Understanding, Accepting and Leveraging Optional in Java

EUGEN PARASCHIVSEPTEMBER 14, 2017[DEVELOPER TIPS, TRICKS & RESOURCES](https://stackify.com/developers/)

**Overview**

One of the most interesting features that Java 8 introduces to the language is the new [*Optional* class](http://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html). The main issue this class is intended to tackle is the infamous *NullPointerException* that every [Java](https://stackify.com/content/java/) programmer knows only too well.

Essentially, this is a wrapper class that contains an optional value, meaning it can either contain an object or it can simply be empty.

Optional comes along with **a strong move towards functional programming in Java** and is meant to help in that paradigm, but definitely also outside of that.

Let’s start with a simple use-case. Before Java 8, any number of operations involving accessing an object’s methods or properties could result in a *NullPointerException*:

String isocode = user.getAddress().getCountry().getIsocode().toUpperCase();

If we wanted to make sure we won’t hit the exception in this short example, we would need to do explicit checks for every value before accessing it:

if (user != null) {

Address address = user.getAddress();

if (address != null) {

Country country = address.getCountry();

if (country != null) {

String isocode = country.getIsocode();

if (isocode != null) {

isocode = isocode.toUpperCase();

}

}

}

}

As you can see, this can easily become cumbersome and hard to maintain.

To ease this process, let’s take a look at how we can use the *Optional* class instead, from creating and verifying an instance, to using the different methods it provides and combining it with other methods that return the same type, the latter being where the true power of *Optional*lies.

**Creating *Optional* Instances**

To reiterate, an object of this type can contain a value or be empty. You can create an empty Optional by using the method with the same name:

@Test(expected = NoSuchElementException.class)

public void whenCreateEmptyOptional\_thenNull() {

Optional<User> emptyOpt = Optional.empty();

emptyOpt.get();

}

Not surprisingly, attempting to access the value of the *emptyOpt* variable results in a *NoSuchElementException*.

To create an *Optional* object that can contain a value – you can use the *of()* and ofNullable() methods. The difference between the two is that the *of()* method will throw a *NullPointerException* if you pass it a *null* value as an argument:

@Test(expected = NullPointerException.class)

public void whenCreateOfEmptyOptional\_thenNullPointerException() {

Optional<User> opt = Optional.of(user);

}

As you can see, we’re not completely rid of the *NullPointerException*. For this reason, you should only use *of()* when you are sure the object is not *null*.

If the object can be both *null*or not-*null*, then you should instead choose the *ofNullable()* method:

Optional<User> opt = Optional.ofNullable(user);

**Accessing the Value of *Optional*Objects**

One way to retrieve the actual object inside the *Optional* instance is to use the *get()* method:

@Test

public void whenCreateOfNullableOptional\_thenOk() {

String name = "John";

Optional<String> opt = Optional.ofNullable(name);

assertEquals("John", opt.get());

}

However, as you saw before, this method throws an exception in case the value is *null*. To avoid this exception, you can choose to first verify if a value is present or not:

@Test

public void whenCheckIfPresent\_thenOk() {

User user = new User("john@gmail.com", "1234");

Optional<User> opt = Optional.ofNullable(user);

assertTrue(opt.isPresent());

assertEquals(user.getEmail(), opt.get().getEmail());

}

Another option for checking the presence of a value is the*ifPresent()*method. In addition to performing the check, this method also takes a *Consumer* argument and executes the lambda expression if the object is not empty:

opt.ifPresent( u -> assertEquals(user.getEmail(), u.getEmail()));

In this example, the assertion is only executed if the user object is not null.

Next, let’s look at ways in which alternatives for empty values can be provided.

[](https://info.stackify.com/cs/c/?cta_guid=a4c047af-5491-4b3b-aadb-776133b17fa1&placement_guid=330ab11e-5d8d-4985-a002-c2d9e85bde5c&portal_id=207384&canon=https%3A%2F%2Fstackify.com%2Foptional-java%2F&redirect_url=APefjpEt-l_AYQJhqevWTXYlGE4gUmPEtumbgAA-4NnUwI3z5sh7mSHZEMnb068eHiMZwEkWrfG1bLxPPUSIfsnnbOj4v7CkDrs0YqgjYuwYXz0YYfWjDsggITPO-iq3tNO8kcalONg3CaT3ACraWWyKdh0OxcYbdNh8Icd_caz9Irb7MHf8qL-fDb4QWtWczQ1r6tcQbU9NFy4Z6xYg0MMJEuUmHErhN-zoAS1GZD1tRNB6yFLRBjJsYMdQJoyUAPz7ZXrgvUN4s30rW7UoqxnN3LmEIpUcWPjruaHdWgtT3-7ULmi1Wdd6jHaVWX-uwONeRH1D60XCA1AO6ScnVXz6iEqqSuY6aB2HBIYyY6h5Bml9SG6OgIw&click=1911d105-eaae-4bd0-8ff2-2a5cc5eda496&hsutk=963a0e061ba860564e5c512c4bc2f666&signature=AAH58kFiUK_2O_-csFB6hCHpVJndPnXSBg&__hstc=23835621.963a0e061ba860564e5c512c4bc2f666.1588736592217.1591550448663.1591589653678.6&__hssc=23835621.5.1591589653678&__hsfp=3352070001)

**Returning Default Values**

The *Optional* class provides APIs for returning the value of the object or a default value if the object is empty.

The first method you can use for this purpose is*orElse()*, which works in a very straight-forward way: it returns the value if it’s present, or the argument it receives if not:

@Test

public void whenEmptyValue\_thenReturnDefault() {

User user = null;

User user2 = new User("anna@gmail.com", "1234");

User result = Optional.ofNullable(user).orElse(user2);

assertEquals(user2.getEmail(), result.getEmail());

}

Here, the *user* object was null, so *user2* was returned as a default instead.

If the initial value of the object is not null, then the default value is ignored:

@Test

public void whenValueNotNull\_thenIgnoreDefault() {

User user = new User("john@gmail.com","1234");

User user2 = new User("anna@gmail.com", "1234");

User result = Optional.ofNullable(user).orElse(user2);

assertEquals("john@gmail.com", result.getEmail());

}

The second API in the same category is *orElseGet()* – which behaves in a slightly different manner. In this case, the method returns the value if one is present, and if not it executes the *Supplier* functional interface that it receives as an argument, and returns the result of that execution:

User result = Optional.ofNullable(user).orElseGet( () -> user2);

**Difference between *orElse()* and *orElseGet()***

At first glance, it might seem as if the two methods have the same effect. However, this is not exactly the case. Let’s create some examples that highlight the similarity as well as the difference in behavior between the two.

First, let’s see how they behave when an object is empty:

@Test

public void givenEmptyValue\_whenCompare\_thenOk() {

User user = null

logger.debug("Using orElse");

User result = Optional.ofNullable(user).orElse(createNewUser());

logger.debug("Using orElseGet");

User result2 = Optional.ofNullable(user).orElseGet(() -> createNewUser());

}

private User createNewUser() {

logger.debug("Creating New User");

return new User("extra@gmail.com", "1234");

}

In the code above, both methods call the *createNewUser()* method which logs a message and returns a *User* object.

The output of this code is:

Using orElse

Creating New User

Using orElseGet

Creating New User

Therefore, when the object is empty and the default object is returned instead, there is no difference in behavior.

Next, let’s take a look at a similar example in which the *Optional* is not empty:

@Test

public void givenPresentValue\_whenCompare\_thenOk() {

User user = new User("john@gmail.com", "1234");

logger.info("Using orElse");

User result = Optional.ofNullable(user).orElse(createNewUser());

logger.info("Using orElseGet");

User result2 = Optional.ofNullable(user).orElseGet(() -> createNewUser());

}

The output this time is:

Using orElse

Creating New User

Using orElseGet

Here, both *Optional* objects contain a non-null value which the methods will return. However, the *orElse()* method will still create the default *User* object.**By contrast, the *orElseGet()* method will no longer create a *User* object.**

**This difference can have a significant effect on**[**performance**](https://stackify.com/java-performance-tuning/) if the operation executed involves more intensive calls, such as a web service call or a database query.

**Returning an Exception**

Next to the *orElse()* and *orElseGet()* methods, Optional also defines an *orElseThrow()* API – which instead of returning an alternate value, throws an exception instead if the object is empty:

@Test(expected = IllegalArgumentException.class)

public void whenThrowException\_thenOk() {

User result = Optional.ofNullable(user)

.orElseThrow( () -> new IllegalArgumentException());

}

Here, if the *user* value is null, an *IllegalArgumentException* is thrown.

This allows us to have a lot more flexible semantics and decide the exception that gets thrown instead of always seeing a *NullPointerException*.

Now that we have a good understanding of how we can leverage Optional by itself, let’s have a look at additional methods that can be used to apply transformations and filtering to *Optional* values.

**Transforming Values**

*Optional* values can be transformed in a number of ways; let’s start with *map()* and *flatMap()* methods.

First, let’s see an example that uses the *map()* API:

@Test

public void whenMap\_thenOk() {

User user = new User("anna@gmail.com", "1234");

String email = Optional.ofNullable(user)

.map(u -> u.getEmail()).orElse("default@gmail.com");

assertEquals(email, user.getEmail());

}

***map()* applies the *Function* argument to the value, then returns the result wrapped in an *Optional*.** This makes it possible to apply and chain further operations on the response – such *orElse()* here.

By comparison,*flatMap()* also takes a *Function* argument that is applied to an *Optional* value, and then returns the result directly.

To see this in action, let’s add a method that returns an *Optional*to the *User* class:

public class User {

private String position;

public Optional<String> getPosition() {

return Optional.ofNullable(position);

}

//...

