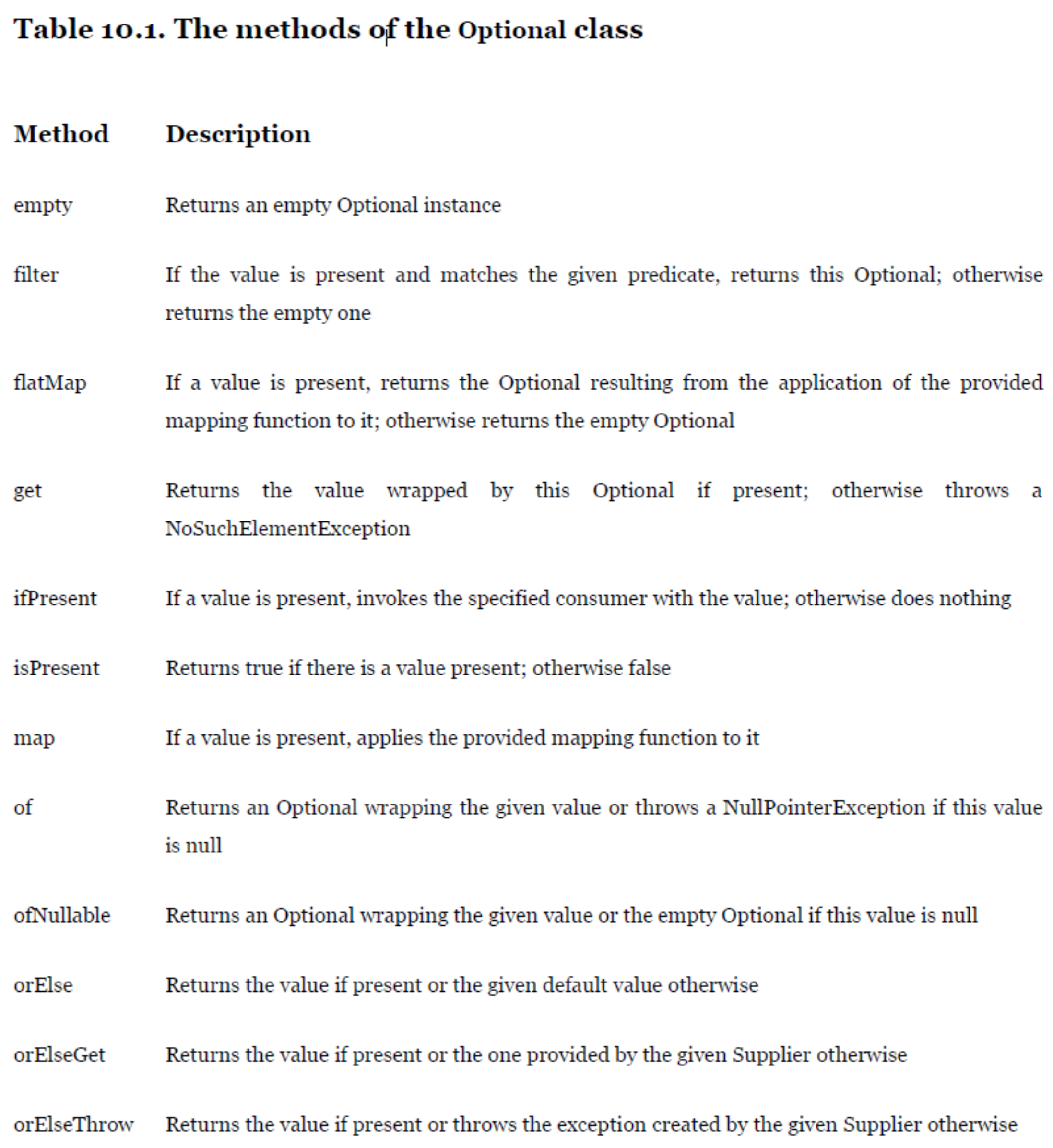


Table 10.1. The methods of the Optional class

Method	Description
empty	Returns an empty Optional instance
filter	If the value is present and matches the given predicate, returns this Optional; otherwise returns the empty one
flatMap	If a value is present, returns the Optional resulting from the application of the provided mapping function to it; otherwise returns the empty Optional
get	Returns the value wrapped by this Optional if present; otherwise throws a NoSuchElementException
ifPresent	If a value is present, invokes the specified consumer with the value; otherwise does nothing
isPresent	Returns true if there is a value present; otherwise false
map	If a value is present, applies the provided mapping function to it
of	Returns an Optional wrapping the given value or throws a NullPointerException if this value is null
ofNullable	Returns an Optional wrapping the given value or the empty Optional if this value is null
orElse	Returns the value if present or the given default value otherwise
orElseGet	Returns the value if present or the one provided by the given Supplier otherwise
orElseThrow	Returns the value if present or throws the exception created by the given Supplier otherwise

10.5. Summary

In this chapter, you've learned the following:

- null references have been historically introduced in programming languages to generally signal the absence of a value.

