**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* OPTIONAL<T> \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Overview**

In this tutorial, we're going to show the *Optional* class that was introduced in Java 8.

The purpose of the class is to provide a type-level solution for representing optional values instead of *null* references.

To get a deeper understanding of why we should care about the *Optional* class, take a look at [the official Oracle article](http://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html).

**Further reading:**

[**Java Optional as Return Type**](https://www.baeldung.com/java-optional-return)

Learn the best practices and when to return the Optional type in Java.

[**Read more**](https://www.baeldung.com/java-optional-return) →

[**Java Optional – orElse() vs orElseGet()**](https://www.baeldung.com/java-optional-or-else-vs-or-else-get)

Explore the differences between Optional orElse() and OrElseGet() methods.

[**Read more**](https://www.baeldung.com/java-optional-or-else-vs-or-else-get) →

[**Filtering a Stream of Optionals in Java**](https://www.baeldung.com/java-filter-stream-of-optional)

A quick and practical guide to filtering Streams of Optionals in Java 8 and Java 9

[**Read more**](https://www.baeldung.com/java-filter-stream-of-optional) →

**2. Creating *Optional* Objects**

There are several ways of creating *Optional* objects.

To create an empty *Optional* object, we simply need to use its *empty()* static method:

@Test

**public** **void** **whenCreatesEmptyOptional\_thenCorrect**() {

Optional<String> empty = Optional.empty();

assertFalse(empty.isPresent());

}

Note that we used the *isPresent()* method to check if there is a value inside the *Optional* object. A value is present only if we have created *Optional* with a non-*null* value. We'll look at the *isPresent()* method in the next section.

We can also create an *Optional* object with the static method *of()*:

@Test

**public** **void** **givenNonNull\_whenCreatesNonNullable\_thenCorrect**() {

String name = "baeldung";

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

assertTrue(opt.isPresent());

}

However, the argument passed to the *of()* method can't be *null.* Otherwise, we'll get a *NullPointerException*:

@Test(expected = NullPointerException.class)

**public** **void** **givenNull\_whenThrowsErrorOnCreate\_thenCorrect**() {

String name = **null**;

Optional.of(name);

}

But in case we expect some *null* values, we can use the *ofNullable()* method:

@Test

**public** **void** **givenNonNull\_whenCreatesNullable\_thenCorrect**() {

String name = "baeldung";

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

assertTrue(opt.isPresent());

}

By doing this, if we pass in a *null* reference, it doesn't throw an exception but rather returns an empty *Optional* object:

@Test

**public** **void** **givenNull\_whenCreatesNullable\_thenCorrect**() {

String name = **null**;

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

assertFalse(opt.isPresent());

}

**3. Checking Value Presence: *isPresent()*and *isEmpty()***

When we have an *Optional* object returned from a method or created by us, we can check if there is a value in it or not with the *isPresent()* method:

@Test

**public** **void** **givenOptional\_whenIsPresentWorks\_thenCorrect**() {

Optional<String> opt = Optional.of("Baeldung");

assertTrue(opt.isPresent());

opt = Optional.ofNullable(**null**);

assertFalse(opt.isPresent());

}

This method returns *true* if the wrapped value is not *null.*

Also, as of Java 11, we can do the opposite with the *isEmpty*method:

@Test

**public** **void** **givenAnEmptyOptional\_thenIsEmptyBehavesAsExpected**() {

Optional<String> opt = Optional.of("Baeldung");

assertFalse(opt.isEmpty());

opt = Optional.ofNullable(**null**);

assertTrue(opt.isEmpty());

}

**4. Conditional Action With *ifPresent()***

The *ifPresent()* method enables us to run some code on the wrapped value if it's found to be non-*null*. Before *Optional*, we'd do:

**if**(name != **null**) {

System.out.println(name.length());

}

This code checks if the name variable is *null* or not before going ahead to execute some code on it. This approach is lengthy, and that's not the only problem — it's also prone to error.

Indeed, what guarantees that after printing that variable, we won't use it again and then **forget to perform the null check?**

**This can result in a *NullPointerException* at runtime if a null value finds its way into that code.** When a program fails due to input issues, it's often a result of poor programming practices.

*Optional* makes us deal with nullable values explicitly as a way of enforcing good programming practices.

Let's now look at how the above code could be refactored in Java 8.

In typical functional programming style, we can execute perform an action on an object that is actually present:

@Test

**public** **void** **givenOptional\_whenIfPresentWorks\_thenCorrect**() {

Optional<String> opt = Optional.of("baeldung");

opt.ifPresent(name -> System.out.println(name.length()));

}

In the above example, we use only two lines of code to replace the five that worked in the first example: one line to wrap the object into an *Optional* object and the next to perform implicit validation as well as execute the code.

**5. Default Value With *orElse()***

The *orElse()* method is used to retrieve the value wrapped inside an *Optional* instance. It takes one parameter, which acts as a default value. The *orElse()*method returns the wrapped value if it's present, and its argument otherwise:

@Test

**public** **void** **whenOrElseWorks\_thenCorrect**() {

String nullName = **null**;

String name = Optional.ofNullable(nullName).orElse("john");

assertEquals("john", name);

}

**6. Default Value With *orElseGet()***

The *orElseGet()* method is similar to *orElse()*. However, instead of taking a value to return if the *Optional* value is not present, it takes a supplier functional interface, which is invoked and returns the value of the invocation:

@Test

**public** **void** **whenOrElseGetWorks\_thenCorrect**() {

String nullName = **null**;

String name = Optional.ofNullable(nullName).orElseGet(() -> "john");

assertEquals("john", name);

}

**7. Difference Between *orElse* and *orElseGet()***

To a lot of programmers who are new to *Optional* or Java 8, the difference between *orElse()* and *orElseGet()* is not clear. As a matter of fact, these two methods give the impression that they overlap each other in functionality.

However, there's a subtle but very important difference between the two that can affect the performance of our code drastically if not well understood.

Let's create a method called *getMyDefault()* in the test class, which takes no arguments and returns a default value:

**public** String **getMyDefault**() {

System.out.println("Getting Default Value");

**return** "Default Value";

}

Let's see two tests and observe their side effects to establish both where *orElse()* and *orElseGet()*overlap and where they differ:

@Test

**public** **void** **whenOrElseGetAndOrElseOverlap\_thenCorrect**() {

String text = **null**;

String defaultText = Optional.ofNullable(text).orElseGet(**this**::getMyDefault);

assertEquals("Default Value", defaultText);

defaultText = Optional.ofNullable(text).orElse(getMyDefault());

assertEquals("Default Value", defaultText);

}

In the above example, we wrap a null text inside an *Optional* object and attempt to get the wrapped value using each of the two approaches.

The side effect is:

Getting default value...

Getting default value...

The *getMyDefault()* method is called in each case. It so happens that **when the wrapped value is not present, then both *orElse()* and *orElseGet()* work exactly the same way.**

Now let's run another test where the value is present, and ideally, the default value should not even be created:

@Test

**public** **void** **whenOrElseGetAndOrElseDiffer\_thenCorrect**() {

String text = "Text present";

System.out.println("Using orElseGet:");

String defaultText

= Optional.ofNullable(text).orElseGet(**this**::getMyDefault);

assertEquals("Text present", defaultText);

System.out.println("Using orElse:");

defaultText = Optional.ofNullable(text).orElse(getMyDefault());

assertEquals("Text present", defaultText);

}

In the above example, we are no longer wrapping a *null* value, and the rest of the code remains the same.

Now let's take a look at the side effect of running this code:

Using orElseGet:

Using orElse:

Getting default value...

Notice that when using *orElseGet()* to retrieve the wrapped value, the *getMyDefault()* method is not even invoked since the contained value is present.

However, when using *orElse()*, whether the wrapped value is present or not, the default object is created. So in this case, we have just created one redundant object that is never used.

In this simple example, there is no significant cost to creating a default object, as the JVM knows how to deal with such. **However, when a method such as *getMyDefault()* has to make a web service call or even query a database, the cost becomes very obvious.**

**8. Exceptions With *orElseThrow()***

The *orElseThrow()* method follows from *orElse()* and *orElseGet()* and adds a new approach for handling an absent value.

Instead of returning a default value when the wrapped value is not present, it throws an exception:

@Test(expected = IllegalArgumentException.class)

**public** **void** **whenOrElseThrowWorks\_thenCorrect**() {

String nullName = **null**;

String name = Optional.ofNullable(nullName).orElseThrow(

IllegalArgumentException::**new**);

}

Method references in Java 8 come in handy here, to pass in the exception constructor.

**Java 10 introduced a simplified no-arg version of *orElseThrow()* method**. In case of an empty *Optional* it throws a *NoSuchElementException*:

@Test(expected = NoSuchElementException.class)

**public** **void** **whenNoArgOrElseThrowWorks\_thenCorrect**() {

String nullName = **null**;

String name = Optional.ofNullable(nullName).orElseThrow();

}

**9. Returning Value With *get()***

The final approach for retrieving the wrapped value is the *get()* method:

@Test

**public** **void** **givenOptional\_whenGetsValue\_thenCorrect**() {

Optional<String> opt = Optional.of("baeldung");

String name = opt.get();

assertEquals("baeldung", name);

}

However, unlike the previous three approaches, *get()* can only return a value if the wrapped object is not *null*; otherwise, it throws a no such element exception:

@Test(expected = NoSuchElementException.class)

**public** **void** **givenOptionalWithNull\_whenGetThrowsException\_thenCorrect**() {

Optional<String> opt = Optional.ofNullable(**null**);

String name = opt.get();

}

This is the major flaw of the *get()* method. Ideally, *Optional* should help us avoid such unforeseen exceptions. Therefore, this approach works against the objectives of *Optional* and will probably be deprecated in a future release.

