O'REILLY®

Modern Java Recipes

SIMPLE SOLUTIONS TO DIFFICULT PROBLEMS IN JAVA 8 AND 9

Ken Kousen

Modern Java Recipes

Simple Solutions to Difficult Problems in Java 8 and 9

Ken Kousen

Modern Java Recipes

by Ken Kousen

Printed in the United States of America.

Published by O’Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com/safari). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

- Editor: Brian Foster
- Production Editor: FILL IN PRODUCTION EDITOR
- Copyeditor: FILL IN COPYEDITOR
- Proofreader: FILL IN PROOFREADER
- Indexer: FILL IN INDEXER
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Rebecca Demarest

- January -4712: First Edition

Revision History for the First Edition

- 2017-03-16: First Early Release
- 2017-03-30: Second Early Release
- 2017-06-01: Third Early Release

See http://oreilly.com/catalog/errata.csp?isbn=9781491973103 for release details.

978-1-491-97310-3

[FILL IN]

Chapter 1. The Basics

The biggest change in Java 8 is the addition of concepts from functional programming to the language. Specifically, the language added lambda expressions, method references, and streams.

If you haven't used the new functional features yet, you'll probably be surprised by how different your code will look from previous Java versions. The changes in Java 8 represent the biggest changes to the language ever. In many ways, it feels like you're learning a completely new language.

The question then becomes, why do this? Why make such drastic changes to a language that's already twenty years old and plans to maintain backward compatibility? Why make such dramatic revisions to a language that has been, by all accounts, extremely successful? Why switch to a functional paradigm after all these years of being one of the most successful object-oriented languages ever?

The answer is that the software development world has changed, so languages that want to be successful in the future need to adapt as well. Back in the mid-90's, when Java was shiny and new, Moore's Law[1] was still fully in force. All you had to do was wait a couple of years and your computer would double in speed.

Today's hardware no longer relies on increasing chip density for speed. Instead, even most phones have multiple cores, which means software needs to be written expecting to be run in a multi-processor environment. Functional programming, with its emphasis on "pure" functions (that return the same result given the same inputs, with no side-effects) and immutability simplifies programming in parallel environments. If you don't have any shared, mutable state, and your program can be decomposed into collections of simple functions, it is easier to understand and predict its behavior.

This, however, is not a book about Haskell, or Erlang, or Frege, or any of the other functional programming languages. This book is about Java, and the changes made to the language to add functional concepts to what is still fundamentally an object-oriented language.

Java now supports lambda expressions, which are essentially methods treated as though they were first-class objects. The language also has method references, which allow you to use an existing method wherever a lambda expression is expected. In order to take advantage of lambda expressions and method references, the language also added a stream model, which produces elements and passes them through a pipeline of transformations and filters without modifying the original source.

The recipes in this chapter describe the basic syntax for lambda expressions, method references, functional interfaces, as well the new support for static and default methods in interfaces. Streams are discussed in detail in Chapter 4.

Lambda Expressions

Problem

You want to use lambda expressions in your code.

Solution

Use one of the varieties of lambda expression syntax and assign the result to a reference of functional interface type.

Discussion

A *functional interface* is an interface with a single abstract method (SAM). A class implements any interface by providing implementations for all the methods in it. This can be done with a top-level class, an inner class, or even an anonymous inner class.

For example, consider the `Runnable` interface, which has been in Java since version 1.0. It contains a single abstract method called `run`, which takes no arguments and returns `void`. The `Thread` class constructor takes a `Runnable` as an argument, so an anonymous inner class implementation is shown in Example 1-1.

Example 1-1. Anonymous Inner Class Implementation of `Runnable`

```
public class RunnableDemo {
    public static void main(String[] args) {
        // Works in Java 7 or earlier:
        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println(
                    "inside runnable using an anonymous inner c
            }
        }).start();
    }
}
```

The anonymous inner class syntax consists of the word `new` followed by the `Runnable` interface name and parentheses, implying that you're defining a class without an explicit name that implements that interface. The code in the braces `{}` then overrides the `run` method, which simply prints a string to the console.

The code in Example 1-2 shows the same example using a lambda expression.