- Java 8 introduces the class java.util.Optional<T> to model the presence or absence of a value.
- You can create Optional objects with the static factory methods Optional.empty, Optional.of, and Optional.ofNullable.
- The Optional class supports many methods such as map, flatMap, and filter, which are conceptually similar to the methods of a stream.
- Using Optional forces you to actively unwrap an optional to deal with the absence of a value; as a result, you protect your code against unintended null pointer exceptions.
- Using Optional can help you design better APIs in which, just by reading the signature of a method, users can tell whether to expect an optional value.

Chapter 11. CompletableFuture: composable asynchronous programming

Figure 11.2. Concurrency vs. parallelism

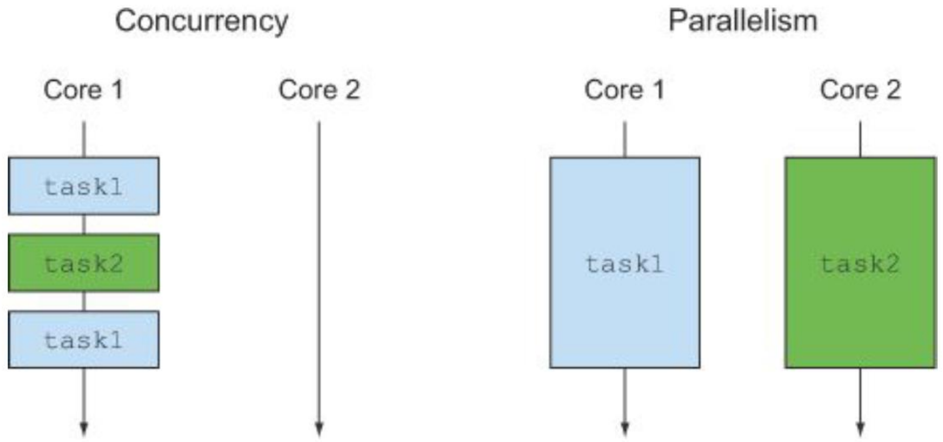
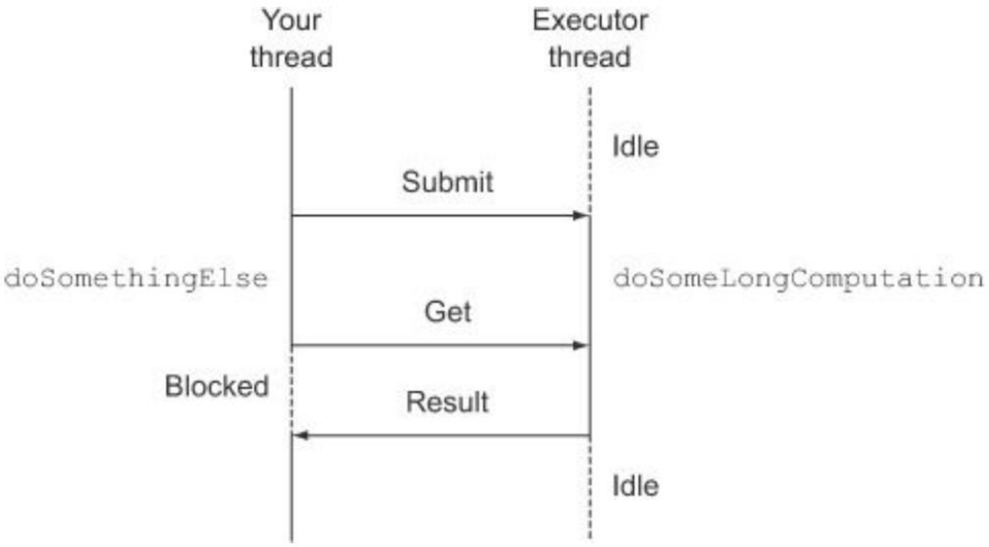


Figure 11.3. Using a Future to execute a long operation asynchronously



11.3.4. Using a custom Executor

In this case, a sensible choice seems to be to create an Executor with a number of threads in its pool that takes into account the actual workload you could expect in your application, but how do you correctly size it?