So, it's advisable to use the other variants that enable us to prepare for and explicitly handle the *null* case.

**10. Conditional Return With *filter()***

We can run an inline test on our wrapped value with the *filter* method. It takes a predicate as an argument and returns an *Optional* object. If the wrapped value passes testing by the predicate, then the *Optional* is returned as-is.

However, if the predicate returns *false*, then it will return an empty *Optional*:

@Test

**public** **void** **whenOptionalFilterWorks\_thenCorrect**() {

Integer year = 2016;

Optional<Integer> yearOptional = Optional.of(year);

**boolean** is2016 = yearOptional.filter(y -> y == 2016).isPresent();

assertTrue(is2016);

**boolean** is2017 = yearOptional.filter(y -> y == 2017).isPresent();

assertFalse(is2017);

}

The *filter* method is normally used this way to reject wrapped values based on a predefined rule. We could use it to reject a wrong email format or a password that is not strong enough.

Let's look at another meaningful example. Say we want to buy a modem, and we only care about its price.

We receive push notifications on modem prices from a certain site and store these in objects:

**public** **class** **Modem** {

**private** Double price;

**public** **Modem**(Double price) {

**this**.price = price;

}

// standard getters and setters

}

We then feed these objects to some code whose sole purpose is to check if the modem price is within our budget range.

Let's now take a look at the code without *Optional*:

**public** **boolean** **priceIsInRange1**(Modem modem) {

**boolean** isInRange = **false**;

**if** (modem != **null** && modem.getPrice() != **null**

&& (modem.getPrice() >= 10

&& modem.getPrice() <= 15)) {

isInRange = **true**;

}

**return** isInRange;

}

Pay attention to how much code we have to write to achieve this, especially in the *if* condition. The only part of the *if* condition that is critical to the application is the last price-range check; the rest of the checks are defensive:

@Test

**public** **void** **whenFiltersWithoutOptional\_thenCorrect**() {

assertTrue(priceIsInRange1(**new** Modem(10.0)));

assertFalse(priceIsInRange1(**new** Modem(9.9)));

assertFalse(priceIsInRange1(**new** Modem(**null**)));

assertFalse(priceIsInRange1(**new** Modem(15.5)));

assertFalse(priceIsInRange1(**null**));

}

Apart from that, it's possible to forget about the null checks over a long day without getting any compile-time errors.

Now let's look at a variant with *Optional#filter*:

**public** **boolean** **priceIsInRange2**(Modem modem2) {

**return** Optional.ofNullable(modem2)

.map(Modem::getPrice)

.filter(p -> p >= 10)

.filter(p -> p <= 15)

.isPresent();

}

**The *map* call is simply used to transform a value to some other value.** Keep in mind that this operation does not modify the original value.

In our case, we are obtaining a price object from the *Model* class. We will look at the *map()* method in detail in the next section.

First of all, if a *null* object is passed to this method, we don't expect any problem.

Secondly, the only logic we write inside its body is exactly what the method name describes — price-range check. *Optional* takes care of the rest:

@Test

**public** **void** **whenFiltersWithOptional\_thenCorrect**() {

assertTrue(priceIsInRange2(**new** Modem(10.0)));

assertFalse(priceIsInRange2(**new** Modem(9.9)));

assertFalse(priceIsInRange2(**new** Modem(**null**)));

assertFalse(priceIsInRange2(**new** Modem(15.5)));

assertFalse(priceIsInRange2(**null**));

}

The previous approach promises to check price range but has to do more than that to defend against its inherent fragility. Therefore, we can use the *filter* method to replace unnecessary *if* statements and reject unwanted values.

**11. Transforming Value With *map()***

In the previous section, we looked at how to reject or accept a value based on a filter.

We can use a similar syntax to transform the *Optional* value with the *map()*method:

@Test

**public** **void** **givenOptional\_whenMapWorks\_thenCorrect**() {

List<String> companyNames = Arrays.asList(

"paypal", "oracle", "", "microsoft", "", "apple");

Optional<List<String>> listOptional = Optional.of(companyNames);

**int** size = listOptional

.map(List::size)

.orElse(0);

assertEquals(6, size);

}

In this example, we wrap a list of strings inside an *Optional* object and use its *map* method to perform an action on the contained list. The action we perform is to retrieve the size of the list.

The *map* method returns the result of the computation wrapped inside *Optional*. We then have to call an appropriate method on the returned *Optional* to retrieve its value.

Notice that the *filter* method simply performs a check on the value and returns a *boolean*. The *map* method however takes the existing value, performs a computation using this value, and returns the result of the computation wrapped in an *Optional* object:

@Test

**public** **void** **givenOptional\_whenMapWorks\_thenCorrect2**() {

String name = "baeldung";

Optional<String> nameOptional = Optional.of(name);

**int** len = nameOptional

.map(String::length)

.orElse(0);

assertEquals(8, len);

}

We can chain *map* and *filter* together to do something more powerful.

Let's assume we want to check the correctness of a password input by a user. We can clean the password using a *map* transformation and check its correctness using a *filter*:

@Test

**public** **void** **givenOptional\_whenMapWorksWithFilter\_thenCorrect**() {

String password = " password ";

Optional<String> passOpt = Optional.of(password);

**boolean** correctPassword = passOpt.filter(

pass -> pass.equals("password")).isPresent();

assertFalse(correctPassword);

correctPassword = passOpt

.map(String::trim)

.filter(pass -> pass.equals("password"))

.isPresent();

assertTrue(correctPassword);

}

As we can see, without first cleaning the input, it will be filtered out — yet users may take for granted that leading and trailing spaces all constitute input. So, we transform a dirty password into a clean one with a *map* before filtering out incorrect ones.

**12. Transforming Value With *flatMap()***

Just like the *map()* method, we also have the *flatMap()* method as an alternative for transforming values. The difference is that *map* transforms values only when they are unwrapped whereas *flatMap* takes a wrapped value and unwraps it before transforming it.

Previously, we created simple *String* and *Integer* objects for wrapping in an *Optional* instance. However, frequently, we will receive these objects from an accessor of a complex object.

To get a clearer picture of the difference, let's have a look at a *Person* object that takes a person's details such as name, age and password:

**public** **class** **Person** {

**private** String name;

**private** **int** age;

**private** String password;

**public** Optional<String> **getName**() {

**return** Optional.ofNullable(name);

}

**public** Optional<Integer> **getAge**() {

**return** Optional.ofNullable(age);

}

**public** Optional<String> **getPassword**() {

**return** Optional.ofNullable(password);

}

// normal constructors and setters

}

We would normally create such an object and wrap it in an *Optional* object just like we did with String.

Alternatively, it can be returned to us by another method call:

Person person = **new** Person("john", 26);

Optional<Person> personOptional = Optional.of(person);

Notice now that when we wrap a *Person* object, it will contain nested *Optional* instances:

@Test

**public** **void** **givenOptional\_whenFlatMapWorks\_thenCorrect2**() {

Person person = **new** Person("john", 26);

Optional<Person> personOptional = Optional.of(person);

Optional<Optional<String>> nameOptionalWrapper

= personOptional.map(Person::getName);

Optional<String> nameOptional

= nameOptionalWrapper.orElseThrow(IllegalArgumentException::**new**);

String name1 = nameOptional.orElse("");

assertEquals("john", name1);

String name = personOptional

.flatMap(Person::getName)

.orElse("");

assertEquals("john", name);

}

Here, we're trying to retrieve the name attribute of the *Person* object to perform an assertion.

Note how we achieve this with *map()* method in the third statement, and then notice how we do the same with *flatMap()* method afterwards.

The *Person::getName* method reference is similar to the *String::trim* call we had in the previous section for cleaning up a password.

The only difference is that *getName()* returns an *Optional* rather than a String as did the *trim()* operation. This, coupled with the fact that a *map* transformation wraps the result in an *Optional* object, leads to a nested *Optional*.

While using *map()* method, therefore, we need to add an extra call to retrieve the value before using the transformed value. This way, the *Optional* wrapper will be removed. This operation is performed implicitly when using *flatMap*.

**13. Chaining *Optional*s in Java 8**

Sometimes, we may need to get the first non-empty *Optional* object from a number of *Optional*s. In such cases, it would be very convenient to use a method like *orElseOptional()*. Unfortunately, such operation is not directly supported in Java 8.

Let's first introduce a few methods that we'll be using throughout this section:

**private** Optional<String> **getEmpty**() {

**return** Optional.empty();

}

**private** Optional<String> **getHello**() {

**return** Optional.of("hello");

}

**private** Optional<String> **getBye**() {

**return** Optional.of("bye");

}

**private** Optional<String> **createOptional**(String input) {

**if** (input == **null** || "".equals(input) || "empty".equals(input)) {

**return** Optional.empty();

}

**return** Optional.of(input);

}

In order to chain several *Optional* objects and get the first non-empty one in Java 8, we can use the *Stream* API:

@Test

**public** **void** **givenThreeOptionals\_whenChaining\_thenFirstNonEmptyIsReturned**() {

Optional<String> found = Stream.of(getEmpty(), getHello(), getBye())

.filter(Optional::isPresent)

.map(Optional::get)

.findFirst();

assertEquals(getHello(), found);

}

The downside of this approach is that all of our *get*methods are always executed, regardless of where a non-empty *Optional* appears in the *Stream*.