Example 1-2. Using a lambda expression in a `Thread` constructor

```
public class RunnableDemo {
    public static void main(String[] args) {
        new Thread(() -> System.out.println(
            "inside Thread constructor using lambda")).start();
    }
}
```

The syntax uses an arrow to separate the arguments (since there are zero arguments here, only a pair of empty parentheses is used) from the body. In this case, the body consists of a single line, so no braces are required. This is known as an expression lambda. Whatever value the expression evaluates to is returned automatically. In this case, since `println` returns `void`, the return from the expression is also `void`, which matches the return type of the `run` method.

A lambda expression must match the argument types and return type in the signature of the single abstract method in the interface. This is called being *compatible* with the method signature. The lambda expression is thus the implementation of the interface method, and can also be assigned to a reference of that interface type.

As a demonstration, Example 1-3 shows the lambda assigned to a variable.

Example 1-3. Assigning a lambda expression to a variable

```
public class RunnableDemo {
    public static void main(String[] args) {
        Runnable r = () -> System.out.println(
            "lambda expression implementing the run method");
        new Thread(r).start();
    }
}
```

Note

There is no class in the Java library called `Lambda`. Lambda expressions can only be assigned to functional interfaces.

Assigning a lambda to the functional interface is the same as saying the lambda is the implementation of the single abstract method inside it. You can think of the lambda as the body of an anonymous inner class that implements the interface. That is why the lambda must be compatible with the abstract method; its argument types and return type must match the signature of that method. Notably, however, the name of the method being implemented is not important. It does not appear anywhere as part of the lambda expression syntax.

This example was particularly simple because the `run` method takes no arguments and returns `void`. Consider instead the functional interface `java.io.FilenameFilter`, which again has been part of the Java standard library since version 1.0. `FilenameFilter` instances are used as arguments to the `File.list` method to restrict the returns files to only those that satisfy the method.

From the Javadocs, the `FilenameFilter` class contains the single abstract method `accept`, with the following signature:

```
boolean accept(File dir, String name)
```

The `File` argument is the directory in which the file is found, and the `String` name is the name of the file.

The code in Example 1-4 implements `FilenameFilter` using an anonymous inner class to return only Java source files.

Example 1-4. An anonymous inner class implementation of `FilenameFilter`

```
import java.io.File;
import java.io.FilenameFilter;
import java.util.Arrays;

public class UseFilenameFilter {
    public static void main(String[] args) {
        File directory = new File("./src/main/java");

        // Anonymous inner class
        String[] names = directory.list(new FilenameFilter() {
            @Override
            public boolean accept(File dir, String name) {
                return name.endsWith(".java");
            }
        });
```

```
        System.out.println(Arrays.asList(names));
    }
}
```

In this case, the `accept` method returns true if the file name ends with `.java` and false otherwise.

The lambda expression version is shown in Example 1-5.

Example 1-5. Lambda expression implementing `FilenameFilter`

```
import java.io.File;
import java.io.FilenameFilter;
import java.util.Arrays;

public class UseFilenameFilter {
    public static void main(String[] args) {
        File directory = new File("./src/main/java");

        // Use lambda expression instead
        String[] names = directory.list((dir, name) -> name.enc
        System.out.println(Arrays.asList(names));
    }
}
```

The resulting code is much simpler. This time the arguments are contained within parentheses, but do not have types declared. At compile time, the compiler knows that the `list` method takes an argument of type `FilenameFilter`, and therefore knows that the signature of its single abstract method (`accept`). It therefore knows that the arguments to `accept` are a `File` and a `String`, so that the compatible lambda expression arguments must match those types. The return type on `accept` is a boolean, so the expression to the right of the arrow must also return a boolean.

If you wish to specify the data types in the code, you are free to do so, as in Example 1-6.

Example 1-6. Lambda expression is explicit data types

```
import java.io.File;
import java.io.FilenameFilter;
```

```
import java.util.Arrays;

public class UseFilenameFilter {
    public static void main(String[] args) {
        File directory = new File("./src/main/java");

        // Explicit data types
        String[] names = directory.list((File dir, String name)
            name.endsWith(".java"));
        System.out.println(Arrays.asList(names));
    }
}
```

Finally, if the implementation of the lambda requires more than one line, you need to use braces and an explicit return statement, as shown in Example 1-7.