}

Since the getter method returns an *Optional* of String value, you can use as the argument for *flatMap()*, where this is called for an *Optional User* object. The return will be the unwrapped *String* value:

@Test

public void whenFlatMap\_thenOk() {

User user = new User("anna@gmail.com", "1234");

user.setPosition("Developer");

String position = Optional.ofNullable(user)

.flatMap(u -> u.getPosition()).orElse("default");

assertEquals(position, user.getPosition().get());

}

**Filtering Values**

Alongside transforming the values, the *Optional* class also offers the possibility to “filter” them based on a condition.

**The *filter()* method takes a *Predicate* as an argument** and returns the value as it is if the test evaluates to true. Otherwise, if the test is false, the returned value is an empty *Optional*.

Let’s see an example of accepting or rejecting a *User* based on a very basic email verification:

@Test

public void whenFilter\_thenOk() {

User user = new User("anna@gmail.com", "1234");

Optional<User> result = Optional.ofNullable(user)

.filter(u -> u.getEmail() != null && u.getEmail().contains("@"));

assertTrue(result.isPresent());

}

The *result* object will contain a non-*null* value as a result of it passing the filter test.

**Chaining Methods of the *Optional* class**

For more powerful uses of *Optional*, you can also chain different combinations of most of its methods, given that most of them return objects of the same type.

Let’s rewrite the example in the introduction using *Optional*.

First, let’s refactor the classes so that the getter methods return *Optional* references:

public class User {

private Address address;

public Optional<Address> getAddress() {

return Optional.ofNullable(address);

}

// ...

}

public class Address {

private Country country;

public Optional<Country> getCountry() {

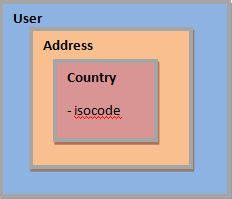
return Optional.ofNullable(country);

}

// ...

}

The structure above can be visually represented as a nested set:

[](http://inprogress.baeldung.com/wp-content/uploads/2017/08/user.png)

Now you can remove the *null* checks and use the *Optional* methods instead:

@Test

public void whenChaining\_thenOk() {

User user = new User("anna@gmail.com", "1234");

String result = Optional.ofNullable(user)

.flatMap(u -> u.getAddress())

.flatMap(a -> a.getCountry())

.map(c -> c.getIsocode())

.orElse("default");

assertEquals(result, "default");

}

The code above can be further reduced by using method references:

String result = Optional.ofNullable(user)

.flatMap(User::getAddress)

.flatMap(Address::getCountry)

.map(Country::getIsocode)

.orElse("default");

As a result, the code looks much cleaner than our early cumbersome, conditional-driven version.

**Java 9 Additions**

Next to the features introduced in Java 8, **Java 9 adds three more methods to the Optional class: *or()*, *ifPresentOrElse()* and *stream()*.**

The *or()* method is similar to *orElse()* and *orElseGet()* in the sense that it provides alternate behavior if the object is empty. In this case, the returned value is another *Optional* object that is produced by a *Supplier* argument.

If the object does contain a value, then the lambda expression is not executed:

@Test

public void whenEmptyOptional\_thenGetValueFromOr() {

User result = Optional.ofNullable(user)

.or( () -> Optional.of(new User("default","1234"))).get();

assertEquals(result.getEmail(), "default");

}

In the example above, if the *user* variable is null, then an *Optional* containing a *User* object with the email “default” is returned.

The *ifPresentOrElse()* method takes two arguments: a *Consumer* and a *Runnable*. If the object contains a value, then the *Consumer* action is executed; otherwise, the *Runnable* action is performed.

This method can be useful if you want to perform an action using the value if one is present, or simply keep track of whether a value was defined or not:

Optional.ofNullable(user).ifPresentOrElse( u -> logger.info("User is:" + u.getEmail()),

() -> logger.info("User not found"));

Lastly, the new *stream()* method allows you to benefit from the extensive *Stream* API by **transforming the instance to a *Stream* object**. This will be an empty *Stream* if no value is present, or a *Stream* containing a single value – in case the *Optional* contains a non-*null* value.

Let’s see an example of processing an *Optional* as a *Stream*:

@Test

public void whenGetStream\_thenOk() {

User user = new User("john@gmail.com", "1234");

List<String> emails = Optional.ofNullable(user)

.stream()

.filter(u -> u.getEmail() != null && u.getEmail().contains("@"))

.map( u -> u.getEmail())

.collect(Collectors.toList());

assertTrue(emails.size() == 1);

assertEquals(emails.get(0), user.getEmail());

}

Here the use of a *Stream* makes it possible to apply the *Stream* interface methods *filter()*, *map()* and *collect()* to obtain a *List*.

**How Should *Optional* Be Used**

There are a few things to consider when using *Optional*, to determine when and how it should be used.

One note of importance is that ***Optional* is not *Serializable***. For that reason, it’s not intended to be used as a field in a class.

If you do need to serialize an object that contains an *Optional* value, [the *Jackson* library](https://stackify.com/java-xml-jackson/) provides support for treating *Optionals* as ordinary objects. What this means is that *Jackson* treats empty objects as *null* and objects with a value as fields containing that value. This functionality can be found in the [jackson-modules-java8](https://github.com/FasterXML/jackson-modules-java8) project.

Another situation when it’s not very helpful to use the type is as a parameter for methods or constructors. This would lead to code that is unnecessarily complicated:

User user = new User("john@gmail.com", "1234", Optional.empty());

Instead, it’s much easier to use method overloading to handle parameter which aren’t mandatory.

**The intended use of *Optional* is mainly as a return type.** After obtaining an instance of this type, you can extract the value if it’s present or provide an alternate behavior if it’s not.

One very useful use-case of the *Optional* class is combining it with streams or other methods that return an *Optional* value to **build fluent APIs**.

Let’s see an example of using the *Stream findFirst()* method which returns an *Optional* object:

@Test

public void whenEmptyStream\_thenReturnDefaultOptional() {

List<User> users = new ArrayList<>();

User user = users.stream().findFirst().orElse(new User("default", "1234"));

assertEquals(user.getEmail(), "default");

}

**Conclusion**

*Optional* is a useful addition to the Java language, intended to minimize the number of *NullPointerExceptions* in your code, though not able to completely remove them.

It’s also a well designed and very natural addition to the new functional support added in Java 8.

Overall, this simple yet powerful class helps create code that’s, simply put, more readable and less error-prone than its procedural counterpart.

# A Guide to Streams in Java 8: In-Depth Tutorial With Examples

EUGEN PARASCHIVMARCH 18, 2020[DEVELOPER TIPS, TRICKS & RESOURCES](https://stackify.com/developers/)

### Overview

The addition of the Stream was one of the major features added to Java 8. This in-depth tutorial is an introduction to the many functionalities supported by streams, with a focus on simple, practical examples.

To understand this material, you need to have a basic, working knowledge of Java 8 (lambda expressions, [*Optional*](https://stackify.com/optional-java/), method references).

### Introduction

First of all, Java 8 Streams should not be confused with Java I/O streams (ex: FileInputStream etc); these have very little to do with each other.

Simply put, streams are wrappers around a data source, allowing us to operate with that data source and making bulk processing convenient and fast.

**A stream does not store data and, in that sense, is not a data structure. It also never modifies the underlying data source.**

This functionality – *[java.util.stream](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html" \t "_blank)* – supports functional-style operations on streams of elements, such as map-reduce transformations on collections.

Let’s now dive into few simple examples of stream creation and usage – before getting into terminology and core concepts.

#### Stream Creation

Let’s first obtain a stream from an existing array:

private static Employee[] arrayOfEmps = {

new Employee(1, "Jeff Bezos", 100000.0),

new Employee(2, "Bill Gates", 200000.0),

new Employee(3, "Mark Zuckerberg", 300000.0)

};

Stream.of(arrayOfEmps);

We can also obtain a stream from an existing list:

private static List<Employee> empList = Arrays.asList(arrayOfEmps);

empList.stream();

Note that **Java 8 added a new stream() method to the Collection interface.**

And we can create a stream from individual objects using Stream.of():

Stream.of(arrayOfEmps[0], arrayOfEmps[1], arrayOfEmps[2]);

Or simply using Stream.builder():

Stream.Builder<Employee> empStreamBuilder = Stream.builder();

empStreamBuilder.accept(arrayOfEmps[0]);

empStreamBuilder.accept(arrayOfEmps[1]);

empStreamBuilder.accept(arrayOfEmps[2]);

Stream<Employee> empStream = empStreamBuilder.build();

There are also other ways to obtain a stream, some of which we will see in sections below.

### Stream Operations

Let’s now see some common usages and operations we can perform on and with the help of the stream support in the language.

#### forEach

forEach() is simplest and most common operation; it loops over the stream elements, calling the supplied function on each element.

The method is so common that is has been introduced directly in Iterable, Map etc:

@Test

public void whenIncrementSalaryForEachEmployee\_thenApplyNewSalary() {

empList.stream().forEach(e -> e.salaryIncrement(10.0));

assertThat(empList, contains(

hasProperty("salary", equalTo(110000.0)),

hasProperty("salary", equalTo(220000.0)),

hasProperty("salary", equalTo(330000.0))

));

}

This will effectively call the salaryIncrement() on each element in the empList.

**forEach() is a terminal operation**, which means that, after the operation is performed, the stream pipeline is considered consumed, and can no longer be used. We’ll talk more about terminal operations in the next section.

#### map

map() produces a new stream after applying a function to each element of the original stream. The new stream could be of different type.