If we want to lazily evaluate the methods passed to *Stream.of()*, we need to use the method reference and the *Supplier* interface:

@Test

**public** **void** **givenThreeOptionals\_whenChaining\_thenFirstNonEmptyIsReturnedAndRestNotEvaluated**() {

Optional<String> found =

Stream.<Supplier<Optional<String>>>of(**this**::getEmpty, **this**::getHello, **this**::getBye)

.map(Supplier::get)

.filter(Optional::isPresent)

.map(Optional::get)

.findFirst();

assertEquals(getHello(), found);

}

In case we need to use methods that take arguments, we have to resort to lambda expressions:

@Test

**public** **void** **givenTwoOptionalsReturnedByOneArgMethod\_whenChaining\_thenFirstNonEmptyIsReturned**() {

Optional<String> found = Stream.<Supplier<Optional<String>>>of(

() -> createOptional("empty"),

() -> createOptional("hello")

)

.map(Supplier::get)

.filter(Optional::isPresent)

.map(Optional::get)

.findFirst();

assertEquals(createOptional("hello"), found);

}

Often, we'll want to return a default value in case all of the chained *Optional*s are empty. We can do so just by adding a call to *orElse()* or *orElseGet()*:

@Test

**public** **void** **givenTwoEmptyOptionals\_whenChaining\_thenDefaultIsReturned**() {

String found = Stream.<Supplier<Optional<String>>>of(

() -> createOptional("empty"),

() -> createOptional("empty")

)

.map(Supplier::get)

.filter(Optional::isPresent)

.map(Optional::get)

.findFirst()

.orElseGet(() -> "default");

assertEquals("default", found);

}

**14. JDK 9 *Optional* API**

The release of Java 9 added even more new methods to the *Optional* API:

* *or()* method for providing a supplier that creates an alternative *Optional*
* *ifPresentOrElse()* method that allows executing an action if the *Optional* is present or another action if not
* *stream()* method for converting an *Optional* to a *Stream*

Here is the complete article for [further reading](https://www.baeldung.com/java-9-optional).

**15. Misuse of *Optional*s**

Finally, let's see a tempting, however dangerous, way to use *Optional*s: passing an *Optional* parameter to a method.

Imagine we have a list of *Person* and we want a method to search through that list for people with a given name. Also, we would like that method to match entries with at least a certain age, if it's specified.

With this parameter being optional, we come with this method:

**public** **static** List<Person> **search**(List<Person> people, String name, Optional<Integer> age) {

// Null checks for people and name

**return** people.stream()

.filter(p -> p.getName().equals(name))

.filter(p -> p.getAge().get() >= age.orElse(0))

.collect(Collectors.toList());

}

Then we release our method, and another developer tries to use it:

someObject.search(people, "Peter", **null**);

Now the developer executes its code and gets a *NullPointerException.* **There we are, having to null check our optional parameter, which defeats our initial purpose in wanting to avoid this kind of situation.**

Here are some possibilities we could have done to handle it better:

**public** **static** List<Person> **search**(List<Person> people, String name, Integer age) {

// Null checks for people and name

**final** Integer ageFilter = age != **null** ? age : 0;

**return** people.stream()

.filter(p -> p.getName().equals(name))

.filter(p -> p.getAge().get() >= ageFilter)

.collect(Collectors.toList());

}

There, the parameter's still optional, but we handle it in only one check.

Another possibility would have been to **create two overloaded methods**:

**public** **static** List<Person> **search**(List<Person> people, String name) {

**return** doSearch(people, name, 0);

}

**public** **static** List<Person> **search**(List<Person> people, String name, **int** age) {

**return** doSearch(people, name, age);

}

**private** **static** List<Person> **doSearch**(List<Person> people, String name, **int** age) {

// Null checks for people and name

**return** people.stream()

.filter(p -> p.getName().equals(name))

.filter(p -> p.getAge().get().intValue() >= age)

.collect(Collectors.toList());

}

That way we offer a clear API with two methods doing different things (though they share the implementation).

So, there are solutions to avoid using *Optional*s as method parameters.**The intent of Java when releasing *Optional* was to use it as a return type**, thus indicating that a method could return an empty value. As a matter of fact, the practice of using *Optional* as a method parameter is even [discouraged by some code inspectors](https://rules.sonarsource.com/java/RSPEC-3553).

**16. *Optional* and Serialization**

As discussed above, *Optional* is meant to be used as a return type. Trying to use it as a field type is not recommended.

Additionally, **using *Optional* in a serializable class will result in a *NotSerializableException*.** Our article [Java *Optional* as Return Type](https://www.baeldung.com/java-optional-return) further addresses the issues with serialization.

And, in [Using *Optional* With Jackson](https://www.baeldung.com/jackson-optional), we explain what happens when *Optional* fields are serialized, along with a few workarounds to achieve the desired results.

One of the most welcome changes in Java 8 was the introduction of [lambda expressions](https://www.baeldung.com/java-8-lambda-expressions-tips), as these allow us to forego anonymous classes, greatly reducing boilerplate code and improving readability.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* METHOD REFERENCING \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Method References

You use [lambda expressions](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) to create anonymous methods. Sometimes, however, a lambda expression does nothing but call an existing method. In those cases, it's often clearer to refer to the existing method by name. Method references enable you to do this; they are compact, easy-to-read lambda expressions for methods that already have a name.

**Method references are a special type of lambda expressions**. They're often used to create simple lambda expressions by referencing existing methods.

There are four kinds of method references:

* Static methods
* Instance methods of particular objects
* Instance methods of an arbitrary object of a particular type
* Constructor

In this tutorial, we'll explore method references in Java.

**2. Reference to a Static Method**

We'll begin with a very simple example, capitalizing and printing a list of *Strings*:

List<String> messages = Arrays.asList("hello", "baeldung", "readers!");

We can achieve this by leveraging a simple lambda expression calling the *[StringUtils.capitalize()](https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/StringUtils.html" \l "capitalize-java.lang.String-)* method directly:

messages.forEach(word -> StringUtils.capitalize(word));

Or, we can use a method reference to simply refer to the *capitalize* static method:

messages.forEach(StringUtils::capitalize);

**Notice that method references always utilize the *::* operator.**

**3. Reference to an Instance Method of a Particular Object**

To demonstrate this type of method reference, let's consider two classes:

**public** **class** **Bicycle** {

**private** String brand;

**private** Integer frameSize;

// standard constructor, getters and setters

}

**public** **class** **BicycleComparator** **implements** **Comparator** {

@Override

**public** **int** **compare**(Bicycle a, Bicycle b) {

**return** a.getFrameSize().compareTo(b.getFrameSize());

}

}

And, let's create a *BicycleComparator* object to compare bicycle frame sizes:

BicycleComparator bikeFrameSizeComparator = **new** BicycleComparator();

We could use a lambda expression to sort bicycles by frame size, but we'd need to specify two bikes for comparison:

createBicyclesList().stream()

.sorted((a, b) -> bikeFrameSizeComparator.compare(a, b));

Instead, we can use a method reference to have the compiler handle parameter passing for us:

createBicyclesList().stream()

.sorted(bikeFrameSizeComparator::compare);

The method reference is much cleaner and more readable, as our intention is clearly shown by the code.

**4. Reference to an Instance Method of an Arbitrary Object of a Particular Type**

This type of method reference is similar to the previous example, but without having to create a custom object to perform the comparison.

Let's create an *Integer* list that we want to sort:

List<Integer> numbers = Arrays.asList(5, 3, 50, 24, 40, 2, 9, 18);

If we use a classic lambda expression, both parameters need to be explicitly passed, while using a method reference is much more straightforward:

numbers.stream()

.sorted((a, b) -> a.compareTo(b));

numbers.stream()

.sorted(Integer::compareTo);

Even though it's still a one-liner, the method reference is much easier to read and understand.

**5. Reference to a Constructor**

We can reference a constructor in the same way that we referenced a static method in our first example. The only difference is that we'll use the *new* keyword.

Let's create a *Bicycle* array out of a *String* list with different brands:

List<String> bikeBrands = Arrays.asList("Giant", "Scott", "Trek", "GT");

First, we'll add a new constructor to our *Bicycle* class:

**public** **Bicycle**(String brand) {

**this**.brand = brand;

**this**.frameSize = 0;

}

Next, we'll use our new constructor from a method reference and make a *Bicycle* array from the original *String* list:

bikeBrands.stream()

.map(Bicycle::**new**)

.toArray(Bicycle[]::**new**);

Notice how we called both *Bicycle* and *Array* constructors using a method reference, giving our code a much more concise and clear appearance.

**6. Additional Examples and Limitations**

As we've seen so far, method references are a great way to make our code and intentions very clear and readable. However, we can't use them to replace all kinds of lambda expressions since they have some limitations.

Their main limitation is a result of what's also their biggest strength: **the output from the previous expression needs to match the input parameters of the referenced method signature**.

Let's see an example of this limitation:

createBicyclesList().forEach(b -> System.out.printf(

"Bike brand is '%s' and frame size is '%d'%n",

b.getBrand(),

b.getFrameSize()));

This simple case can't be expressed with a method reference, because the *printf* method requires 3 parameters in our case, and using *createBicyclesList().forEach()* would only allow the method reference to infer one parameter (the *Bicycle* object).

**Finally, let's explore how to create a no-operation function that can be referenced from a lambda expression.**

In this case, we'll want to use a lambda expression without using its parameters.