Example 1-7. A block lambda

```
import java.io.File;
import java.io.FilenameFilter;
import java.util.Arrays;

public class UseFilenameFilter {
    public static void main(String[] args) {
        File directory = new File("./src/main/java");

        String[] names = directory.list((File dir, String name)
            return name.endsWith(".java");
        });
        System.out.println(Arrays.asList(names));
    }
}
```

This is known as a block lambda. In this case the body still consists of a single line, but the braces now allow for multiple statements. The `return` keyword is now required.

Lambda expressions never exist alone. There is always a *context* for the expression, which indicates the functional interface to which the expression is assigned. A lambda can be an argument to a method, a return type from a method, or assigned to a reference. In each case, the type of the assignment must be a functional interface.

See Also

Method References

Problem

You want to use a method reference to access an existing method and treat it like a lambda expression.

Solution

Use the double-colon notation to separate an instance reference or class name from the method.

Discussion

If a lambda expression is essentially treating a method as though it was a object, then a method reference treats an existing method as though it was a lambda.

For example, the `forEach` method in `Iterable` takes a `Consumer` as an argument. Example 1-8 shows that the `Consumer` can be implemented as either a lambda expression or as a method reference.

Example 1-8. Using a method reference to access `println`

```
Stream.of(3, 1, 4, 1, 5, 9)
        .forEach(x -> System.out.println(x)); ❶

Stream.of(3, 1, 4, 1, 5, 9)
        .forEach(System.out::println);     ❷

Consumer<Integer> printer = System.out::println;  ❸
Stream.of(3, 1, 4, 1, 5, 9)
        .forEach(printer);
```

❶

Using a lambda expression

❷

Using a method reference

❸

Assigning the method reference to a functional interface

The double-colon notation provides the reference to the `println` method on the `System.out` instance, which is a reference of type `PrintStream`. No parentheses are placed at the end of the method reference.

Tip

If you write a lambda expression that consists of one line that invokes a method, consider using the equivalent method reference instead.

The method reference provides a couple of (minor) advantages over the lambda syntax. First, it tends to be shorter, and second, it often includes the name of the class containing the method. Both make the code easier to read.

Method references can be used with static methods as well, as shown in Example 1-9.

Example 1-9. Using a method reference to a static method

```
Stream.generate(Math::random) ❶
        .limit(10)
        .forEach(System.out::println); ❷
```

❶

static method

❷

instance method

The `generate` method on `Stream` takes a `Supplier` as an argument, which is a functional interface whose single abstract method takes no arguments and produces a single result. The `random` method in the `Math` class is compatible with that signature, because it also takes no arguments and produces a single, uniformly-distributed, pseudo-random double between 0 and 1. The method reference `Math::random` refers to that method as the implementation of the `Supplier` interface.

Since `Stream.generate` produces an infinite stream, the `limit` method is used to ensure only 10 values are produced, which are then printed to standard output using the `System.out::println` method reference as an implementation of `Consumer`.

Syntax

There are three forms of the method reference syntax, and one is a bit misleading:

`object::instanceMethod`

> Refer to an instance method using a reference to the supplied object, as in `System.out::println`

`Class::staticMethod`

> Refer to static method, as in `Math::max`

`Class::instanceMethod`

> Invoke the instance method on a reference to an object supplied by the context, as in `String::length`

That last example is the confusing one, because as Java developers we're accustomed to seeing only static methods invoked via a class name. Remember that lambda expressions and method references never exist in a vacuum—there's always a context. In the case of an object reference, the context will supply the argument(s) to the method. In the printing case, the equivalent lambda expression is (as shown in context in Example 1-8 above):

```
// equivalent to System.out::println
x -> System.out.println(x)
```

The context provides the value of `x`, which is used as the method argument.

The situation is similar for the static `max` method:

```
// equivalent to Math::max
(x,y) -> Math.max(x,y)
```

Now the context needs to supply two arguments, and the lambda returns the greater one.