The following example converts the stream of Integers into the stream of Employees:

@Test

public void whenMapIdToEmployees\_thenGetEmployeeStream() {

Integer[] empIds = { 1, 2, 3 };

List<Employee> employees = Stream.of(empIds)

.map(employeeRepository::findById)

.collect(Collectors.toList());

assertEquals(employees.size(), empIds.length);

}

Here, we obtain an Integer stream of employee ids from an array. Each Integer is passed to the function employeeRepository::findById() – which returns the corresponding Employee object; this effectively forms an Employee stream.

#### collect

We saw how collect() works in the previous example; its one of the common ways to get stuff out of the stream once we are done with all the processing:

@Test

public void whenCollectStreamToList\_thenGetList() {

List<Employee> employees = empList.stream().collect(Collectors.toList());

assertEquals(empList, employees);

}

collect() performs mutable fold operations (repackaging elements to some data structures and applying some additional logic, concatenating them, etc.) on data elements held in the Stream instance.

The strategy for this operation is provided via the Collector interface implementation. In the example above, we used the toList collector to collect all Stream elements into a List instance.

#### filter

Next, let’s have a look at filter(); this produces a new stream that contains elements of the original stream that pass a given test (specified by a Predicate).

Let’s have a look at how that works:

@Test

public void whenFilterEmployees\_thenGetFilteredStream() {

Integer[] empIds = { 1, 2, 3, 4 };

List<Employee> employees = Stream.of(empIds)

.map(employeeRepository::findById)

.filter(e -> e != null)

.filter(e -> e.getSalary() > 200000)

.collect(Collectors.toList());

assertEquals(Arrays.asList(arrayOfEmps[2]), employees);

}

In the example above, we first filter out null references for invalid employee ids and then again apply a filter to only keep employees with salaries over a certain threshold.

#### findFirst

findFirst() returns an Optional for the first entry in the stream; the Optional can, of course, be empty:

@Test

public void whenFindFirst\_thenGetFirstEmployeeInStream() {

Integer[] empIds = { 1, 2, 3, 4 };

Employee employee = Stream.of(empIds)

.map(employeeRepository::findById)

.filter(e -> e != null)

.filter(e -> e.getSalary() > 100000)

.findFirst()

.orElse(null);

assertEquals(employee.getSalary(), new Double(200000));

}

Here, the first employee with the salary greater than 100000 is returned. If no such employee exists, then null is returned.

#### toArray

We saw how we used collect() to get data out of the stream. If we need to get an array out of the stream, we can simply use toArray():

@Test

public void whenStreamToArray\_thenGetArray() {

Employee[] employees = empList.stream().toArray(Employee[]::new);

assertThat(empList.toArray(), equalTo(employees));

}

The syntax Employee[]::new creates an empty array of Employee – which is then filled with elements from the stream.

#### flatMap

A stream can hold complex data structures like Stream<List<String>>. In cases like this, flatMap() helps us to flatten the data structure to simplify further operations:

@Test

public void whenFlatMapEmployeeNames\_thenGetNameStream() {

List<List<String>> namesNested = Arrays.asList(

Arrays.asList("Jeff", "Bezos"),

Arrays.asList("Bill", "Gates"),

Arrays.asList("Mark", "Zuckerberg"));

List<String> namesFlatStream = namesNested.stream()

.flatMap(Collection::stream)

.collect(Collectors.toList());

assertEquals(namesFlatStream.size(), namesNested.size() \* 2);

}

Notice how we were able to convert the Stream<List<String>> to a simpler Stream<String> – using the flatMap() API.

#### peek

We saw forEach() earlier in this section, which is a terminal operation. However, sometimes we need to perform multiple operations on each element of the stream before any terminal operation is applied.

peek() can be useful in situations like this. Simply put, it performs the specified operation on each element of the stream and returns a new stream which can be used further. **peek() is an intermediate operation**:

@Test

public void whenIncrementSalaryUsingPeek\_thenApplyNewSalary() {

Employee[] arrayOfEmps = {

new Employee(1, "Jeff Bezos", 100000.0),

new Employee(2, "Bill Gates", 200000.0),

new Employee(3, "Mark Zuckerberg", 300000.0)

};

List<Employee> empList = Arrays.asList(arrayOfEmps);

empList.stream()

.peek(e -> e.salaryIncrement(10.0))

.peek(System.out::println)

.collect(Collectors.toList());

assertThat(empList, contains(

hasProperty("salary", equalTo(110000.0)),

hasProperty("salary", equalTo(220000.0)),

hasProperty("salary", equalTo(330000.0))

));

}

Here, the first peek() is used to increment the salary of each employee. The second peek() is used to print the employees. Finally, collect() is used as the terminal operation.

### Method Types and Pipelines

**As we’ve been discussing, stream operations are divided into intermediate and terminal operations.**

Intermediate operations such as filter() return a new stream on which further processing can be done. Terminal operations, such as forEach(), mark the stream as consumed, after which point it can no longer be used further.

**A stream pipeline consists of a stream source, followed by zero or more intermediate operations, and a terminal operation.**

Here’s a sample stream pipeline, where empList is the source, filter() is the intermediate operation and count is the terminal operation:

@Test

public void whenStreamCount\_thenGetElementCount() {

Long empCount = empList.stream()

.filter(e -> e.getSalary() > 200000)

.count();

assertEquals(empCount, new Long(1));

}

Some operations are deemed**short-circuiting operations**. Short-circuiting operations allow computations on infinite streams to complete in finite time:

@Test

public void whenLimitInfiniteStream\_thenGetFiniteElements() {

Stream<Integer> infiniteStream = Stream.iterate(2, i -> i \* 2);

List<Integer> collect = infiniteStream

.skip(3)

.limit(5)

.collect(Collectors.toList());

assertEquals(collect, Arrays.asList(16, 32, 64, 128, 256));

}

Here, we use short-circuiting operations skip() to skip first 3 elements, and limit() to limit to 5 elements from the infinite stream generated using iterate().

We’ll talk more about infinite streams later on.

### Lazy Evaluation

One of the most important characteristics of streams is that they allow for significant optimizations through lazy evaluations.

**Computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed.**

**All intermediate operations are lazy, so they’re not executed until a result of a processing is actually needed.**

For example, consider the findFirst() example we saw earlier. How many times is the map() operation performed here? 4 times, since the input array contains 4 elements?

@Test

public void whenFindFirst\_thenGetFirstEmployeeInStream() {

Integer[] empIds = { 1, 2, 3, 4 };

Employee employee = Stream.of(empIds)

.map(employeeRepository::findById)

.filter(e -> e != null)

.filter(e -> e.getSalary() > 100000)

.findFirst()

.orElse(null);

assertEquals(employee.getSalary(), new Double(200000));

}

Stream performs the map and two filter operations, one element at a time.

It first performs all the operations on id 1. Since the salary of id 1 is not greater than 100000, the processing moves on to the next element.

Id 2 satisfies both of the filter predicates and hence the stream evaluates the terminal operation findFirst() and returns the result.

No operations are performed on id 3 and 4.

Processing streams lazily allows avoiding examining all the data when that’s not necessary. This behavior becomes even more important when the input stream is infinite and not just very large.

### Comparison Based Stream Operations

[](https://info.stackify.com/cs/c/?cta_guid=ec7e292a-1e2b-4f96-898b-00451b275126&placement_guid=4d2fc7b7-c1a4-4dd9-9b1a-599207901996&portal_id=207384&canon=https%3A%2F%2Fstackify.com%2Fstreams-guide-java-8%2F&redirect_url=APefjpF4bPAN8g8JLRUm_aba7cecJ9H5oN3wD9JPyc2m89ZQMYxB96Etn-VDOGZEt1kUxbCvi63Z2Zmxix3lNe4bWCDxRRUTA0bMcNm-F-xhPGuJfxjVR7NUuC1SqwUx-9Xm9MOvjD2kBFPPzJS9n3zDzpbZOLl08YcZtRuNZ_y-bWJ708TIoSeMPjSCI33qGs5rIIqHO6R4XXkEmhCwWhNnPwI7r8Ybo5zyWhiqsWaattdUM-eGfd6krQdeBG1KeuPGJ6ORlyuTYdmP8xcaxMYc_B4g1MzkntWNCXkgOuPp84vUxmH6ZsA&click=7f420170-aa13-44e7-a7fe-a0c26d741b60&hsutk=963a0e061ba860564e5c512c4bc2f666&signature=AAH58kEZrmTWVN5xpz3hVJK1HVO0yViuHQ&utm_referrer=https%3A%2F%2Fwww.google.com%2F&__hstc=23835621.963a0e061ba860564e5c512c4bc2f666.1588736592217.1591550448663.1591589653678.6&__hssc=23835621.1.1591589653678&__hsfp=3352070001)

#### sorted

Let’s start with the sorted() operation – this sorts the stream elements based on the comparator passed we pass into it.

For example, we can sort Employees based on their names:

@Test

public void whenSortStream\_thenGetSortedStream() {

List<Employee> employees = empList.stream()

.sorted((e1, e2) -> e1.getName().compareTo(e2.getName()))

.collect(Collectors.toList());

assertEquals(employees.get(0).getName(), "Bill Gates");

assertEquals(employees.get(1).getName(), "Jeff Bezos");

assertEquals(employees.get(2).getName(), "Mark Zuckerberg");

}

Note that short-circuiting will not be applied for sorted().

This means, in the example above, even if we had used findFirst() after the sorted(), the sorting of all the elements is done before applying the findFirst(). This happens because the operation cannot know what the first element is until the entire stream is sorted.