First, let's create the *doNothingAtAll* method:

**private** **static** <T> **void** **doNothingAtAll**(Object... o) {

}

As it is a [varargs](https://www.baeldung.com/java-varargs) method, it will work in any lambda expression, no matter the referenced object or number of parameters inferred.

Now, let's see it in action:

createBicyclesList()

.forEach((o) -> MethodReferenceExamples.doNothingAtAll(o));

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* LAMBDA EXPRESSIONS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## ****Overview****

Now that Java 8 has reached wide usage, patterns and best practices have begun to emerge for some of its headlining features. In this tutorial, we'll take a closer look at functional interfaces and lambda expressions.

## Further reading:

## [Why Do Local Variables Used in Lambdas Have to Be Final or Effectively Final?](https://www.baeldung.com/java-lambda-effectively-final-local-variables)

Learn why Java requires local variables to be effectively final when used in a lambda.

[**Read more**](https://www.baeldung.com/java-lambda-effectively-final-local-variables) →

## [Java 8 – Powerful Comparison with Lambdas](https://www.baeldung.com/java-8-sort-lambda)

Elegant Sort in Java 8 - Lambda Expressions go right past syntactic sugar and bring powerful functional semantics into Java.

[**Read more**](https://www.baeldung.com/java-8-sort-lambda) →

## ****2. Prefer Standard Functional Interfaces****

Functional interfaces, which are gathered in the **[java.util.function](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/package-summary.html)** package, satisfy most developers' needs in providing target types for lambda expressions and method references. Each of these interfaces is general and abstract, making them easy to adapt to almost any lambda expression. Developers should explore this package before creating new functional interfaces.

Let's consider an interface Foo:

@FunctionalInterface

**public** **interface** **Foo** {

String **method**(String string);

}

In addition, we have a method add() in some class UseFoo, which takes this interface as a parameter:

**public** String **add**(String string, Foo foo) {

**return** foo.method(string);

}

To execute it, we would write:

Foo foo = parameter -> parameter + " from lambda";

String result = useFoo.add("Message ", foo);

If we look closer, we'll see that Foo is nothing more than a function that accepts one argument and produces a result. Java 8 already provides such an interface in [*Function<T,R>*](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Function.html) from the [java.util.function](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/package-summary.html) package.

Now we can remove interface Foo completely and change our code to:

**public** String **add**(String string, Function<String, String> fn) {

**return** fn.apply(string);

}

To execute this, we can write:

Function<String, String> fn =

parameter -> parameter + " from lambda";

String result = useFoo.add("Message ", fn);

## ****3. Use the****@FunctionalInterface****Annotation****

Now let's annotate our functional interfaces with [*@FunctionalInterface*](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/FunctionalInterface.html). At first, this annotation seems to be useless. Even without it, our interface will be treated as functional as long as it has just one abstract method.

However, let's imagine a big project with several interfaces; it's hard to control everything manually. An interface, which was designed to be functional, could accidentally be changed by adding another abstract method/methods, rendering it unusable as a functional interface.

By using the @FunctionalInterface annotation, the compiler will trigger an error in response to any attempt to break the predefined structure of a functional interface. It is also a very handy tool to make our application architecture easier to understand for other developers.

So we can use this:

@FunctionalInterface

**public** **interface** **Foo** {

String **method**();

}

Instead of just:

**public** **interface** **Foo** {

String **method**();

}

## ****4. Don't Overuse Default Methods in Functional Interfaces****

We can easily add default methods to the functional interface. This is acceptable to the functional interface contract as long as there is only one abstract method declaration:

@FunctionalInterface

**public** **interface** **Foo** {

String **method**(String string);

**default** **void** **defaultMethod**() {}

}

Functional interfaces can be extended by other functional interfaces if their abstract methods have the same signature:

@FunctionalInterface

**public** **interface** **FooExtended** **extends** **Baz**, **Bar** {}

@FunctionalInterface

**public** **interface** **Baz** {

String **method**(String string);

**default** String **defaultBaz**() {}

}

@FunctionalInterface

**public** **interface** **Bar** {

String **method**(String string);

**default** String **defaultBar**() {}

}

Just as with regular interfaces, **extending different functional interfaces with the same default method can be problematic**.

For example, let's add the defaultCommon() method to the Bar and Baz interfaces:

@FunctionalInterface

**public** **interface** **Baz** {

String **method**(String string);

**default** String **defaultBaz**() {}

**default** String **defaultCommon**(){}

}

@FunctionalInterface

**public** **interface** **Bar** {

String **method**(String string);

**default** String **defaultBar**() {}

**default** String **defaultCommon**() {}

}

In this case, we'll get a compile-time error:

interface FooExtended inherits unrelated defaults **for** defaultCommon() from types Baz and Bar...

To fix this, the defaultCommon() method should be overridden in the FooExtended interface. We can provide a custom implementation of this method; however, **we can also reuse the implementation from the parent interface**:

@FunctionalInterface

**public** **interface** **FooExtended** **extends** **Baz**, **Bar** {

@Override

**default** String **defaultCommon**() {

**return** Bar.**super**.defaultCommon();

}

}

It's important to note that we have to be careful. **Adding too many default methods to the interface is not a very good architectural decision.** This should be considered a compromise, only to be used when required for upgrading existing interfaces without breaking backward compatibility.

## ****5. Instantiate Functional Interfaces With Lambda Expressions****

The compiler will allow us to use an inner class to instantiate a functional interface; however, this can lead to very verbose code. We should prefer to use lambda expressions:

Foo foo = parameter -> parameter + " from Foo";

Over an inner class:

Foo fooByIC = **new** Foo() {

@Override

**public** String **method**(String string) {

**return** string + " from Foo";

}

};

**The lambda expression approach can be used for any suitable interface from old libraries.** It is usable for interfaces like Runnable, Comparator, and so on; **however, this doesn't mean that we should review our whole older code base and change everything.**

## ****6. Avoid Overloading Methods With Functional Interfaces as Parameters****

We should use methods with different names to avoid collisions:

**public** **interface** **Processor** {

String **process**(Callable<String> c) **throws** Exception;

String **process**(Supplier<String> s);

}

**public** **class** **ProcessorImpl** **implements** **Processor** {

@Override

**public** String **process**(Callable<String> c) **throws** Exception {

// implementation details

}

@Override

**public** String **process**(Supplier<String> s) {

// implementation details

}

}

At first glance, this seems reasonable, but any attempt to execute either of the ProcessorImpl‘s methods:

String result = processor.process(() -> "abc");

Ends with an error with the following message:

reference to process is ambiguous

both method **process**(java.util.concurrent.Callable<java.lang.String>)

in com.baeldung.java8.lambda.tips.ProcessorImpl

and method **process**(java.util.function.Supplier<java.lang.String>)

in com.baeldung.java8.lambda.tips.ProcessorImpl match

To solve this problem, we have two options. **The first option is to use methods with different names:**

String **processWithCallable**(Callable<String> c) **throws** Exception;

String **processWithSupplier**(Supplier<String> s);

**The second option is to perform casting manually,** which is not preferred:

String result = processor.process((Supplier<String>) () -> "abc");

## ****7. Don’t Treat Lambda Expressions as Inner Classes****

Despite our previous example, where we essentially substituted inner class by a lambda expression, the two concepts are different in an important way: scope.

When we use an inner class, it creates a new scope. We can hide local variables from the enclosing scope by instantiating new local variables with the same names. We can also use the keyword ***this*** inside our inner class as a reference to its instance.

Lambda expressions, however, work with enclosing scope. We can’t hide variables from the enclosing scope inside the lambda’s body. In this case, the keyword ***this*** is a reference to an enclosing instance.

For example, in the class UseFoo, we have an instance variable value:

**private** String value = "Enclosing scope value";

Then in some method of this class, place the following code and execute this method:

**public** String **scopeExperiment**() {

Foo fooIC = **new** Foo() {

String value = "Inner class value";

@Override

**public** String **method**(String string) {

**return** **this**.value;

}

};

String resultIC = fooIC.method("");

Foo fooLambda = parameter -> {

String value = "Lambda value";

**return** **this**.value;

};

String resultLambda = fooLambda.method("");

**return** "Results: resultIC = " + resultIC +

", resultLambda = " + resultLambda;

}

If we execute the scopeExperiment() method, we'll get the following result: Results: resultIC = Inner class value, resultLambda = Enclosing scope value

As we can see, by calling this.value in IC, we can access a local variable from its instance. In the case of the lambda, this.value call gives us access to the variable value, which is defined in the UseFoo class, but not to the variable value defined inside the lambda's body.

## ****8. Keep Lambda Expressions Short and Self-explanatory****

If possible, we should use one line constructions instead of a large block of code. Remember, **lambdas should be an** **expression, not a narrative.**Despite its concise syntax,**lambdas should specifically express the functionality they provide.**

This is mainly stylistic advice, as performance will not change drastically. In general, however, it is much easier to understand and to work with such code.

This can be achieved in many ways; let's have a closer look.

### **8.1. Avoid Blocks of Code in Lambda's Body**

In an ideal situation, lambdas should be written in one line of code. With this approach, the lambda is a self-explanatory construction, which declares what action should be executed with what data (in the case of lambdas with parameters).

If we have a large block of code, the lambda's functionality is not immediately clear.