The "instance method through the class name" syntax is interpreted differently. The equivalent lambda is:

```
// equivalent to String::length
```

```
x -> x.length()
```

This time, when the context provides `x`, it is used as the target of the method, rather than as an argument.

TIP

> If you refer to a method that takes multiple arguments via the class name, the first element supplied by the context becomes the target and the remaining elements are arguments to the method.

See Example 1-10 for an example.

Example 1-10. Invoking a multiple-argument instance method from a class reference

```
List<String> strings =
    Arrays.asList("this", "is", "a", "list", "of", "strings");
List<String> sorted = strings.stream()
        .sorted((s1, s2) -> s1.compareTo(s2))   ❶
        .collect(Collectors.toList());

List<String> sorted = strings.stream()
        .sorted(String::compareTo)              ❶
        .collect(Collectors.toList());
```

❶

Method reference and equivalent lambda

The `sorted` method on `Stream` takes a `Comparator<T>` as an argument, whose single abstract method is `int compare(String other)`. The `sorted` method supplies each pair of strings to the comparator and sorts them based on the sign of the returned integer. In this case, the context is a pair of strings. The method reference syntax, using the class name `String`, invokes the `compareTo` method on the first element (`s1` in the lambda expression) and uses the second element `s2` as the argument to the method.

In stream processing, you frequently access an instance method using the class name in a method reference if you are processing a series of inputs. The code in Example 1-11 shows the invocation of the `length` method on each individual `String` in the stream.

Example 1-11. Invoking the `length` method on `String` using a method reference

```
Stream.of("this", "is", "a", "stream", "of", "strings")
        .map(String::length)                ❶
        .forEach(System.out::println);   ❷
```

❶

Instance method via class name

❷

Instance method via object reference

This example transforms each string into an integer by invoking the `length` method, then prints each result.

A method reference is essentially an abbreviated syntax for a lambda. Lambda expressions are more general, in that each method reference has an equivalent lambda expression but not vice versa. The equivalent lambdas for the method references from Example 1-11 are shown in Example 1-12.

Example 1-12. Lambda expression equivalents for method references

```
Stream.of("this", "is", "a", "stream", "of", "strings")
        .map(s -> s.length())
        .forEach(x -> System.out.println(x));
```

As with any lambda expression, the context matters. You can also use `this` or `super` as the left-side of a method reference if there is any ambiguity.

See Also

You can also invoke constructors using the method reference syntax. Constructor references are shown in “Constructor References”.

Constructor References

Problem

You want to instantiate an object using a method reference as part of a stream pipeline.

Solution

Use the `new` keyword as part of a method reference.

Discussion

When people talk about the new syntax added to Java 8, they mention lambda expressions, method references, and streams. For example, say you had a list of people and you wanted to convert it to a list of names. One way to do so would be the snippet shown in Example 1-13.

Example 1-13. Converting a list of People to a list of names

```
List<String> names = people.stream()
    .map(person -> person.getName())  ❶
    .collect(Collectors.toList());

// or, alternatively,

List<String> names = people.stream()
    .map(Person::getName)              ❷
    .collect(Collectors.toList());
```

❶

Lambda expression

❷

Method reference

What if you want to go the other way? What if you have a list of strings and you want to create a list of `Person` references from it? In that case you can use a method reference, but this time using the keyword `new`. That's a *constructor reference*.

To show how it is used, start with a `Person` class, which is just about the simplest Plain Old Java Object (POJO) imaginable. All it does is wrap a simple string attribute called `name` in Example 1-14.

Example 1-14. A Person class

```
public class Person {
    private String name;

    public Person() {}

    public Person(String name) {
        this.name = name;
    }

    // getters and setters ...

    // equals, hashCode, and toString methods ...
}
```

Given a collection of strings, you can map each one into a `Person` either a lambda expression or the constructor reference in Example 1-15.