#### min and max

As the name suggests, min() and max() return the minimum and maximum element in the stream respectively, based on a comparator. They return an Optional since a result may or may not exist (due to, say, filtering):

@Test

public void whenFindMin\_thenGetMinElementFromStream() {

Employee firstEmp = empList.stream()

.min((e1, e2) -> e1.getId() - e2.getId())

.orElseThrow(NoSuchElementException::new);

assertEquals(firstEmp.getId(), new Integer(1));

}

We can also avoid defining the comparison logic by using Comparator.comparing():

@Test

public void whenFindMax\_thenGetMaxElementFromStream() {

Employee maxSalEmp = empList.stream()

.max(Comparator.comparing(Employee::getSalary))

.orElseThrow(NoSuchElementException::new);

assertEquals(maxSalEmp.getSalary(), new Double(300000.0));

}

#### distinct

distinct() does not take any argument and returns the distinct elements in the stream, eliminating duplicates. It uses the equals() method of the elements to decide whether two elements are equal or not:

@Test

public void whenApplyDistinct\_thenRemoveDuplicatesFromStream() {

List<Integer> intList = Arrays.asList(2, 5, 3, 2, 4, 3);

List<Integer> distinctIntList = intList.stream().distinct().collect(Collectors.toList());

assertEquals(distinctIntList, Arrays.asList(2, 5, 3, 4));

}

#### allMatch, anyMatch, and noneMatch

These operations all take a predicate and return a boolean. Short-circuiting is applied and processing is stopped as soon as the answer is determined:

@Test

public void whenApplyMatch\_thenReturnBoolean() {

List<Integer> intList = Arrays.asList(2, 4, 5, 6, 8);

boolean allEven = intList.stream().allMatch(i -> i % 2 == 0);

boolean oneEven = intList.stream().anyMatch(i -> i % 2 == 0);

boolean noneMultipleOfThree = intList.stream().noneMatch(i -> i % 3 == 0);

assertEquals(allEven, false);

assertEquals(oneEven, true);

assertEquals(noneMultipleOfThree, false);

}

allMatch() checks if the predicate is true for all the elements in the stream. Here, it returns false as soon as it encounters 5, which is not divisible by 2.

anyMatch() checks if the predicate is true for any one element in the stream. Here, again short-circuiting is applied and true is returned immediately after the first element.

noneMatch() checks if there are no elements matching the predicate. Here, it simply returns false as soon as it encounters 6, which is divisible by 3.

### Stream Specializations

From what we discussed so far, Stream is a stream of object references. However, there are also the IntStream, LongStream, and DoubleStream – which are primitive specializations for int, long and double respectively. These are quite convenient when dealing with a lot of numerical primitives.

These specialized streams do not extend Stream but extend BaseStream on top of which Stream is also built.

As a consequence, not all operations supported by Stream are present in these stream implementations. For example, the standard min() and max() take a comparator, whereas the specialized streams do not.

#### Creation

The most common way of creating an IntStream is to call mapToInt() on an existing stream:

@Test

public void whenFindMaxOnIntStream\_thenGetMaxInteger() {

Integer latestEmpId = empList.stream()

.mapToInt(Employee::getId)

.max()

.orElseThrow(NoSuchElementException::new);

assertEquals(latestEmpId, new Integer(3));

}

Here, we start with a Stream<Employee> and get an IntStream by supplying the Employee::getId to mapToInt. Finally, we call max() which returns the highest integer.

We can also use IntStream.of() for creating the IntStream:

IntStream.of(1, 2, 3);

or IntStream.range():

IntStream.range(10, 20)

which creates IntStream of numbers 10 to 19.

One important distinction to note before we move on to the next topic:

Stream.of(1, 2, 3)

This returns a Stream<Integer> and not IntStream.

Similarly, using map() instead of mapToInt() returns a Stream<Integer> and not an IntStream.:

empList.stream().map(Employee::getId);

#### Specialized Operations

Specialized streams provide additional operations as compared to the standard Stream – which are quite convenient when dealing with numbers.

For example sum(), average(), range() etc:

@Test

public void whenApplySumOnIntStream\_thenGetSum() {

Double avgSal = empList.stream()

.mapToDouble(Employee::getSalary)

.average()

.orElseThrow(NoSuchElementException::new);

assertEquals(avgSal, new Double(200000));

}

### Reduction Operations

**A reduction operation (also called as fold) takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation.** We already saw few reduction operations like findFirst(), min() and max().

Let’s see the general-purpose reduce() operation in action.

#### reduce

The most common form of reduce() is:

T reduce(T identity, BinaryOperator<T> accumulator)

where identity is the starting value and accumulator is the binary operation we repeated apply.

For example:

@Test

public void whenApplyReduceOnStream\_thenGetValue() {

Double sumSal = empList.stream()

.map(Employee::getSalary)

.reduce(0.0, Double::sum);

assertEquals(sumSal, new Double(600000));

}

Here, we start with the initial value of 0 and repeated apply Double::sum() on elements of the stream. Effectively we’ve implemented the DoubleStream.sum() by applying reduce() on Stream.

### Advanced collect

We already saw how we used Collectors.toList() to get the list out of the stream. Let’s now see few more ways to collect elements from the stream.

#### joining

@Test

public void whenCollectByJoining\_thenGetJoinedString() {

String empNames = empList.stream()

.map(Employee::getName)

.collect(Collectors.joining(", "))

.toString();

assertEquals(empNames, "Jeff Bezos, Bill Gates, Mark Zuckerberg");

}

Collectors.joining() will insert the delimiter between the two String elements of the stream. It internally uses a java.util.StringJoiner to perform the joining operation.

#### toSet

We can also use toSet() to get a set out of stream elements:

@Test

public void whenCollectBySet\_thenGetSet() {

Set<String> empNames = empList.stream()

.map(Employee::getName)

.collect(Collectors.toSet());

assertEquals(empNames.size(), 3);

}

#### toCollection

We can use Collectors.toCollection() to extract the elements into any other collection by passing in a Supplier<Collection>. We can also use a constructor reference for the Supplier:

@Test

public void whenToVectorCollection\_thenGetVector() {

Vector<String> empNames = empList.stream()

.map(Employee::getName)

.collect(Collectors.toCollection(Vector::new));

assertEquals(empNames.size(), 3);

}

Here, an empty collection is created internally, and its add() method is called on each element of the stream.

#### summarizingDouble

summarizingDouble() is another interesting collector – which applies a double-producing mapping function to each input element and returns a special class containing statistical information for the resulting values:

@Test

public void whenApplySummarizing\_thenGetBasicStats() {

DoubleSummaryStatistics stats = empList.stream()

.collect(Collectors.summarizingDouble(Employee::getSalary));

assertEquals(stats.getCount(), 3);

assertEquals(stats.getSum(), 600000.0, 0);

assertEquals(stats.getMin(), 100000.0, 0);

assertEquals(stats.getMax(), 300000.0, 0);

assertEquals(stats.getAverage(), 200000.0, 0);

}

Notice how we can analyze the salary of each employee and get statistical information on that data – such as min, max, average etc.

summaryStatistics() can be used to generate similar result when we’re using one of the specialized streams:

@Test

public void whenApplySummaryStatistics\_thenGetBasicStats() {

DoubleSummaryStatistics stats = empList.stream()

.mapToDouble(Employee::getSalary)

.summaryStatistics();

assertEquals(stats.getCount(), 3);

assertEquals(stats.getSum(), 600000.0, 0);

assertEquals(stats.getMin(), 100000.0, 0);

assertEquals(stats.getMax(), 300000.0, 0);

assertEquals(stats.getAverage(), 200000.0, 0);

}

#### partitioningBy

We can partition a stream into two – based on whether the elements satisfy certain criteria or not.

Let’s split our List of numerical data, into even and ods:

@Test

public void whenStreamPartition\_thenGetMap() {

List<Integer> intList = Arrays.asList(2, 4, 5, 6, 8);

Map<Boolean, List<Integer>> isEven = intList.stream().collect(

Collectors.partitioningBy(i -> i % 2 == 0));

assertEquals(isEven.get(true).size(), 4);

assertEquals(isEven.get(false).size(), 1);

}

Here, the stream is partitioned into a Map, with even and odds stored as true and false keys.

#### groupingBy

groupingBy() offers advanced partitioning – where we can partition the stream into more than just two groups.

It takes a classification function as its parameter. This classification function is applied to each element of the stream.

The value returned by the function is used as a key to the map that we get from the groupingBy collector:

@Test

public void whenStreamGroupingBy\_thenGetMap() {

Map<Character, List<Employee>> groupByAlphabet = empList.stream().collect(

Collectors.groupingBy(e -> new Character(e.getName().charAt(0))));

assertEquals(groupByAlphabet.get('B').get(0).getName(), "Bill Gates");

assertEquals(groupByAlphabet.get('J').get(0).getName(), "Jeff Bezos");

assertEquals(groupByAlphabet.get('M').get(0).getName(), "Mark Zuckerberg");

}

In this quick example, we grouped the employees based on the initial character of their first name.

#### mapping

groupingBy() discussed in the section above, groups elements of the stream with the use of a Map.

However, sometimes we might need to group data into a type other than the element type.

Here’s how we can do that; we can use mapping() which can actually adapt the collector to a different type – using a mapping function:

@Test

public void whenStreamMapping\_thenGetMap() {

Map<Character, List<Integer>> idGroupedByAlphabet = empList.stream().collect(

Collectors.groupingBy(e -> new Character(e.getName().charAt(0)),

Collectors.mapping(Employee::getId, Collectors.toList())));

assertEquals(idGroupedByAlphabet.get('B').get(0), new Integer(2));

assertEquals(idGroupedByAlphabet.get('J').get(0), new Integer(1));

assertEquals(idGroupedByAlphabet.get('M').get(0), new Integer(3));

}

Here mapping() maps the stream element Employee into just the employee id – which is an Integer – using the getId() mapping function. These ids are still grouped based on the initial character of employee first name.