With this in mind, do the following:

Foo foo = parameter -> buildString(parameter);

**private** String **buildString**(String parameter) {

String result = "Something " + parameter;

//many lines of code

**return** result;

}

Instead of:

Foo foo = parameter -> { String result = "Something " + parameter;

//many lines of code

**return** result;

};

**It is important to note, we shouldn't use this “one-line lambda” rule as dogma**. If we have two or three lines in lambda's definition, it may not be valuable to extract that code into another method.

### **8.2. Avoid Specifying Parameter Types**

A compiler, in most cases, is able to resolve the type of lambda parameters with the help of [**type inference**](https://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html). Consequently, adding a type to the parameters is optional and can be omitted.

We can do this:

(a, b) -> a.toLowerCase() + b.toLowerCase();

Instead of this:

(String a, String b) -> a.toLowerCase() + b.toLowerCase();

### **8.3. Avoid Parentheses Around a Single Parameter**

Lambda syntax only requires parentheses around more than one parameter, or when there is no parameter at all. That's why it's safe to make our code a little bit shorter, and to exclude parentheses when there is only one parameter.

So we can do this:

a -> a.toLowerCase();

Instead of this:

(a) -> a.toLowerCase();

### **8.4. Avoid Return Statement and Braces**

**Braces** and ***return*** statements are optional in one-line lambda bodies. This means that they can be omitted for clarity and conciseness.

We can do this:

a -> a.toLowerCase();

Instead of this:

a -> {**return** a.toLowerCase()};

### **8.5. Use Method References**

Very often, even in our previous examples, lambda expressions just call methods which are already implemented elsewhere. In this situation, it is very useful to use another Java 8 feature, [**method references**](https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html).

The lambda expression would be:

a -> a.toLowerCase();

We could substitute it with:

String::toLowerCase;

This is not always shorter, but it makes the code more readable.

## ****9. Use “Effectively Final” Variables****

Accessing a non-final variable inside lambda expressions will cause a compile-time error, **but that doesn’t mean that we should mark every target variable as final.**

According to the “[**effectively final**](https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html)” concept, a compiler treats every variable as final as long as it is assigned only once.

It's safe to use such variables inside lambdas because the compiler will control their state and trigger a compile-time error immediately after any attempt to change them.

For example, the following code will not compile:

**public** **void** **method**() {

String localVariable = "Local";

Foo foo = parameter -> {

String localVariable = parameter;

**return** localVariable;

};

}

The compiler will inform us that:

Variable 'localVariable' is already defined **in** the scope.

This approach should simplify the process of making lambda execution thread-safe.

## ****10. Protect Object Variables From Mutation****

One of the main purposes of lambdas is use in parallel computing, which means that they're really helpful when it comes to thread-safety.

The “effectively final” paradigm helps a lot here, but not in every case. Lambdas can't change a value of an object from enclosing scope. But in the case of mutable object variables, a state could be changed inside lambda expressions.

Consider the following code:

**int**[] total = **new** **int**[1];

Runnable r = () -> total[0]++;

r.run();

This code is legal, as total variable remains “effectively final,” but will the object it references have the same state after execution of the lambda? No!

Keep this example as a reminder to avoid code that can cause unexpected mutations.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* STREAMS API IN JAVA \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## ****Overview****

In this article, we'll have a quick look at one of the major pieces of new functionality Java 8 had added – Streams.

We'll explain what streams are about and showcase the creation and basic stream operations with simple examples.

## ****2. Stream API****

One of the major new features in Java 8 is the introduction of the stream functionality – [java.util.stream](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html) – which contains classes for processing sequences of elements.

The central API class is the [*Stream<T>*](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html). The following section will demonstrate how streams can be created using the existing data-provider sources.

### **2.1. Stream Creation**

Streams can be created from different element sources e.g. collection or array with the help of stream() and of() methods:

String[] arr = **new** String[]{"a", "b", "c"};

Stream<String> stream = Arrays.stream(arr);

stream = Stream.of("a", "b", "c");

A stream() default method is added to the Collection interface and allows creating a Stream<T> using any collection as an element source:

Stream<String> stream = list.stream();

### **2.2. Multi-threading With Streams**

Stream API also simplifies multithreading by providing the parallelStream() method that runs operations over stream's elements in parallel mode.

The code below allows to run method doWork() in parallel for every element of the stream:

list.parallelStream().forEach(element -> doWork(element));

In the following section, we will introduce some of the basic Stream API operations.

## ****3. Stream Operations****

There are many useful operations that can be performed on a stream.

They are divided into**intermediate operations** (return Stream<T>) and **terminal operations** (return a result of definite type). Intermediate operations allow chaining.

It's also worth noting that operations on streams don't change the source.

Here's a quick example:

**long** count = list.stream().distinct().count();

So, the distinct() method represents an intermediate operation, which creates a new stream of unique elements of the previous stream. And the count() method is a terminal operation, which returns stream's size.

### **3.1. Iterating**

Stream API helps to substitute for, for-each, and while loops. It allows concentrating on operation's logic, but not on the iteration over the sequence of elements. For example:

**for** (String string : list) {

**if** (string.contains("a")) {

**return** **true**;

}

}

This code can be changed just with one line of Java 8 code:

**boolean** isExist = list.stream().anyMatch(element -> element.contains("a"));

### **3.2. Filtering**

The filter() method allows us to pick a stream of elements that satisfy a predicate.

For example, consider the following list:

ArrayList<String> list = **new** ArrayList<>();

list.add("One");

list.add("OneAndOnly");

list.add("Derek");

list.add("Change");

list.add("factory");

list.add("justBefore");

list.add("Italy");

list.add("Italy");

list.add("Thursday");

list.add("");

list.add("");

The following code creates a Stream<String> of the List<String>, finds all elements of this stream which contain char “d”, and creates a new stream containing only the filtered elements:

Stream<String> stream = list.stream().filter(element -> element.contains("d"));

### **3.3. Mapping**

To convert elements of a Stream by applying a special function to them and to collect these new elements into a Stream, we can use the map() method:

List<String> uris = **new** ArrayList<>();

uris.add("C:\\My.txt");

Stream<Path> stream = uris.stream().map(uri -> Paths.get(uri));

So, the code above converts Stream<String> to the Stream<Path> by applying a specific lambda expression to every element of the initial Stream.

If you have a stream where every element contains its own sequence of elements and you want to create a stream of these inner elements, you should use the flatMap() method:

List<Detail> details = **new** ArrayList<>();

details.add(**new** Detail());

Stream<String> stream

= details.stream().flatMap(detail -> detail.getParts().stream());

In this example, we have a list of elements of type Detail. The Detail class contains a field PARTS, which is a List<String>. With the help of the flatMap() method, every element from field PARTS will be extracted and added to the new resulting stream. After that, the initial Stream<Detail> will be lost.

### **3.4. Matching**

Stream API gives a handy set of instruments to validate elements of a sequence according to some predicate. To do this, one of the following methods can be used: anyMatch(), allMatch(), noneMatch(). Their names are self-explanatory. Those are terminal operations that return a boolean:

**boolean** isValid = list.stream().anyMatch(element -> element.contains("h")); // true

**boolean** isValidOne = list.stream().allMatch(element -> element.contains("h")); // false

**boolean** isValidTwo = list.stream().noneMatch(element -> element.contains("h")); // false

For empty streams, the allMatch() method with any given predicate will return true:

Stream.empty().allMatch(Objects::nonNull); // true

This is a sensible default, as we can't find any element that doesn't satisfy the predicate.

Similarly, the anyMatch() method always returns false for empty streams:

Stream.empty().anyMatch(Objects::nonNull); // false

Again, this is reasonable, as we can't find an element satisfying this condition.

### **3.5. Reduction**

Stream API allows reducing a sequence of elements to some value according to a specified function with the help of the reduce() method of the type Stream. This method takes two parameters: first – start value, second – an accumulator function.

Imagine that you have a List<Integer> and you want to have a sum of all these elements and some initial Integer (in this example 23). So, you can run the following code and result will be 26 (23 + 1 + 1 + 1).

List<Integer> integers = Arrays.asList(1, 1, 1);

Integer reduced = integers.stream().reduce(23, (a, b) -> a + b);

### **3.6. Collecting**

The reduction can also be provided by the collect() method of type Stream. This operation is very handy in case of converting a stream to a Collection or a Map and representing a stream in the form of a single string. There is a utility class Collectors which provide a solution for almost all typical collecting operations. For some, not trivial tasks, a custom Collector can be created.

List<String> resultList

= list.stream().map(element -> element.toUpperCase()).collect(Collectors.toList());

This code uses the terminal collect() operation to reduce a Stream<String> to the List<String>.

## ****Overview****

In this comprehensive tutorial, we'll go through the practical uses of Java 8 Streams from creation to parallel execution.

To understand this material, readers need to have a basic knowledge of Java 8 (lambda expressions, Optional, method references) and of the Stream API. In order to be more familiar with these topics, please take a look at our previous articles: [New Features in Java 8](https://www.baeldung.com/java-8-new-features) and [Introduction to Java 8 Streams](https://www.baeldung.com/java-8-streams-introduction).

## Further reading:

## [Lambda Expressions and Functional Interfaces: Tips and Best Practices](https://www.baeldung.com/java-8-lambda-expressions-tips)

Tips and best practices on using Java 8 lambdas and functional interfaces.

[**Read more**](https://www.baeldung.com/java-8-lambda-expressions-tips) →

## [Guide to Java 8's Collectors](https://www.baeldung.com/java-8-collectors)

The article discusses Java 8 Collectors, showing examples of built-in collectors, as well as showing how to build custom collector.