Example 1-15. Transforming strings into `Person` instances

```
List<String> names =
    Arrays.asList("Grace Hopper", "Barbara Liskov", "Ada Lovela
        "Karen Spärck Jones");

List<Person> people = names.stream()
    .map(name -> new Person(name))  ❶
    .collect(Collectors.toList());

// or, alternatively,

List<Person> people = names.stream()
    .map(Person::new)               ❷
    .collect(Collectors.toList());
```

❶

Using a lambda expression to invoke the constructor

❷

Using a constructor reference instantiating `Person`

The syntax `Person::new` refers to the constructor in the `Person` class. As with all lambda expressions, the context determines which constructor is executed. Because the context supplies a string, the one-arg `String` constructor is used.

Copy Constructor

A copy constructor takes a `Person` argument and returns a new `Person` with the same attributes, as shown in Example 1-16.

Example 1-16. A copy constructor for Person

```
public Person(Person p) {
    this.name = p.name;
}
```

This is useful if you want to isolate streaming code from the original instances. For example, if you already have a list of people, convert the list is a stream and then back into a list, the references are the same (see Example 1-17).

Example 1-17. Converting a list to a stream and back

```
Person before = new Person("Grace Hopper");

List<Person> people = Stream.of(before)
    .collect(Collectors.toList());
Person after = people.get(0);

assertTrue(before == after);                                  ❶

before.setName("Grace Murray Hopper");                        ❷
assertEquals("Grace Murray Hopper", after.getName());  ❸
```

❶

Same object

❷

Change name using `before` reference

❸

Name has changed in the `after` reference

Using a copy constructor, you can break that connection, as in Example 1-18.

Example 1-18. Using the copy constructor

```
people = Stream.of(before)
      .map(Person::new)              ❶
      .collect(Collectors.toList());
after = people.get(0);
assertFalse(before == after);        ❷
assertEquals(before, after);         ❸

before.setName("Rear Admiral Dr. Grace Murray Hopper");
assertFalse(before.equals(after));
```

❶ Use copy constructor

❷ Different objects

❸ But equivalent

This time, when invoking the `map` method, the context is a stream of `Person` instances. Therefore the `Person::new` syntax invokes the constructor that takes a `Person` and returns a new, but equivalent, instance. I've broken the connection between the before reference and the after reference.[2]

Varargs Constructor

Consider now a varargs constructor added to the `Person` POJO, shown in Example 1-19.

Example 1-19. A Person constructor that takes a variable argument list of String

```
public Person(String... names) {
    this.name = Arrays.stream(names)
                      .collect(Collectors.joining(" "));
}
```

This constructor takes zero or more string arguments and concatenates them together with a single space as the delimiter.

How can that constructor get invoked? Any client that passes zero or more string arguments separated by commas will call it. One way to do that is to take advantage of the `split` method on `String` that takes a delimiter and returns a String array:

```
String[] split(String delimiter)
```

Therefore, the code in Example 1-20 splits each string in the list into individual words and invokes the varargs constructor, Example 1-20.

Example 1-20. Using the varargs constructor

```
names.stream()                          ❶
    .map(name -> name.split(" "))      ❷
    .map(Person::new)                  ❸
    .collect(Collectors.toList());     ❹
```

❶

Create a stream of strings

❷

Map to a stream of string arrays

❸

Map to a stream of `Person`

❹

Collect to a list of `Person`

This time, the context for the `map` method that contains the `Person::new` constructor reference is a stream of string arrays, so the varargs constructor is called. If you add a simple print statement to that constructor:

```
System.out.println("Varargs ctor, names=" + Arrays.toList(names
```

then the result is:

```
Varargs ctor, names=[Grace, Hopper]
Varargs ctor, names=[Barbara, Liskov]
Varargs ctor, names=[Ada, Lovelace]
Varargs ctor, names=[Karen, Spärck, Jones]
```

Arrays

Constructor references can also be used with arrays. If you want an array of `Person` instances, `Person[]`, instead of a list, you can use the `toArray` method on `Stream`, whose signature is:

```
<A> A[] toArray(IntFunction<A[]> generator)
```

This method uses `A` to represent the generic type of the array returned containing the elements of the stream, which is created using the provided generator function. The cool part is that a constructor reference can be used for that, too, as in Example 1-21.