#### reducing

reducing() is similar to reduce() – which we explored before. It simply returns a collector which performs a reduction of its input elements:

@Test

public void whenStreamReducing\_thenGetValue() {

Double percentage = 10.0;

Double salIncrOverhead = empList.stream().collect(Collectors.reducing(

0.0, e -> e.getSalary() \* percentage / 100, (s1, s2) -> s1 + s2));

assertEquals(salIncrOverhead, 60000.0, 0);

}

Here reducing() gets the salary increment of each employee and returns the sum.

reducing() is most useful when used in a multi-level reduction, downstream of groupingBy() or partitioningBy(). To perform a simple reduction on a stream, use reduce() instead.

For example, let’s see how we can use reducing() with groupingBy():

@Test

public void whenStreamGroupingAndReducing\_thenGetMap() {

Comparator<Employee> byNameLength = Comparator.comparing(Employee::getName);

Map<Character, Optional<Employee>> longestNameByAlphabet = empList.stream().collect(

Collectors.groupingBy(e -> new Character(e.getName().charAt(0)),

Collectors.reducing(BinaryOperator.maxBy(byNameLength))));

assertEquals(longestNameByAlphabet.get('B').get().getName(), "Bill Gates");

assertEquals(longestNameByAlphabet.get('J').get().getName(), "Jeff Bezos");

assertEquals(longestNameByAlphabet.get('M').get().getName(), "Mark Zuckerberg");

}

Here we group the employees based on the initial character of their first name. Within each group, we find the employee with the longest name.

### Parallel Streams

Using the support for parallel streams, we can perform stream operations in parallel without having to write any boilerplate code; we just have to designate the stream as parallel:

@Test

public void whenParallelStream\_thenPerformOperationsInParallel() {

Employee[] arrayOfEmps = {

new Employee(1, "Jeff Bezos", 100000.0),

new Employee(2, "Bill Gates", 200000.0),

new Employee(3, "Mark Zuckerberg", 300000.0)

};

List<Employee> empList = Arrays.asList(arrayOfEmps);

empList.stream().parallel().forEach(e -> e.salaryIncrement(10.0));

assertThat(empList, contains(

hasProperty("salary", equalTo(110000.0)),

hasProperty("salary", equalTo(220000.0)),

hasProperty("salary", equalTo(330000.0))

));

}

Here salaryIncrement() would get executed in parallel on multiple elements of the stream, by simply adding the parallel() syntax.

This functionality can, of course, be [tuned and configured further](http://www.baeldung.com/java-8-parallel-streams-custom-threadpool), if you need more control over the performance characteristics of the operation.

As is the case with writing multi-threaded code, we need to be aware of few things while using parallel streams:

1. We need to ensure that the code is thread-safe. Special care needs to be taken if the operations performed in parallel modifies shared data.
2. We should not use parallel streams if the order in which operations are performed or the order returned in the output stream matters. For example operations like findFirst() may generate the different result in case of parallel streams.
3. Also, we should ensure that it is worth making the code execute in parallel. Understanding the performance characteristics of the operation in particular, [but also of the system as a whole](https://stackify.com/java-performance-tuning/) – is naturally very important here.

### Infinite Streams

Sometimes, we might want to perform operations while the elements are still getting generated. We might not know beforehand how many elements we’ll need. Unlike using list or map, where all the elements are already populated, we can use infinite streams, also called as unbounded streams.

There are two ways to generate infinite streams:

#### generate

We provide a Supplier to generate() which gets called whenever new stream elements need to be generated:

@Test

public void whenGenerateStream\_thenGetInfiniteStream() {

Stream.generate(Math::random)

.limit(5)

.forEach(System.out::println);

}

Here, we pass Math::random() as a Supplier, which returns the next random number.

With infinite streams, we need to provide a condition to eventually terminate the processing. One common way of doing this is using limit(). In above example, we limit the stream to 5 random numbers and print them as they get generated.

Please note that the Supplier passed to generate() could be stateful and such stream may not produce the same result when used in parallel.

#### iterate

iterate() takes two parameters: an initial value, called seed element and a function which generates next element using the previous value. iterate(), by design, is stateful and hence may not be useful in parallel streams:

@Test

public void whenIterateStream\_thenGetInfiniteStream() {

Stream<Integer> evenNumStream = Stream.iterate(2, i -> i \* 2);

List<Integer> collect = evenNumStream

.limit(5)

.collect(Collectors.toList());

assertEquals(collect, Arrays.asList(2, 4, 8, 16, 32));

}

Here, we pass 2 as the seed value, which becomes the first element of our stream. This value is passed as input to the lambda, which returns 4. This value, in turn, is passed as input in the next iteration.

This continues until we generate the number of elements specified by limit() which acts as the terminating condition.

### File Operations

Let’s see how we could use the stream in file operations.

#### File Write Operation

@Test

public void whenStreamToFile\_thenGetFile() throws IOException {

String[] words = {

"hello",

"refer",

"world",

"level"

};

try (PrintWriter pw = new PrintWriter(

Files.newBufferedWriter(Paths.get(fileName)))) {

Stream.of(words).forEach(pw::println);

}

}

Here we use forEach() to write each element of the stream into the file by calling PrintWriter.println().

#### File Read Operation

private List<String> getPalindrome(Stream<String> stream, int length) {

return stream.filter(s -> s.length() == length)

.filter(s -> s.compareToIgnoreCase(

new StringBuilder(s).reverse().toString()) == 0)

.collect(Collectors.toList());

}

@Test

public void whenFileToStream\_thenGetStream() throws IOException {

List<String> str = getPalindrome(Files.lines(Paths.get(fileName)), 5);

assertThat(str, contains("refer", "level"));

}

Here Files.lines() returns the lines from the file as a Stream which is consumed by the getPalindrome() for further processing.

getPalindrome() works on the stream, completely unaware of how the stream was generated. This also increases code reusability and simplifies unit testing.

### Java Streams Improvements In Java 9

Java 8 brought Java streams to the world. However, the following version of the language also contributed to the feature. So, we’ll now give a brief overview of the improvements that Java 9 brought to the Streams API. Let’s do it.

#### takeWhile

The takeWhile method is one of the new additions to the Streams API. It does what its name implies: it **takes**(elements from a stream) **while**a given condition is true. The moment the condition becomes false, it quits and returns a new stream with just the elements that matched the predicate. In other words, it’s like a filter with a condition. Let’s see a quick example.

Stream.iterate(1, i -> i + 1)

.takeWhile(n -> n <= 10)

.map(x -> x \* x)

.forEach(System.out::println);

In the code above we obtain an infinite stream and then use the takeWhile method to select the numbers that are less than or equals to 10. After that, we calculate their squares and print those.

You might be wondering what’s the difference between takeWhile and filter. After all, you could accomplish the same result with the following code:

Stream.iterate(1, i -> i + 1)

.filter(x -> x <= 10)

.map(x -> x \* x)

.forEach(System.out::println);

Well, in this particular scenario, the two methods achieve the same result, but that’s not always the case. Let’s illustrate the difference with another example:

Stream.of(1,2,3,4,5,6,7,8,9,0,9,8,7,6,5,4,3,2,1,0)

.takeWhile(x -> x <= 5)

.forEach(System.out::println);

Stream.of(1,2,3,4,5,6,7,8,9,0,9,8,7,6,5,4,3,2,1,0)

.filter(x -> x <= 5)

.forEach(System.out::println);

Here, we have two identical streams, which we filter using takeWhile and filter, respectively. So, what’s the difference? If you run the code above you’ll see that the first version prints out:

1

2

3

4

5

while the version with filter results in

1

2

3

4

5

0

5

4

3

2

1

0

As you can see, filter() applies the predicate throughout the whole sequence. On the other hand, takeWhile stops evaluating as soon as it finds the first occurrence where the condition is false.

#### dropWhile

The dropWhile method does pretty much the same thing the takewhile does but in reverse. Confused? It’s simple: while takewhile takes while its condition is true, dropwhile drops elements while the condition is true. That is to say: the previous method uses the predicate (the condition) to select the elements to preserve in the new stream it returns. This method does the opposite, using the condition to select the items not to include in the resulting stream. Let’s see an example:

Stream.of(1,2,3,4,5,6,7,8,9,0,9,8,7,6,5,4,3,2,1,0)

.dropWhile(x -> x <= 5)

.forEach(System.out::println);

This is the same as the previous example, the only difference being that we’re using dropWhile instead of takeWhile. That is to say, we’re now dropping elements that are less than or equals to five. The resulting items are:

6

7

8

9

0

9

8

7

6

5

4

3

2

1

0

As you can see, there are numbers less than or equals to five in the latter half of the sequence. Why? It’s simple: they came after the first element which failed to match the predicate, so the method stopped dropping at that point.

#### iterate

We’ve already mentioned the original iterate() method that was introduced in the 8th version of Java. Java 9 brings an override of the method. So, what’s the difference?

As you’ve learned, the original incarnation of the method had two arguments: the initializer (a.k.a. the seed) and the function that generates the next value. The problem with the method is that it didn’t include a way for the loop to quit. That’s great when you’re trying to create infinite streams, but that’s not always the case.

In Java 9 we have the new version of iterate(), which adds a new parameter, which is a predicate used to decide when the loop should terminate. As long as the condition remains true, we keep going.