[**Read more**](https://www.baeldung.com/java-8-collectors) →

## ****2. Stream Creation****

There are many ways to create a stream instance of different sources. Once created, the instance **will not modify its source,**therefore allowing the creation of multiple instances from a single source.

### **2.1. Empty Stream**

We should use the **empty()** method in case of the creation of an empty stream:

Stream<String> streamEmpty = Stream.empty();

We often use the empty() method upon creation to avoid returning null for streams with no element:

**public** Stream<String> **streamOf**(List<String> list) {

**return** list == **null** || list.isEmpty() ? Stream.empty() : list.stream();

}

### **2.2. Stream of**Collection

We can also create a stream of any type of Collection (Collection, List, Set):

Collection<String> collection = Arrays.asList("a", "b", "c");

Stream<String> streamOfCollection = collection.stream();

### **2.3. Stream of Array**

An array can also be the source of a stream:

Stream<String> streamOfArray = Stream.of("a", "b", "c");

We can also create a stream out of an existing array or of part of an array:

String[] arr = **new** String[]{"a", "b", "c"};

Stream<String> streamOfArrayFull = Arrays.stream(arr);

Stream<String> streamOfArrayPart = Arrays.stream(arr, 1, 3);

### **2.4.** Stream.builder()

**When builder is used, the desired type should be additionally specified in the right part of the statement,** otherwise the build() method will create an instance of the Stream<Object>:

Stream<String> streamBuilder =

Stream.<String>builder().add("a").add("b").add("c").build();

### **2.5.** Stream.generate()

The **generate()** method accepts a Supplier<T> for element generation. As the resulting stream is infinite, the developer should specify the desired size, or the generate() method will work until it reaches the memory limit:

Stream<String> streamGenerated =

Stream.generate(() -> "element").limit(10);

The code above creates a sequence of ten strings with the value “element.”

### **2.6.** Stream.iterate()

Another way of creating an infinite stream is by using the **iterate()** method:

Stream<Integer> streamIterated = Stream.iterate(40, n -> n + 2).limit(20);

The first element of the resulting stream is the first parameter of the iterate() method. When creating every following element, the specified function is applied to the previous element. In the example above the second element will be 42.

### **2.7. Stream of Primitives**

Java 8 offers the possibility to create streams out of three primitive types: int, long and double. As Stream<T> is a generic interface, and there is no way to use primitives as a type parameter with generics, three new special interfaces were created: **IntStream, LongStream, DoubleStream.**

Using the new interfaces alleviates unnecessary auto-boxing, which allows for increased productivity:

IntStream intStream = IntStream.range(1, 3);

LongStream longStream = LongStream.rangeClosed(1, 3);

The **range(int startInclusive, int endExclusive)** method creates an ordered stream from the first parameter to the second parameter. It increments the value of subsequent elements with the step equal to 1. The result doesn't include the last parameter, it is just an upper bound of the sequence.

The ***rangeClosed(int startInclusive, int endInclusive)*** method does the same thing with only one difference, the second element is included. We can use these two methods to generate any of the three types of streams of primitives.

Since Java 8, the [Random](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Random.html) class provides a wide range of methods for generating streams of primitives. For example, the following code creates a DoubleStream, which has three elements:

Random random = **new** Random();

DoubleStream doubleStream = random.doubles(3);

### **2.8. Stream of**String

We can also use String as a source for creating a stream with the help of the chars() method of the String class. Since there is no interface for CharStream in JDK, we use the IntStream to represent a stream of chars instead.

IntStream streamOfChars = "abc".chars();

The following example breaks a String into sub-strings according to specified RegEx:

Stream<String> streamOfString =

Pattern.compile(", ").splitAsStream("a, b, c");

### **2.9. Stream of File**

Furthermore, Java NIO class Files allows us to generate a Stream<String> of a text file through the lines() method. Every line of the text becomes an element of the stream:

Path path = Paths.get("C:\\file.txt");

Stream<String> streamOfStrings = Files.lines(path);

Stream<String> streamWithCharset =

Files.lines(path, Charset.forName("UTF-8"));

The Charset can be specified as an argument of the lines() method.

## ****3. Referencing a Stream****

We can instantiate a stream, and have an accessible reference to it, as long as only intermediate operations are called. Executing a terminal operation makes a stream inaccessible.

To demonstrate this, we will forget for a while that the best practice is to chain the sequence of operation. Besides its unnecessary verbosity, technically the following code is valid:

Stream<String> stream =

Stream.of("a", "b", "c").filter(element -> element.contains("b"));

Optional<String> anyElement = stream.findAny();

However, an attempt to reuse the same reference after calling the terminal operation will trigger the IllegalStateException:

Optional<String> firstElement = stream.findFirst();

As the IllegalStateException is a RuntimeException, a compiler will not signalize about a problem. So it is very important to remember that **Java 8 streams can't be reused.**

This kind of behavior is logical. We designed streams to apply a finite sequence of operations to the source of elements in a functional style, not to store elements.

So to make the previous code work properly, some changes should be made:

List<String> elements =

Stream.of("a", "b", "c").filter(element -> element.contains("b"))

.collect(Collectors.toList());

Optional<String> anyElement = elements.stream().findAny();

Optional<String> firstElement = elements.stream().findFirst();

## ****4. Stream Pipeline****

To perform a sequence of operations over the elements of the data source and aggregate their results, we need three parts: the **source**, **intermediate operation(s)** and a **terminal operation.**

Intermediate operations return a new modified stream. For example, to create a new stream of the existing one without few elements, the skip() method should be used:

Stream<String> onceModifiedStream =

Stream.of("abcd", "bbcd", "cbcd").skip(1);

If we need more than one modification, we can chain intermediate operations. Let's assume that we also need to substitute every element of the current Stream<String> with a sub-string of the first few chars. We can do this by chaining the *skip()* and *map()* methods:

Stream<String> twiceModifiedStream =

stream.skip(1).map(element -> element.substring(0, 3));

As we can see, the map() method takes a lambda expression as a parameter. If we want to learn more about lambdas, we can take a look at our tutorial [Lambda Expressions and Functional Interfaces: Tips and Best Practices](https://www.baeldung.com/java-8-lambda-expressions-tips).

A stream by itself is worthless; the user is interested in the result of the terminal operation, which can be a value of some type or an action applied to every element of the stream. **We can only use one terminal operation per stream.**

The correct and most convenient way to use streams is by a **stream pipeline, which is a chain of the stream source, intermediate operations, and a terminal operation:**

List<String> list = Arrays.asList("abc1", "abc2", "abc3");

**long** size = list.stream().skip(1)

.map(element -> element.substring(0, 3)).sorted().count();

## ****5. Lazy Invocation****

**Intermediate operations are lazy.** This means that **they will be invoked only if it is necessary for the terminal operation execution.**

For example, let's call the method wasCalled(), which increments an inner counter every time it's called:

**private** **long** counter;

**private** **void** **wasCalled**() {

counter++;

}

Now let's call the method wasCalled() from operation filter():

List<String> list = Arrays.asList(“abc1”, “abc2”, “abc3”);

counter = 0;

Stream<String> stream = list.stream().filter(element -> {

wasCalled();

**return** element.contains("2");

});

As we have a source of three elements, we can assume that the filter() method will be called three times, and the value of the counter variable will be 3. However, running this code doesn't change counter at all, it is still zero, so the filter() method wasn't even called once. The reason why is missing of the terminal operation.

Let's rewrite this code a little bit by adding a map() operation and a terminal operation, findFirst(). We will also add the ability to track the order of method calls with the help of logging:

Optional<String> stream = list.stream().filter(element -> {

log.info("filter() was called");

**return** element.contains("2");

}).map(element -> {

log.info("map() was called");

**return** element.toUpperCase();

}).findFirst();

The resulting log shows that we called the filter() method twice and the map() method once. This is because the pipeline executes vertically. In our example, the first element of the stream didn't satisfy the filter's predicate. Then we invoked the filter() method for the second element, which passed the filter. Without calling the filter() for the third element, we went down through the pipeline to the map() method.

The findFirst() operation satisfies by just one element. So in this particular example, the lazy invocation allowed us to avoid two method calls, one for the filter() and one for the map().

## ****6. Order of Execution****

From the performance point of view, **the right order is one of the most important aspects of chaining operations in the stream pipeline:**

**long** size = list.stream().map(element -> {

wasCalled();

**return** element.substring(0, 3);

}).skip(2).count();

Execution of this code will increase the value of the counter by three. This means that we called the map() method of the stream three times, but the value of the size is one. So the resulting stream has just one element, and we executed the expensive map() operations for no reason two out of the three times.

If we change the order of the skip() and the map() methods, the counter will increase by only one. So we will call the map() method only once:

**long** size = list.stream().skip(2).map(element -> {

wasCalled();

**return** element.substring(0, 3);

}).count();

This brings us to the following rule: **intermediate operations which reduce the size of the stream should be placed before operations which are applying to each element.** So we need to keep methods such as skip(), filter(), and distinct() at the top of our stream pipeline.

## ****7. Stream Reduction****

The API has many terminal operations which aggregate a stream to a type or to a primitive: count(), max(), min(), and sum(). However, these operations work according to the predefined implementation. So what **if a developer needs to customize a Stream's reduction mechanism?** There are two methods which allow us to do this, the ***reduce()*** and the **collect()** methods.

### **7.1. The**reduce()**Method**

There are three variations of this method, which differ by their signatures and returning types. They can have the following parameters:

**identity –**the initial value for an accumulator, or a default value if a stream is empty and there is nothing to accumulate

**accumulator –**a function which specifies the logic of the aggregation of elements. As the accumulator creates a new value for every step of reducing, the quantity of new values equals the stream's size and only the last value is useful. This is not very good for the performance.