Example 1-21. Creating an array of `Person` references

```
Person[] people = names.stream()
    .map(Person::new)            ❶
    .toArray(Person[]::new);     ❷
```

❶

Constructor reference for `Person`

❷

Constructor reference for an array of `Person`

The `toArray` method argument creates an array of `Person` references of the proper size and populates it with the instantiated `Person` instances.

Constructor references are just method references by another name, using the word new to invoke a constructor. Which constructor is determined by the context, as usual. This technique gives a lot of flexibility when processing streams.

See Also

Method references are discussed in “Method References”.

Functional Interfaces

Problem

You want to use an existing functional interface, or write your own.

Solution

Create an interface with a single, abstract method, and add the `@FunctionalInterface` annotation.

Discussion

A functional interface in Java 8 is an interface with a single, abstract method. As such, it can be the target for a lambda expression or method reference.

The use of the term "abstract" here is significant. Prior to Java 8, all methods in interfaces were considered abstract by default — you didn't even need to add the keyword.

For example, here is the definition of an interface called `PalindromeChecker`, shown in Example 1-22.

Example 1-22. A Palindrome Checker interface

```
@FunctionalInterface
public interface PalindromeChecker {
    boolean isPalidrome(String s);
}
```

All methods in an interface are `public`, so you can leave out the access modifier, and as stated you can also leave out the `abstract` keyword.

Since this interface has only a single, abstract method, it is a functional interface. Java 8 provides an annotation called `@FunctionalInterface` in the `java.lang` package that can be applied to the interface, as shown in the example.

The annotation is not required, but is a good idea, for two reasons. First, it triggers a compile-time check that this interface does, in fact, satisfy the requirement. If the interface has either zero or more than one abstract methods, the compiler will give an error.

The other benefit to adding the `@FunctionalInterface` annotation is that it generates a statement in the JavaDocs as follows:

```
Functional Interface:
This is a functional interface and can therefore be used as the
```

`target for a lambda expression or method reference.`

Functional interfaces can have `default` and `static` methods as well. Both default and static methods have implementations, so they don't count against the single abstract method requirement. Example 1-23 shows an example.

Example 1-23. MyInterface is a functional interface with static and default methods

```
@FunctionalInterface
public interface MyInterface {
    int myMethod();  ❶
    // int myOtherMethod();

    default String sayHello() {
        return "Hello, World!";
    }

    static void myStaticMethod() {
        System.out.println("I'm a static method in an interface
    }
}
```

❶

The single abstract method

Note that if the commented method `myOtherMethod` was included, the interface would no longer satisfy the functional interface requirement. The annotation would generate an error of the form "multiple non-overriding abstract methods found".

Interfaces can extend other interfaces, even more than one. The annotation checks the current interface. So if one interface extends an existing functional interface and adds another abstract method, it is not itself a functional interface. See Example 1-24.

Example 1-24. Extending a functional interface — no longer functional

```
public interface MyChildInterface extends MyInterface {
    int myOtherMethod(); ❶
}
```

❶

Additional abstract method

The `MyChildInterface` is not a functional interface, because it has two abstract methods: `myMethod` which it inherits from `MyInterface`, and `myOtherMethod` which it declares. Without the `@FunctionalInterface` annotation, this compiles, because it's a standard interface. It cannot, however, be the target of a lambda expression.

The phrase, "target of a lambda expression" means that a reference of the interface type can be declared and a lambda expression or method reference can be assigned to the result. For example, consider Example 1-25, a JUnit test that checks a lambda implementation of the `PalindromeChecker` interface.

Example 1-25. Testing a palindrome checker using a lambda

```
import org.junit.Test;

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

import static org.junit.Assert.assertTrue;

public class PalindromeCheckerTest {
    private List<String> palindromes = Arrays.asList(
        "Madam, in Eden, I'm Adam",
        "Flee to me, remote elf!",
        "Go hang a salami; I'm a lasagna hog"
    );

    @Test
    public void isPalidromeUsingLambda() throws Exception {
        palindromes.forEach(s -> {
            StringBuilder sb = new StringBuilder();
            for (char c : s.toCharArray()) {
                if (Character.isLetter(c)) {
                    sb.append(c);
                }
            }
            String forward = sb.toString().toLowerCase();
            String backward = sb.reverse().toString().toLowerCa
            assertTrue(forward.equals(backward));
        });
```

```
    }
}
```

The test uses the default method `forEach` on the list to iterate over the collection, passing each string to the given lambda expression. At the end of the expression is the static `assertTrue` method (note the static import) that checks the results. The `forEach` method takes a `Consumer` as an argument, which means the supplied lambda expression must take one argument and return nothing.