Consider the following example:

Stream.

iterate(1, i -> i < 256, i -> i \* 2)

.forEach(System.out::println);

The code above prints the powers of two, as long as they’re less than 256. We could say that the new iterate() method is a replacement for the good-old for statement. In fact, the code above is equivalent to the following excerpt:

for (int i = 1; i < 256; i\*=2) {

System.out.println(i);

}

#### ofNullable

The last item in this list of additions to the Stream APIs is a powerful way not only to avoid the dreaded [null pointer exception](https://stackify.com/types-of-exceptions-java/) but also to write cleaner code. Hopefully, it’s very straightforward. Check out the following example:

Stream<Integer> result = number != null

? Stream.of(number)

: Stream.empty();

Assume that number refers to some integer obtained through the UI, the network, filesystem or another external untrusted source. So, it could be null. We wouldn’t want to create a stream with a null element; that could result in a null pointer exception at some point. To avoid that that we can check for null and return an empty stream.

The example above is a contrived example, sure. In real life, code in similar scenarios could become really messy, really fast. We could employ ofNullable() instead:

Stream<Integer> result = Stream.ofNullable(number);

The new method returns empty Optionals in it receives null, avoiding runtime errors in scenarios that would normally cause one, like in the following example:

Integer number = null;

Stream<Integer> result = Stream.ofNullable(number);

result.map(x -> x \* x).forEach(System.out::println);

%MCEPASTEBIN%

### Java Streams: What Are The Next Steps?

In this article, we focused on the details of the new Stream functionality in Java 8. We saw various operations supported and how lambdas and pipelines can be used to write concise code. We also saw some characteristics of streams like lazy evaluation, parallel and infinite streams. You’ll find the sources of the examples [over on GitHub](https://github.com/Baeldung/stackify/tree/master/core-java/src/test/java/com/stackify/stream).

Now, what should you do next? Well, there’s a lot to explore in your journey to be a better Java developer, so here are a few suggestions.

For starters, you can continue your exploration of the concepts you’ve seen today with a look at [the reactive paradigm](https://stackify.com/reactive-spring-5/), made possible by very similar concepts to the one we discussed here.

Additionally, keep in touch with the Stackify blog. We’re always publishing articles that might be of interest to you. You might need to learn more about the [main Java frameworks](https://stackify.com/10-of-the-most-popular-java-frameworks-of-2020/), or how to [properly handle exceptions in the language](https://stackify.com/best-practices-exceptions-java/). In today’s article, we’ve covered an important feature that was introduced with Java 8. The language has come a long way since then and you might want to [check out more recent developments](https://stackify.com/java-12-new-features-and-enhancements-developers-should-know/).

Finally, to be a great developer you can’t overlook performance. We have posts that cover from [Java performance tuning tips](https://stackify.com/java-performance-tuning/) to the [main tools you should check about](https://stackify.com/java-performance-tools-8-types-tools-need-know/), and a lot more in between.

And speaking of tools, you might want to take a look at the free profiler by Stackify, Prefix. With Prefix, you can monitor both Windows desktop and web applications, reviewing their performance, finding hidden exceptions and solving bugs before they get to production.

# Package java.util.stream Description

Classes to support functional-style operations on streams of elements, such as map-reduce transformations on collections. For example:

int sum = widgets.stream()

.filter(b -> b.getColor() == RED)

.mapToInt(b -> b.getWeight())

.sum();

Here we use widgets, a Collection<Widget>, as a source for a stream, and then perform a filter-map-reduce on the stream to obtain the sum of the weights of the red widgets. (Summation is an example of a [reduction](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#Reduction) operation.)

The key abstraction introduced in this package is stream. The classes [Stream](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html), [IntStream](https://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.html" \o "interface in java.util.stream), [LongStream](https://docs.oracle.com/javase/8/docs/api/java/util/stream/LongStream.html" \o "interface in java.util.stream), and [DoubleStream](https://docs.oracle.com/javase/8/docs/api/java/util/stream/DoubleStream.html" \o "interface in java.util.stream) are streams over objects and the primitive int, long and double types. Streams differ from collections in several ways:

* No storage. A stream is not a data structure that stores elements; instead, it conveys elements from a source such as a data structure, an array, a generator function, or an I/O channel, through a pipeline of computational operations.
* Functional in nature. An operation on a stream produces a result, but does not modify its source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
* Laziness-seeking. Many stream operations, such as filtering, mapping, or duplicate removal, can be implemented lazily, exposing opportunities for optimization. For example, "find the first String with three consecutive vowels" need not examine all the input strings. Stream operations are divided into intermediate (Stream-producing) operations and terminal (value- or side-effect-producing) operations. Intermediate operations are always lazy.
* Possibly unbounded. While collections have a finite size, streams need not. Short-circuiting operations such as limit(n) or findFirst() can allow computations on infinite streams to complete in finite time.
* Consumable. The elements of a stream are only visited once during the life of a stream. Like an [Iterator](https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html), a new stream must be generated to revisit the same elements of the source.

Streams can be obtained in a number of ways. Some examples include:

* From a [Collection](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html) via the stream() and parallelStream() methods;
* From an array via [Arrays.stream(Object[])](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html" \l "stream-T:A-);
* From static factory methods on the stream classes, such as [Stream.of(Object[])](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html" \l "of-T...-), [IntStream.range(int, int)](https://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.html" \l "range-int-int-) or [Stream.iterate(Object, UnaryOperator)](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html" \l "iterate-T-java.util.function.UnaryOperator-);
* The lines of a file can be obtained from [BufferedReader.lines()](https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html" \l "lines--);
* Streams of file paths can be obtained from methods in [Files](https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html);
* Streams of random numbers can be obtained from [Random.ints()](https://docs.oracle.com/javase/8/docs/api/java/util/Random.html" \l "ints--);
* Numerous other stream-bearing methods in the JDK, including [BitSet.stream()](https://docs.oracle.com/javase/8/docs/api/java/util/BitSet.html#stream--), [Pattern.splitAsStream(java.lang.CharSequence)](https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html#splitAsStream-java.lang.CharSequence-), and [JarFile.stream()](https://docs.oracle.com/javase/8/docs/api/java/util/jar/JarFile.html" \l "stream--).

Additional stream sources can be provided by third-party libraries using [these techniques](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#StreamSources).

## Stream operations and pipelines

Stream operations are divided into intermediate and terminal operations, and are combined to form stream pipelines. A stream pipeline consists of a source (such as a Collection, an array, a generator function, or an I/O channel); followed by zero or more intermediate operations such as Stream.filter or Stream.map; and a terminal operation such as Stream.forEach or Stream.reduce.

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. Traversal of the pipeline source does not begin until the terminal operation of the pipeline is executed.

Terminal operations, such as Stream.forEach or IntStream.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; if you need to traverse the same data source again, you must return to the data source to get a new stream. In almost all cases, terminal operations are eager, completing their traversal of the data source and processing of the pipeline before returning. Only the terminal operations iterator() and spliterator() are not; these are provided as an "escape hatch" to enable arbitrary client-controlled pipeline traversals in the event that the existing operations are not sufficient to the task.

Processing streams lazily allows for significant efficiencies; in a pipeline such as the filter-map-sum example above, filtering, mapping, and summing can be fused into a single pass on the data, with minimal intermediate state. Laziness also allows avoiding examining all the data when it is not necessary; for operations such as "find the first string longer than 1000 characters", it is only necessary to examine just enough strings to find one that has the desired characteristics without examining all of the strings available from the source. (This behavior becomes even more important when the input stream is infinite and not merely large.)

Intermediate operations are further divided into stateless and stateful operations. Stateless operations, such as filter and map, retain no state from previously seen element when processing a new element -- each element can be processed independently of operations on other elements. Stateful operations, such as distinct and sorted, may incorporate state from previously seen elements when processing new elements.

Stateful operations may need to process the entire input before producing a result. For example, one cannot produce any results from sorting a stream until one has seen all elements of the stream. As a result, under parallel computation, some pipelines containing stateful intermediate operations may require multiple passes on the data or may need to buffer significant data. Pipelines containing exclusively stateless intermediate operations can be processed in a single pass, whether sequential or parallel, with minimal data buffering.

Further, some operations are deemed short-circuiting operations. An intermediate operation is short-circuiting if, when presented with infinite input, it may produce a finite stream as a result. A terminal operation is short-circuiting if, when presented with infinite input, it may terminate in finite time. Having a short-circuiting operation in the pipeline is a necessary, but not sufficient, condition for the processing of an infinite stream to terminate normally in finite time.

### *Parallelism*

Processing elements with an explicit for-loop is inherently serial. Streams facilitate parallel execution by reframing the computation as a pipeline of aggregate operations, rather than as imperative operations on each individual element. All streams operations can execute either in serial or in parallel. The stream implementations in the JDK create serial streams unless parallelism is explicitly requested. For example, Collection has methods [Collection.stream()](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html" \l "stream--) and [Collection.parallelStream()](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html" \l "parallelStream--), which produce sequential and parallel streams respectively; other stream-bearing methods such as [IntStream.range(int, int)](https://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.html" \l "range-int-int-) produce sequential streams but these streams can be efficiently parallelized by invoking their [BaseStream.parallel()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/BaseStream.html" \l "parallel--) method. To execute the prior "sum of weights of widgets" query in parallel, we would do:

int sumOfWeights = widgets.parallelStream()

.filter(b -> b.getColor() == RED)

.mapToInt(b -> b.getWeight())

.sum();

The only difference between the serial and parallel versions of this example is the creation of the initial stream, using "parallelStream()" instead of "stream()". When the terminal operation is initiated, the stream pipeline is executed sequentially or in parallel depending on the orientation of the stream on which it is invoked. Whether a stream will execute in serial or parallel can be determined with the isParallel() method, and the orientation of a stream can be modified with the [BaseStream.sequential()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/BaseStream.html#sequential--) and [BaseStream.parallel()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/BaseStream.html#parallel--) operations. When the terminal operation is initiated, the stream pipeline is executed sequentially or in parallel depending on the mode of the stream on which it is invoked.

Except for operations identified as explicitly nondeterministic, such as findAny(), whether a stream executes sequentially or in parallel should not change the result of the computation.