**combiner –**a function which aggregates the results of the accumulator. We only call combiner in a parallel mode to reduce the results of accumulators from different threads.

Now let's look at these three methods in action:

OptionalInt reduced =

IntStream.range(1, 4).reduce((a, b) -> a + b);

reduced = 6 (1 + 2 + 3)

**int** reducedTwoParams =

IntStream.range(1, 4).reduce(10, (a, b) -> a + b);

reducedTwoParams = 16 (10 + 1 + 2 + 3)

**int** reducedParams = Stream.of(1, 2, 3)

.reduce(10, (a, b) -> a + b, (a, b) -> {

log.info("combiner was called");

**return** a + b;

});

The result will be the same as in the previous example (16), and there will be no login, which means that combiner wasn't called. To make a combiner work, a stream should be parallel:

**int** reducedParallel = Arrays.asList(1, 2, 3).parallelStream()

.reduce(10, (a, b) -> a + b, (a, b) -> {

log.info("combiner was called");

**return** a + b;

});

The result here is different (36), and the combiner was called twice. Here the reduction works by the following algorithm: the accumulator ran three times by adding every element of the stream to identity. These actions are being done in parallel. As a result, they have (10 + 1 = 11; 10 + 2 = 12; 10 + 3 = 13;). Now combiner can merge these three results. It needs two iterations for that (12 + 13 = 25; 25 + 11 = 36).

### **7.2. The**collect()**Method**

The reduction of a stream can also be executed by another terminal operation, the collect() method. It accepts an argument of the type Collector, which specifies the mechanism of reduction. There are already created, predefined collectors for most common operations. They can be accessed with the help of the Collectors type.

In this section, we will use the following List as a source for all streams:

List<Product> productList = Arrays.asList(**new** Product(23, "potatoes"),

**new** Product(14, "orange"), **new** Product(13, "lemon"),

**new** Product(23, "bread"), **new** Product(13, "sugar"));

**Converting a stream to the Collection (*Collection, List*or*Set*):**

List<String> collectorCollection =

productList.stream().map(Product::getName).collect(Collectors.toList());

**Reducing to *String*:**

String listToString = productList.stream().map(Product::getName)

.collect(Collectors.joining(", ", "[", "]"));

The joiner() method can have from one to three parameters (delimiter, prefix, suffix). The most convenient thing about using joiner() is that the developer doesn't need to check if the stream reaches its end to apply the suffix and not to apply a delimiter. Collector will take care of that.

**Processing the average value of all numeric elements of the stream:**

**double** averagePrice = productList.stream()

.collect(Collectors.averagingInt(Product::getPrice));

**Processing the sum of all numeric elements of the stream:**

**int** summingPrice = productList.stream()

.collect(Collectors.summingInt(Product::getPrice));

The methods averagingXX(), summingXX() and summarizingXX() can work with primitives (int, long, double) and with their wrapper classes (Integer, Long, Double). One more powerful feature of these methods is providing the mapping. As a result, the developer doesn't need to use an additional map() operation before the collect() method.

**Collecting statistical information about stream’s elements:**

IntSummaryStatistics statistics = productList.stream()

.collect(Collectors.summarizingInt(Product::getPrice));

By using the resulting instance of type IntSummaryStatistics, the developer can create a statistical report by applying the toString() method. The result will be a String common to this one “IntSummaryStatistics{count=5, sum=86, min=13, average=17,200000, max=23}.”

It is also easy to extract from this object separate values for count, sum, min, and average by applying the methods getCount(), getSum(), getMin(), getAverage(), and getMax(). All of these values can be extracted from a single pipeline.

**Grouping of stream’s elements according to the specified function:**

Map<Integer, List<Product>> collectorMapOfLists = productList.stream()

.collect(Collectors.groupingBy(Product::getPrice));

In the example above, the stream was reduced to the Map, which groups all products by their price.

**Dividing stream’s elements into groups according to some predicate:**

Map<Boolean, List<Product>> mapPartioned = productList.stream()

.collect(Collectors.partitioningBy(element -> element.getPrice() > 15));

**Pushing the collector to perform additional transformation:**

Set<Product> unmodifiableSet = productList.stream()

.collect(Collectors.collectingAndThen(Collectors.toSet(),

Collections::unmodifiableSet));

In this particular case, the collector has converted a stream to a Set, and then created the unchangeable Set out of it.

**Custom collector:**

If for some reason a custom collector should be created, the easiest and least verbose way of doing so is to use the method of() of the type Collector.

Collector<Product, ?, LinkedList<Product>> toLinkedList =

Collector.of(LinkedList::**new**, LinkedList::add,

(first, second) -> {

first.addAll(second);

**return** first;

});

LinkedList<Product> linkedListOfPersons =

productList.stream().collect(toLinkedList);

In this example, an instance of the Collector got reduced to the LinkedList<Persone>.

## ****8. Parallel Streams****

Before Java 8, parallelization was complex. The emergence of the [ExecutorService](https://www.baeldung.com/java-executor-service-tutorial) and the *[ForkJoin](https://www.baeldung.com/java-fork-join)* simplified a developer’s life a little bit, but it was still worth remembering how to create a specific executor, how to run it, and so on. Java 8 introduced a way of accomplishing parallelism in a functional style.

The API allows us to create parallel streams, which perform operations in a parallel mode. When the source of a stream is a *Collection* or an *array*, it can be achieved with the help of the ***parallelStream()*** method:

Stream<Product> streamOfCollection = productList.parallelStream();

**boolean** isParallel = streamOfCollection.isParallel();

**boolean** bigPrice = streamOfCollection

.map(product -> product.getPrice() \* 12)

.anyMatch(price -> price > 200);

If the source of a stream is something other than a Collection or an array, the **parallel()** method should be used:

IntStream intStreamParallel = IntStream.range(1, 150).parallel();

**boolean** isParallel = intStreamParallel.isParallel();

Under the hood, Stream API automatically uses the ForkJoin framework to execute operations in parallel. By default, the common thread pool will be used and there is no way (at least for now) to assign some custom thread pool to it. [This can be overcome by using a custom set of parallel collectors.](https://github.com/pivovarit/parallel-collectors)

When using streams in parallel mode, avoid blocking operations. It is also best to use parallel mode when tasks need a similar amount of time to execute. If one task lasts much longer than the other, it can slow down the complete app’s workflow.

The stream in parallel mode can be converted back to the sequential mode by using the sequential() method:

IntStream intStreamSequential = intStreamParallel.sequential();

**boolean** isParallel = intStreamSequential.isParallel();

# Java Stream API Parallel Collectors - overcoming limitations of standard Parallel Streams

Parallel Collectors is a toolkit easing parallel collection processing in Java using Stream API... but without limitations imposed by standard Parallel Streams.

list.stream()

.collect(parallel(i -> foo(i), toList(), executor, parallelism))

.orTimeout(1000, MILLISECONDS)

.thenAcceptAsync(System.out::println, otherExecutor)

.thenRun(() -> System.out.println("Finished!"));

They are:

* lightweight (yes, you could achieve the same with Project Reactor, but that's often a hammer way too big for the job)
* powerful (combined power of Stream API and CompletableFutures allows to specify timeouts, compose with other CompletableFutures, or just perform the whole processing asynchronously)
* configurable (it's possible to provide your own Executor, parallelism)
* non-blocking (no need to block the calling thread while waiting for the result to arrive)
* short-circuiting (if one of the operations raises an exception, remaining tasks will get interrupted)
* non-invasive (they are just custom implementations of Collector interface, no magic inside, zero-dependencies)
* versatile (missing an API for your use case? process the resulting Stream with the whole generosity of Stream API by reusing already available Collectors)

### Maven Dependencies

<dependency>

<groupId>com.pivovarit</groupId>

<artifactId>parallel-collectors</artifactId>

<version>2.5.0</version>

</dependency>

##### Gradle

compile 'com.pivovarit:parallel-collectors:2.5.0'

## Philosophy

Parallel Collectors are unopinionated by design, so it's up to their users to use them responsibly, which involves things like:

* proper configuration of a provided Executor and its lifecycle management
* choosing the appropriate parallelism level
* making sure that the tool is applied in the right context

Make sure to read API documentation before using these in production.

## Basic API

The main entrypoint is the com.pivovarit.collectors.ParallelCollectors class - which follows the convention established by java.util.stream.Collectors and features static factory methods returning custom java.util.stream.Collector implementations spiced up with parallel processing capabilities.

By design, it's obligatory to supply a custom Executor instance and manage its lifecycle.

All parallel collectors are one-off and must not be reused.