Since we know an implementation, we can add it to the interface itself, as in Example 1-26.

Example 1-26. A palindrome checker with a static implementation method

```
@FunctionalInterface
public interface PalindromeChecker {
    boolean isPalidrome(String s);

    static boolean checkPalindrome(String s) {
        StringBuilder sb = new StringBuilder();
        for (char c : s.toCharArray()) {
            if (Character.isLetter(c)) {
                sb.append(c);
            }
        }
        String forward = sb.toString().toLowerCase();
        String backward = sb.reverse().toString().toLowerCase()
        return forward.equals(backward);
    }
}
```

The existence of the static method doesn’t change the fact that the interface is functional. To test this, see the test method in Example 1-27.

Example 1-27. Added test method to check the implementation

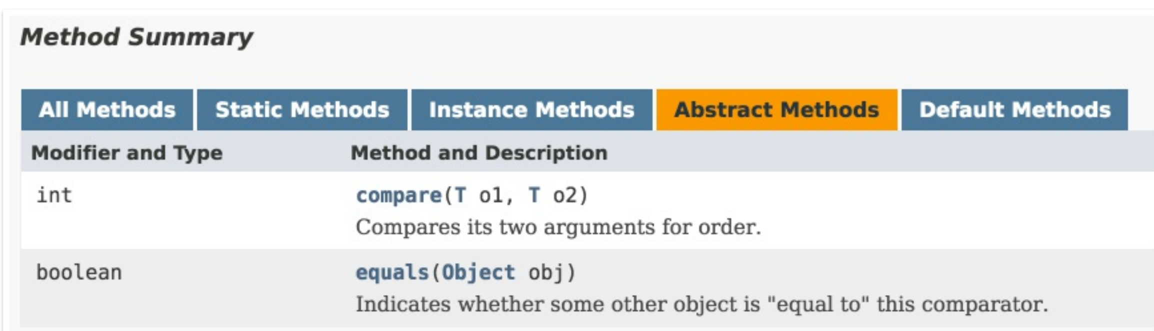
```
@Test
public void isPalidromeUsingMethodRef() throws Exception {
    assertTrue(
        palindromes.stream()
            .allMatch(PalindromeChecker::checkPalindrome));
```

```
    assertFalse(
        PalindromeChecker.checkPalindrome("This is NOT a palin
}
```

The `allMatch` method on `Stream` takes a `Predicate`, another of the functional interfaces in the `java.util.function` package. A `Predicate` takes a single argument and returns a boolean. The `checkPalindrome` method satisfies this requirement, and here is accessed using a method reference.

The `allMatch` method returns true only if every element of the stream satisfies the predicate. Just to make sure the tested implementation doesn't simply return `true` for all cases, the `assertFalse` test checks a string that isn't a palindrome.

One edge case should also be noted. The `Comparator` interface is used for sorting, which is discussed in other recipes. If you look at the JavaDocs for that interface and select the `Abstract Methods` tab, you see the methods shown in Figure 1-1.


Method Summary

All Methods | Static Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
int	compare(T o1, T o2) Compares its two arguments for order.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this comparator.

Figure 1-1. Abstract methods in the Comparator class

Wait, what? How can this be a functional interface if there are two abstract methods, especially if one of them is actually implemented in `java.lang.Object`?

As it turns out, this has always been legal. You can declare methods in `Object` as abstract in an interface, but that doesn't make them abstract. Usually the reason for doing so is to add documentation that explains the contract of the interface. In the case of `Comparator`, the contract is that if two elements return `true` from the `equals` method, the `compare` method should return zero. Adding the `equals` method to `Comparator` allows the associated JavaDocs to explain that.