Sizing thread pools

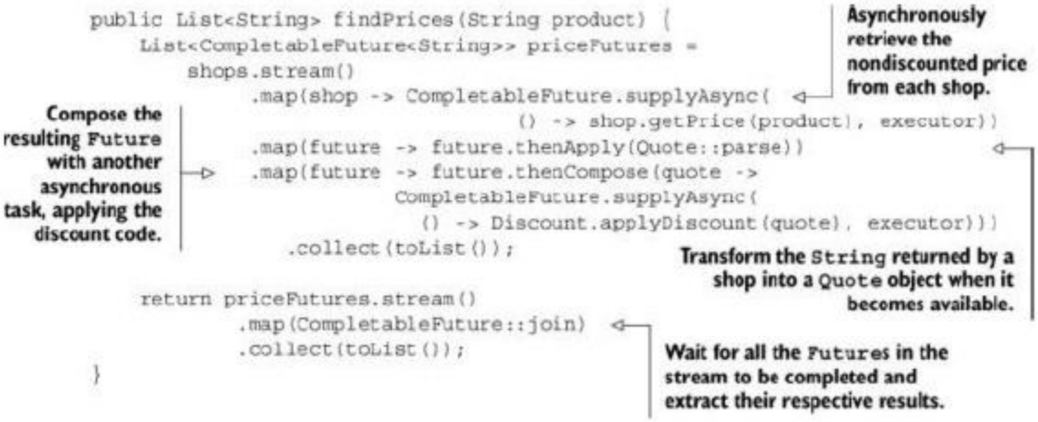
In the great book *Java Concurrency in Practice* (http://mng.bz/979c), Brian Goetz and coauthors give some advice to find the optimal size for a thread pool. This is important because if the number of threads in the pool is too big, they'll end up competing for scarce CPU and memory resources, wasting their time performing context switching. Conversely, if this number is too small (as it very likely is in your application), some of the cores of the CPU will remain underutilized. In particular, Goetz suggests that the right pool size to approximate a desired CPU utilization rate can be calculated with the following formula:

$N_{threads} = N_{CPU} * U_{CPU} * (1 + W/C)$

where

- N_{CPU} is the number of cores, available through Runtime.getRuntime().availableProcessors()
- U_{CPU} is the target CPU utilization (between 0 and 1), and
- W/C is the ratio of wait time to compute time

The application is spending about the 99% of the time waiting for the shops' responses, so you could estimate a W/C ratio of 100. This means that if your target is 100% CPU utilization, you should have a pool with 400 threads. In practice it will be wasteful to have more threads than shops, because in doing so you'll have threads in your pool that are never used. For this reason, you need to set up an Executor with a fixed number of threads equal to the number of shops you have to query, so there will be exactly one thread for each shop. But you must also set an upper limit of 100 threads in order to avoid a server crash for a larger number of shops, as shown in the following listing.



```
public List<String> findPrices(String product) {
    List<CompletableFuture<String>> priceFutures =
        shops.stream()
             .map(shop -> CompletableFuture.supplyAsync(
                                  () -> shop.getPrice(product), executor))
             .map(future -> future.thenApply(Quote::parse))
             .map(future -> future.thenCompose(quote ->
                          CompletableFuture.supplyAsync(
                            () -> Discount.applyDiscount(quote), executor)))
                .collect(toList());

    return priceFutures.stream()
            .map(CompletableFuture::join)
            .collect(toList());
}
```

Asynchronously retrieve the nondiscounted price from each shop.

Transform the String returned by a shop into a Quote object when it becomes available.

Compose the resulting Future with another asynchronous task, applying the discount code.

Wait for all the Futures in the stream to be completed and extract their respective results.

Things look a bit more complex this time, so try to understand what's going on here, step by step. The sequence of these three transformations is depicted in figure 11.5.