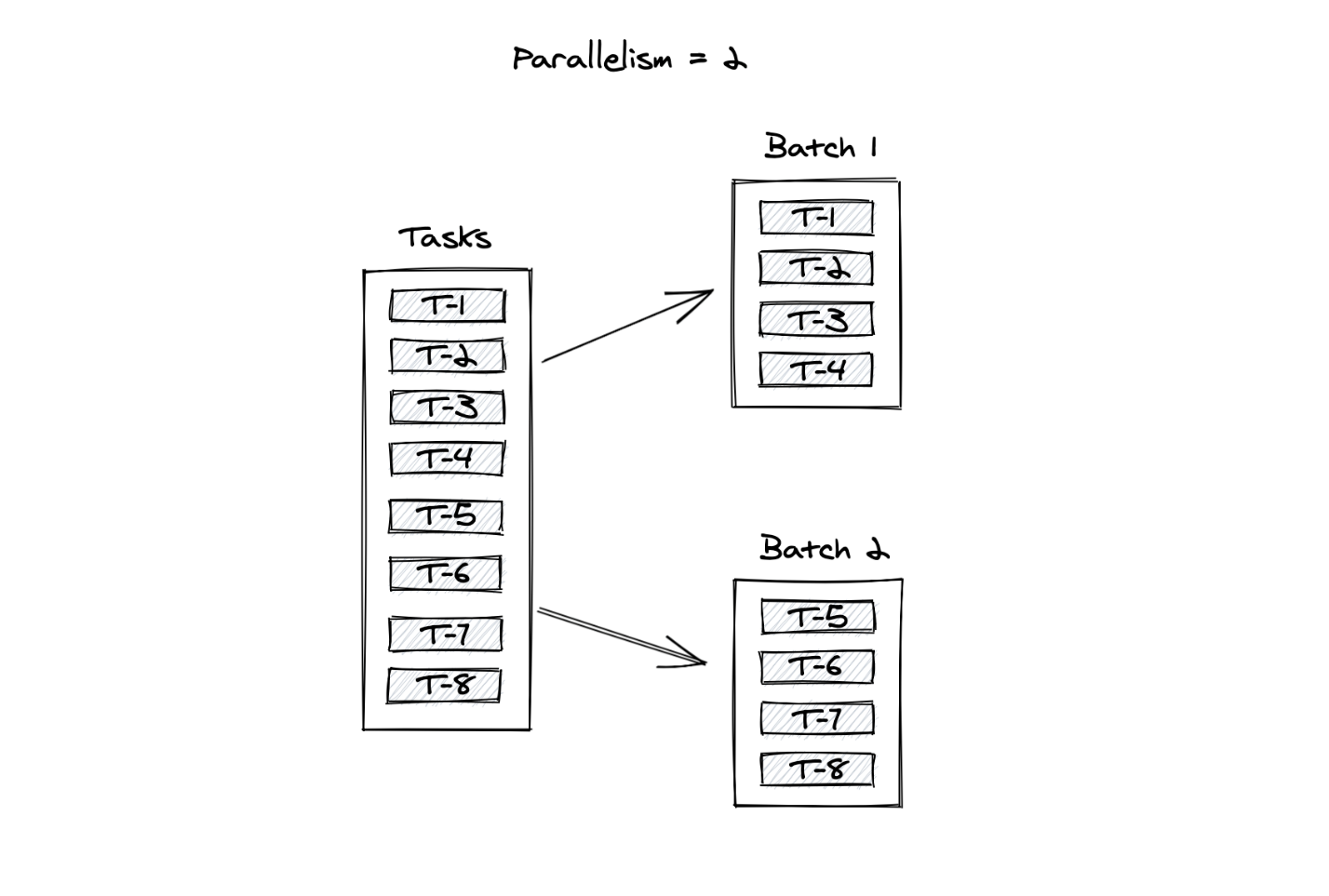
### Available Collectors:

* CompletableFuture<Collection<T>> parallel(Function, Collector, Executor, parallelism)
* CompletableFuture<Stream<T>> parallel(Function, Executor, parallelism)
* Stream<T> parallelToStream(Function, Executor, parallelism)
* Stream<T> parallelToOrderedStream(Function, Executor, parallelism)

#### Batching Collectors

By default, all ExecutorService threads compete for each task separately - which results in a basic form of work-stealing, which, unfortunately, is not free, but can decrease processing time for subtasks with varying processing time.

However, if the processing time for all subtasks is similar, it might be better to distribute tasks in batches to avoid excessive contention:

[](https://github.com/pivovarit/parallel-collectors/blob/master/docs/batching.png)

Batching alternatives are available under the ParallelCollectors.Batching namespace.

### Leveraging CompletableFuture

Parallel Collectors™ expose results wrapped in CompletableFuture instances which provides great flexibility and possibility of working with them in a non-blocking fashion:

CompletableFuture<List<String>> result = list.stream()

.collect(parallel(i -> foo(i), toList(), executor));

This makes it possible to conveniently apply callbacks, and compose with other CompletableFutures:

list.stream()

.collect(parallel(i -> foo(i), toSet(), executor))

.thenAcceptAsync(System.out::println, otherExecutor)

.thenRun(() -> System.out.println("Finished!"));

Or just join() if you just want to block the calling thread and wait for the result:

List<String> result = list.stream()

.collect(parallel(i -> foo(i), toList(), executor))

.join();

What's more, since JDK9, [you can even provide your own timeout easily](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/CompletableFuture.html#orTimeout(long,java.util.concurrent.TimeUnit)).

## Examples

##### 1. Apply i -> foo(i) in parallel on a custom Executor and collect to List

Executor executor = ...

CompletableFuture<List<String>> result = list.stream()

.collect(parallel(i -> foo(i), toList(), executor));

##### 2. Apply i -> foo(i) in parallel on a custom Executor with max parallelism of 4 and collect to Set

Executor executor = ...

CompletableFuture<Set<String>> result = list.stream()

.collect(parallel(i -> foo(i), toSet(), executor, 4));

##### 3. Apply i -> foo(i) in parallel on a custom Executor and collect to LinkedList

Executor executor = ...

CompletableFuture<List<String>> result = list.stream()

.collect(parallel(i -> foo(i), toCollection(LinkedList::new), executor));

##### 4. Apply i -> foo(i) in parallel on a custom Executor and stream results in completion order

Executor executor = ...

list.stream()

.collect(parallelToStream(i -> foo(i), executor))

.forEach(i -> ...);

##### 5. Apply i -> foo(i) in parallel on a custom Executor and stream results in original order

Executor executor = ...

list.stream()

.collect(parallelToOrderedStream(i -> foo(i), executor))

.forEach(i -> ...);

## Rationale

Stream API is a great tool for collection processing, especially if you need to parallelize execution of CPU-intensive tasks, for example:

public static void parallelSetAll(int[] array, IntUnaryOperator generator) {

Objects.requireNonNull(generator);

IntStream.range(0, array.length).parallel().forEach(i -> { array[i] = generator.applyAsInt(i); });

}

**However, Parallel Streams execute tasks on a shared ForkJoinPool instance**.

Unfortunately, it's not the best choice for running blocking operations even when using ManagedBlocker - [as explained here by Tagir Valeev](https://stackoverflow.com/a/37518272/2229438)) - this could easily lead to the saturation of the common pool, and to a performance degradation of everything that uses it.

For example:

List<String> result = list.parallelStream()

.map(i -> foo(i)) // runs implicitly on ForkJoinPool.commonPool()

.collect(Collectors.toList());

In order to avoid such problems, **the solution is to isolate blocking tasks** and run them on a separate thread pool... but there's a catch.

**Sadly, Streams can only run parallel computations on the common ForkJoinPool** which effectively restricts the applicability of them to CPU-bound jobs.

However, there's a trick that allows running parallel Streams in a custom FJP instance... but it's not considered reliable:

Note, however, that this technique of submitting a task to a fork-join pool to run the parallel stream in that pool is an implementation "trick" and is not guaranteed to work. Indeed, the threads or thread pool that is used for execution of parallel streams is unspecified. By default, the common fork-join pool is used, but in different environments, different thread pools might end up being used.

Says [Stuart Marks on StackOverflow](https://stackoverflow.com/questions/28985704/parallel-stream-from-a-hashset-doesnt-run-in-parallel/29272776#29272776).

Not even mentioning that this approach was seriously flawed before JDK-10 - if a Stream was targeted towards another pool, splitting would still need to adhere to the parallelism of the common pool, and not the one of the targeted pool [[JDK8190974]](https://bugs.openjdk.java.net/browse/JDK-8190974).

### Dependencies

None - the library is implemented using core Java libraries.

### Limitations

Upstream Stream is always evaluated as a whole, even if the following operation is short-circuiting. This means that none of these should be used for working with infinite streams.

This limitation is imposed by the design of the Collector API.

### Good Practices

* Consider providing reasonable timeouts for CompletableFutures in order to not block for unreasonably long in case when something bad happens [(how-to)](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/CompletableFuture.html#orTimeout(long,java.util.concurrent.TimeUnit))
* Name your thread pools - it makes debugging easier [(how-to)](https://stackoverflow.com/a/9748697/2229438)
* Limit the size of a working queue of your thread pool [(source)](https://mechanical-sympathy.blogspot.com/2012/05/apply-back-pressure-when-overloaded.html)
* Limit the level of parallelism [(source)](https://mechanical-sympathy.blogspot.com/2012/05/apply-back-pressure-when-overloaded.html)
* A no-longer-used ExecutorService should be shut down to allow reclamation of its resources
* Keep in mind that CompletableFuture#then(Apply|Combine|Consume|Run|Accept) might be executed by the calling thread. If this is not suitable, use CompletableFuture#then(Apply|Combine|Consume|Run|Accept)Async instead, and provide a custom executor instance.

## Words of Caution

Even if this tool makes it easy to parallelize things, it doesn't always mean that you should. **Parallelism comes with a price that can be often higher than not using it at all.** Threads are expensive to create, maintain and switch between, and you can only create a limited number of them.

It's essential to follow up on the root cause and double-check if parallelism is the way to go.

**It often turns out that the root cause can be addressed by using a simple JOIN statement, batching, reorganizing your data... or even just by choosing a different API method.**