Most stream operations accept parameters that describe user-specified behavior, which are often lambda expressions. To preserve correct behavior, these behavioral parameters must be non-interfering, and in most cases must be stateless. Such parameters are always instances of a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) such as [Function](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html), and are often lambda expressions or method references.

### *Non-interference*

Streams enable you to execute possibly-parallel aggregate operations over a variety of data sources, including even non-thread-safe collections such as ArrayList. This is possible only if we can prevent interference with the data source during the execution of a stream pipeline. Except for the escape-hatch operations iterator() and spliterator(), execution begins when the terminal operation is invoked, and ends when the terminal operation completes. For most data sources, preventing interference means ensuring that the data source is not modified at all during the execution of the stream pipeline. The notable exception to this are streams whose sources are concurrent collections, which are specifically designed to handle concurrent modification. Concurrent stream sources are those whose Spliterator reports the CONCURRENT characteristic.

Accordingly, behavioral parameters in stream pipelines whose source might not be concurrent should never modify the stream's data source. A behavioral parameter is said to interfere with a non-concurrent data source if it modifies, or causes to be modified, the stream's data source. The need for non-interference applies to all pipelines, not just parallel ones. Unless the stream source is concurrent, modifying a stream's data source during execution of a stream pipeline can cause exceptions, incorrect answers, or nonconformant behavior. For well-behaved stream sources, the source can be modified before the terminal operation commences and those modifications will be reflected in the covered elements. For example, consider the following code:

List<String> l = new ArrayList(Arrays.asList("one", "two"));

Stream<String> sl = l.stream();

l.add("three");

String s = sl.collect(joining(" "));

First a list is created consisting of two strings: "one"; and "two". Then a stream is created from that list. Next the list is modified by adding a third string: "three". Finally the elements of the stream are collected and joined together. Since the list was modified before the terminal collect operation commenced the result will be a string of "one two three". All the streams returned from JDK collections, and most other JDK classes, are well-behaved in this manner; for streams generated by other libraries, see [Low-level stream construction](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#StreamSources) for requirements for building well-behaved streams.

### *Stateless behaviors*

Stream pipeline results may be nondeterministic or incorrect if the behavioral parameters to the stream operations are stateful. A stateful lambda (or other object implementing the appropriate functional interface) is one whose result depends on any state which might change during the execution of the stream pipeline. An example of a stateful lambda is the parameter to map() in:

Set<Integer> seen = Collections.synchronizedSet(new HashSet<>());

stream.parallel().map(e -> { if (seen.add(e)) return 0; else return e; })...

Here, if the mapping operation is performed in parallel, the results for the same input could vary from run to run, due to thread scheduling differences, whereas, with a stateless lambda expression the results would always be the same.

Note also that attempting to access mutable state from behavioral parameters presents you with a bad choice with respect to safety and performance; if you do not synchronize access to that state, you have a data race and therefore your code is broken, but if you do synchronize access to that state, you risk having contention undermine the parallelism you are seeking to benefit from. The best approach is to avoid stateful behavioral parameters to stream operations entirely; there is usually a way to restructure the stream pipeline to avoid statefulness.

### *Side-effects*

Side-effects in behavioral parameters to stream operations are, in general, discouraged, as they can often lead to unwitting violations of the statelessness requirement, as well as other thread-safety hazards.

If the behavioral parameters do have side-effects, unless explicitly stated, there are no guarantees as to the [*visibility*](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility) of those side-effects to other threads, nor are there any guarantees that different operations on the "same" element within the same stream pipeline are executed in the same thread. Further, the ordering of those effects may be surprising. Even when a pipeline is constrained to produce a result that is consistent with the encounter order of the stream source (for example, IntStream.range(0,5).parallel().map(x -> x\*2).toArray() must produce [0, 2, 4, 6, 8]), no guarantees are made as to the order in which the mapper function is applied to individual elements, or in what thread any behavioral parameter is executed for a given element.

Many computations where one might be tempted to use side effects can be more safely and efficiently expressed without side-effects, such as using [reduction](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#Reduction) instead of mutable accumulators. However, side-effects such as using println() for debugging purposes are usually harmless. A small number of stream operations, such as forEach() and peek(), can operate only via side-effects; these should be used with care.

As an example of how to transform a stream pipeline that inappropriately uses side-effects to one that does not, the following code searches a stream of strings for those matching a given regular expression, and puts the matches in a list.

ArrayList<String> results = new ArrayList<>();

stream.filter(s -> pattern.matcher(s).matches())

.forEach(s -> results.add(s)); // Unnecessary use of side-effects!

This code unnecessarily uses side-effects. If executed in parallel, the non-thread-safety of ArrayList would cause incorrect results, and adding needed synchronization would cause contention, undermining the benefit of parallelism. Furthermore, using side-effects here is completely unnecessary; the forEach() can simply be replaced with a reduction operation that is safer, more efficient, and more amenable to parallelization:

List<String>results =

stream.filter(s -> pattern.matcher(s).matches())

.collect(Collectors.toList()); // No side-effects!

### *Ordering*

Streams may or may not have a defined encounter order. Whether or not a stream has an encounter order depends on the source and the intermediate operations. Certain stream sources (such as List or arrays) are intrinsically ordered, whereas others (such as HashSet) are not. Some intermediate operations, such as sorted(), may impose an encounter order on an otherwise unordered stream, and others may render an ordered stream unordered, such as [BaseStream.unordered()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/BaseStream.html" \l "unordered--). Further, some terminal operations may ignore encounter order, such as forEach().

If a stream is ordered, most operations are constrained to operate on the elements in their encounter order; if the source of a stream is a List containing [1, 2, 3], then the result of executing map(x -> x\*2) must be [2, 4, 6]. However, if the source has no defined encounter order, then any permutation of the values [2, 4, 6] would be a valid result.

For sequential streams, the presence or absence of an encounter order does not affect performance, only determinism. If a stream is ordered, repeated execution of identical stream pipelines on an identical source will produce an identical result; if it is not ordered, repeated execution might produce different results.

For parallel streams, relaxing the ordering constraint can sometimes enable more efficient execution. Certain aggregate operations, such as filtering duplicates (distinct()) or grouped reductions (Collectors.groupingBy()) can be implemented more efficiently if ordering of elements is not relevant. Similarly, operations that are intrinsically tied to encounter order, such as limit(), may require buffering to ensure proper ordering, undermining the benefit of parallelism. In cases where the stream has an encounter order, but the user does not particularly care about that encounter order, explicitly de-ordering the stream with [unordered()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/BaseStream.html#unordered--) may improve parallel performance for some stateful or terminal operations. However, most stream pipelines, such as the "sum of weight of blocks" example above, still parallelize efficiently even under ordering constraints.

## Reduction operations

A reduction operation (also called a fold) takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation, such as finding the sum or maximum of a set of numbers, or accumulating elements into a list. The streams classes have multiple forms of general reduction operations, called [reduce()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#reduce-java.util.function.BinaryOperator-) and [collect()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#collect-java.util.stream.Collector-), as well as multiple specialized reduction forms such as [sum()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.html#sum--), [max()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.html#max--), or [count()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.html#count--).

Of course, such operations can be readily implemented as simple sequential loops, as in:

int sum = 0;

for (int x : numbers) {

sum += x;

}

However, there are good reasons to prefer a reduce operation over a mutative accumulation such as the above. Not only is a reduction "more abstract" -- it operates on the stream as a whole rather than individual elements -- but a properly constructed reduce operation is inherently parallelizable, so long as the function(s) used to process the elements are [associative](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#Associativity) and [stateless](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#NonInterfering). For example, given a stream of numbers for which we want to find the sum, we can write:

int sum = numbers.stream().reduce(0, (x,y) -> x+y);

or:

int sum = numbers.stream().reduce(0, Integer::sum);

These reduction operations can run safely in parallel with almost no modification:

int sum = numbers.parallelStream().reduce(0, Integer::sum);

Reduction parallellizes well because the implementation can operate on subsets of the data in parallel, and then combine the intermediate results to get the final correct answer. (Even if the language had a "parallel for-each" construct, the mutative accumulation approach would still required the developer to provide thread-safe updates to the shared accumulating variable sum, and the required synchronization would then likely eliminate any performance gain from parallelism.) Using reduce() instead removes all of the burden of parallelizing the reduction operation, and the library can provide an efficient parallel implementation with no additional synchronization required.

The "widgets" examples shown earlier shows how reduction combines with other operations to replace for loops with bulk operations. If widgets is a collection of Widget objects, which have a getWeight method, we can find the heaviest widget with:

OptionalInt heaviest = widgets.parallelStream()

.mapToInt(Widget::getWeight)

.max();

In its more general form, a reduce operation on elements of type <T> yielding a result of type <U> requires three parameters:

<U> U reduce(U identity,

BiFunction<U, ? super T, U> accumulator,

BinaryOperator<U> combiner);

Here, the identity element is both an initial seed value for the reduction and a default result if there are no input elements. The accumulator function takes a partial result and the next element, and produces a new partial result. The combiner function combines two partial results to produce a new partial result. (The combiner is necessary in parallel reductions, where the input is partitioned, a partial accumulation computed for each partition, and then the partial results are combined to produce a final result.)

More formally, the identity value must be an identity for the combiner function. This means that for all u, combiner.apply(identity, u) is equal to u. Additionally, the combiner function must be [associative](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#Associativity) and must be compatible with the accumulator function: for all u and t, combiner.apply(u, accumulator.apply(identity, t)) must be equals() to accumulator.apply(u, t).