Figure 11.5. Composing synchronous operations and asynchronous tasks

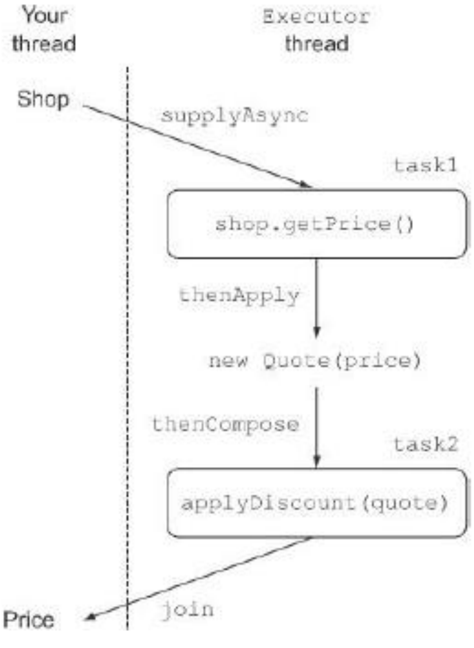
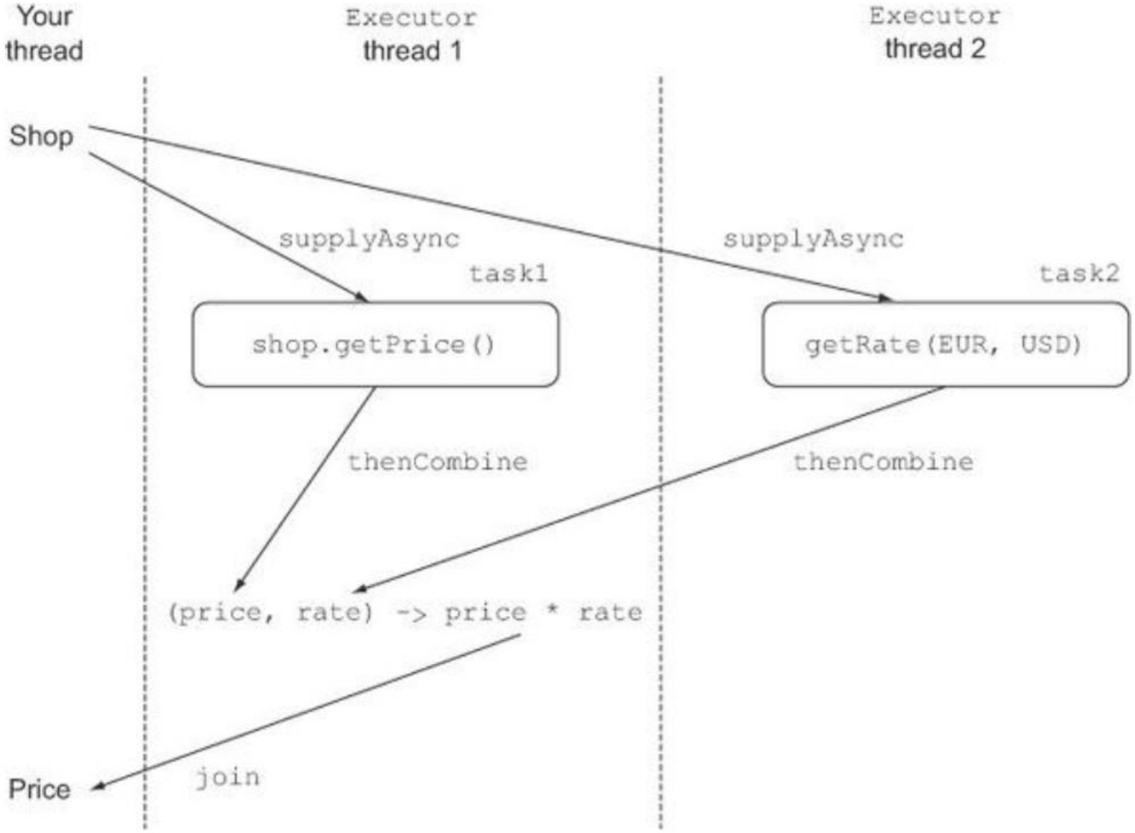


Figure 11.6. Combining two independent asynchronous tasks



11.6. Summary

In this chapter, you learned the following:

- Executing relatively long-lasting operations using asynchronous tasks can increase the performance and responsiveness of your application, especially if it relies on one or more remote external services.
- You should consider providing an asynchronous API to your clients. You can easily implement it using CompletableFutures features.
- A CompletableFuture also allows you to propagate and manage errors generated within an asynchronous task.
- You can asynchronously consume from a synchronous API by simply wrapping its invocation in a CompletableFuture.
- You can compose or combine multiple asynchronous tasks both when they're independent and when the result of one of them is used as the input to another.
- You can register a callback on a CompletableFuture to reactively execute some code when the Future completes and its result becomes available.
- You can determine when all values in a list of CompletableFutures have completed, or alternatively you can wait for just the first to complete.

Chapter 12. New Date and Time API

12.4. Summary

In this chapter, you've learned the following:

- The old java.util.Date class and all other classes used to model date and time in Java before Java 8 have many inconsistencies and design flaws, including their mutability and some poorly chosen offsets, defaults, and naming.
- The date-time objects of the new Date and Time API are all immutable.
- This new API provides two different time representations to manage the different needs of humans and machines when operating on it.
- You can manipulate date and time objects in both an absolute and relative manner, and the result of these manipulations is always a new instance, leaving the original one unchanged.
- TemporalAdjusters allow you to manipulate a date in a more complex way than just changing one of its values, and you can define and use your own custom date transformations.
- You can define a formatter to both print and parse date-time objects in a specific format. These formatters can be created from a pattern or programmatically and they're all thread-safe.
- You can represent a time zone, both relative to a specific region/location and as a fixed offset from UTC/Greenwich, and apply it to a date-time object in order to localize it.