The three-argument form is a generalization of the two-argument form, incorporating a mapping step into the accumulation step. We could re-cast the simple sum-of-weights example using the more general form as follows:

int sumOfWeights = widgets.stream()

.reduce(0,

(sum, b) -> sum + b.getWeight())

Integer::sum);

though the explicit map-reduce form is more readable and therefore should usually be preferred. The generalized form is provided for cases where significant work can be optimized away by combining mapping and reducing into a single function.

### *Mutable reduction*

A mutable reduction operation accumulates input elements into a mutable result container, such as a Collection or StringBuilder, as it processes the elements in the stream.

If we wanted to take a stream of strings and concatenate them into a single long string, we could achieve this with ordinary reduction:

String concatenated = strings.reduce("", String::concat)

We would get the desired result, and it would even work in parallel. However, we might not be happy about the performance! Such an implementation would do a great deal of string copying, and the run time would be O(n^2) in the number of characters. A more performant approach would be to accumulate the results into a [StringBuilder](https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html" \o "class in java.lang), which is a mutable container for accumulating strings. We can use the same technique to parallelize mutable reduction as we do with ordinary reduction.

The mutable reduction operation is called [collect()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#collect-java.util.stream.Collector-), as it collects together the desired results into a result container such as a Collection. A collect operation requires three functions: a supplier function to construct new instances of the result container, an accumulator function to incorporate an input element into a result container, and a combining function to merge the contents of one result container into another. The form of this is very similar to the general form of ordinary reduction:

<R> R collect(Supplier<R> supplier,

BiConsumer<R, ? super T> accumulator,

BiConsumer<R, R> combiner);

As with reduce(), a benefit of expressing collect in this abstract way is that it is directly amenable to parallelization: we can accumulate partial results in parallel and then combine them, so long as the accumulation and combining functions satisfy the appropriate requirements. For example, to collect the String representations of the elements in a stream into an ArrayList, we could write the obvious sequential for-each form:

ArrayList<String> strings = new ArrayList<>();

for (T element : stream) {

strings.add(element.toString());

}

Or we could use a parallelizable collect form:

ArrayList<String> strings = stream.collect(() -> new ArrayList<>(),

(c, e) -> c.add(e.toString()),

(c1, c2) -> c1.addAll(c2));

or, pulling the mapping operation out of the accumulator function, we could express it more succinctly as:

List<String> strings = stream.map(Object::toString)

.collect(ArrayList::new, ArrayList::add, ArrayList::addAll);

Here, our supplier is just the [ArrayList constructor](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html" \l "ArrayList--), the accumulator adds the stringified element to an ArrayList, and the combiner simply uses [addAll](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html" \l "addAll-java.util.Collection-) to copy the strings from one container into the other.

The three aspects of collect -- supplier, accumulator, and combiner -- are tightly coupled. We can use the abstraction of a [Collector](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html) to capture all three aspects. The above example for collecting strings into a List can be rewritten using a standard Collector as:

List<String> strings = stream.map(Object::toString)

.collect(Collectors.toList());

Packaging mutable reductions into a Collector has another advantage: composability. The class [Collectors](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html) contains a number of predefined factories for collectors, including combinators that transform one collector into another. For example, suppose we have a collector that computes the sum of the salaries of a stream of employees, as follows:

Collector<Employee, ?, Integer> summingSalaries

= Collectors.summingInt(Employee::getSalary);

(The ? for the second type parameter merely indicates that we don't care about the intermediate representation used by this collector.) If we wanted to create a collector to tabulate the sum of salaries by department, we could reuse summingSalaries using [groupingBy](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html" \l "groupingBy-java.util.function.Function-java.util.stream.Collector-):

Map<Department, Integer> salariesByDept

= employees.stream().collect(Collectors.groupingBy(Employee::getDepartment,

summingSalaries));

As with the regular reduction operation, collect() operations can only be parallelized if appropriate conditions are met. For any partially accumulated result, combining it with an empty result container must produce an equivalent result. That is, for a partially accumulated result p that is the result of any series of accumulator and combiner invocations, p must be equivalent to combiner.apply(p, supplier.get()).

Further, however the computation is split, it must produce an equivalent result. For any input elements t1 and t2, the results r1 and r2 in the computation below must be equivalent:

A a1 = supplier.get();

accumulator.accept(a1, t1);

accumulator.accept(a1, t2);

R r1 = finisher.apply(a1); // result without splitting

A a2 = supplier.get();

accumulator.accept(a2, t1);

A a3 = supplier.get();

accumulator.accept(a3, t2);

R r2 = finisher.apply(combiner.apply(a2, a3)); // result with splitting

Here, equivalence generally means according to [Object.equals(Object)](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html" \l "equals-java.lang.Object-). but in some cases equivalence may be relaxed to account for differences in order.

### *Reduction, concurrency, and ordering*

With some complex reduction operations, for example a collect() that produces a Map, such as:

Map<Buyer, List<Transaction>> salesByBuyer

= txns.parallelStream()

.collect(Collectors.groupingBy(Transaction::getBuyer));

it may actually be counterproductive to perform the operation in parallel. This is because the combining step (merging one Map into another by key) can be expensive for some Map implementations.

Suppose, however, that the result container used in this reduction was a concurrently modifiable collection -- such as a [ConcurrentHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html" \o "class in java.util.concurrent). In that case, the parallel invocations of the accumulator could actually deposit their results concurrently into the same shared result container, eliminating the need for the combiner to merge distinct result containers. This potentially provides a boost to the parallel execution performance. We call this a concurrent reduction.

A [Collector](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html) that supports concurrent reduction is marked with the [Collector.Characteristics.CONCURRENT](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.Characteristics.html" \l "CONCURRENT) characteristic. However, a concurrent collection also has a downside. If multiple threads are depositing results concurrently into a shared container, the order in which results are deposited is non-deterministic. Consequently, a concurrent reduction is only possible if ordering is not important for the stream being processed. The [Stream.collect(Collector)](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html" \l "collect-java.util.stream.Collector-) implementation will only perform a concurrent reduction if

* The stream is parallel;
* The collector has the [Collector.Characteristics.CONCURRENT](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.Characteristics.html" \l "CONCURRENT) characteristic, and;
* Either the stream is unordered, or the collector has the [Collector.Characteristics.UNORDERED](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.Characteristics.html" \l "UNORDERED) characteristic.

You can ensure the stream is unordered by using the [BaseStream.unordered()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/BaseStream.html" \l "unordered--) method. For example:

Map<Buyer, List<Transaction>> salesByBuyer

= txns.parallelStream()

.unordered()

.collect(groupingByConcurrent(Transaction::getBuyer));

(where [Collectors.groupingByConcurrent(java.util.function.Function<? super T, ? extends K>)](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html#groupingByConcurrent-java.util.function.Function-) is the concurrent equivalent of groupingBy).

Note that if it is important that the elements for a given key appear in the order they appear in the source, then we cannot use a concurrent reduction, as ordering is one of the casualties of concurrent insertion. We would then be constrained to implement either a sequential reduction or a merge-based parallel reduction.

### *Associativity*

An operator or function op is associative if the following holds:

(a op b) op c == a op (b op c)

The importance of this to parallel evaluation can be seen if we expand this to four terms:

a op b op c op d == (a op b) op (c op d)

So we can evaluate (a op b) in parallel with (c op d), and then invoke op on the results.

Examples of associative operations include numeric addition, min, and max, and string concatenation.

## Low-level stream construction

So far, all the stream examples have used methods like [Collection.stream()](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html" \l "stream--) or [Arrays.stream(Object[])](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html" \l "stream-T:A-) to obtain a stream. How are those stream-bearing methods implemented?

The class [StreamSupport](https://docs.oracle.com/javase/8/docs/api/java/util/stream/StreamSupport.html" \o "class in java.util.stream) has a number of low-level methods for creating a stream, all using some form of a [Spliterator](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html" \o "interface in java.util). A spliterator is the parallel analogue of an [Iterator](https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html); it describes a (possibly infinite) collection of elements, with support for sequentially advancing, bulk traversal, and splitting off some portion of the input into another spliterator which can be processed in parallel. At the lowest level, all streams are driven by a spliterator.

There are a number of implementation choices in implementing a spliterator, nearly all of which are tradeoffs between simplicity of implementation and runtime performance of streams using that spliterator. The simplest, but least performant, way to create a spliterator is to create one from an iterator using [Spliterators.spliteratorUnknownSize(java.util.Iterator, int)](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterators.html" \l "spliteratorUnknownSize-java.util.Iterator-int-). While such a spliterator will work, it will likely offer poor parallel performance, since we have lost sizing information (how big is the underlying data set), as well as being constrained to a simplistic splitting algorithm.

A higher-quality spliterator will provide balanced and known-size splits, accurate sizing information, and a number of other [characteristics](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#characteristics--) of the spliterator or data that can be used by implementations to optimize execution.

Spliterators for mutable data sources have an additional challenge; timing of binding to the data, since the data could change between the time the spliterator is created and the time the stream pipeline is executed. Ideally, a spliterator for a stream would report a characteristic of IMMUTABLE or CONCURRENT; if not it should be [late-binding](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#binding). If a source cannot directly supply a recommended spliterator, it may indirectly supply a spliterator using a Supplier, and construct a stream via the Supplier-accepting versions of [stream()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/StreamSupport.html#stream-java.util.function.Supplier-int-boolean-). The spliterator is obtained from the supplier only after the terminal operation of the stream pipeline commences.

These requirements significantly reduce the scope of potential interference between mutations of the stream source and execution of stream pipelines. Streams based on spliterators with the desired characteristics, or those using the Supplier-based factory forms, are immune to modifications of the data source prior to commencement of the terminal operation (provided the behavioral parameters to the stream operations meet the required criteria for non-interference and statelessness). See [Non-Interference](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#NonInterference) for more details.