From a Java 8 perspective, this fortunately means that methods from `Object` don't count against the single abstract method limit, and `Comparator` is still a functional interface.

See Also

Static Methods In Interfaces

Problem

You want to add a class-level utility method to an interface, along with an implementation.

Solution

Make the method `static` and provide the implementation in the usual way.

Discussion

Static members of Java classes are class-level, meaning they are associated with the class as a whole rather than with a particular instance. That makes their use in interfaces problematic from a design point of view. Some of the questions include:

- What does a class-level member mean when the interface is implemented by many different classes?
- Does a class need to implement an interface in order to use a static method in it?
- Static methods in classes are accessed by the class name. If a class implements an interface, does a static method get called from the class name or the interface name?

The designers of Java could have decided these questions in several different ways. Prior to Java 8, the decision was not to allow static members in interfaces at all.

Unfortunately, however, that lead to the creation of *utility* classes: classes that contain only static methods. A typical example is `java.util.Collections`, which contains methods for sorting and searching, wrapping collections in synchronized or unmodifiable types, and more. In the NIO package, `java.nio.file.Paths` is another example, which contains only static methods that parse `Path` instances from strings or URIs.

Now, in Java 8, you can add static methods to interfaces whenever you like. The requirements are:

- Add the `static` keyword to the method.
- Provide an implementation (which cannot be overridden). In this they are like `default` methods, and are included in the default tab in the JavaDocs.

- Access the method using the interface name. Classes do **not** need to implement an interface to use its static methods.

One example of a convenient static method in an interface is the `comparing` method in `java.util.Comparator`, along with its primitive variants `comparingInt`, `comparingLong`, and `comparingDouble`. The `Comparator` interface also has static methods `naturalOrder` and `reverseOrder`. Example 1-28 shows how they are used.

Example 1-28. Sorting strings

```
List<String> bonds = Arrays.asList("Connery", "Lazenby", "Moore
    "Dalton", "Brosnan", "Craig");

// Sorted in natural order
List<String> sorted = bonds.stream()
    .sorted(Comparator.naturalOrder())  // same as "sorted()"
    .collect(Collectors.toList());
// [Brosnan, Connery, Craig, Dalton, Lazenby, Moore]

// Sorted in the reverse of the natural order
sorted = bonds.stream()
    .sorted(Comparator.reverseOrder())
    .collect(Collectors.toList());
// [Moore, Lazenby, Dalton, Craig, Connery, Brosnan]

// Sorted by name, all lowercase
sorted = bonds.stream()
    .sorted(Comparator.comparing(String::toLowerCase))
    .collect(Collectors.toList());
// [Brosnan, Connery, Craig, Dalton, Lazenby, Moore]

// Sorted by length
sorted = bonds.stream()
    .sorted(Comparator.comparingInt(String::length))
    .collect(Collectors.toList());
// [Moore, Craig, Dalton, Connery, Lazenby, Brosnan]

// Sorted by length then natural order
sorted = bonds.stream()
    .sorted(Comparator.comparingInt(String::length)
        .thenComparing(Comparator.naturalOrder()))
    .collect(Collectors.toList());
// [Craig, Moore, Dalton, Brosnan, Connery, Lazenby]
```

The example shows how to use the static methods in `Comparator` to sort the list of actors who have played James Bond over the years.[3] Comparators are discussed further in a separate chapter.

Static methods in interfaces remove the need to create separate utility classes, though that option is still available if a design calls for it.

The key points to remember are:

- Static methods must have an implementation
- You can not override a static method
- Call static methods from the interface name
- You do not need to implement an interface to use its static methods

See Also

Static methods from interfaces are used throughout this book, but “Sorting Using A Comparator” covers the static methods from `Comparator` used here.

Default Methods In Interfaces

Problem

You want to provide an implementation of a method inside an interface.

Solution

Use the keyword `default` on the interface method, and add the implementation in the normal way.

Discussion

The traditional reason Java never supported multiple inheritance is the so-called *diamond problem*. Say you have an inheritance hierarchy as shown in the (vaguely UML-like) Figure 1-2.

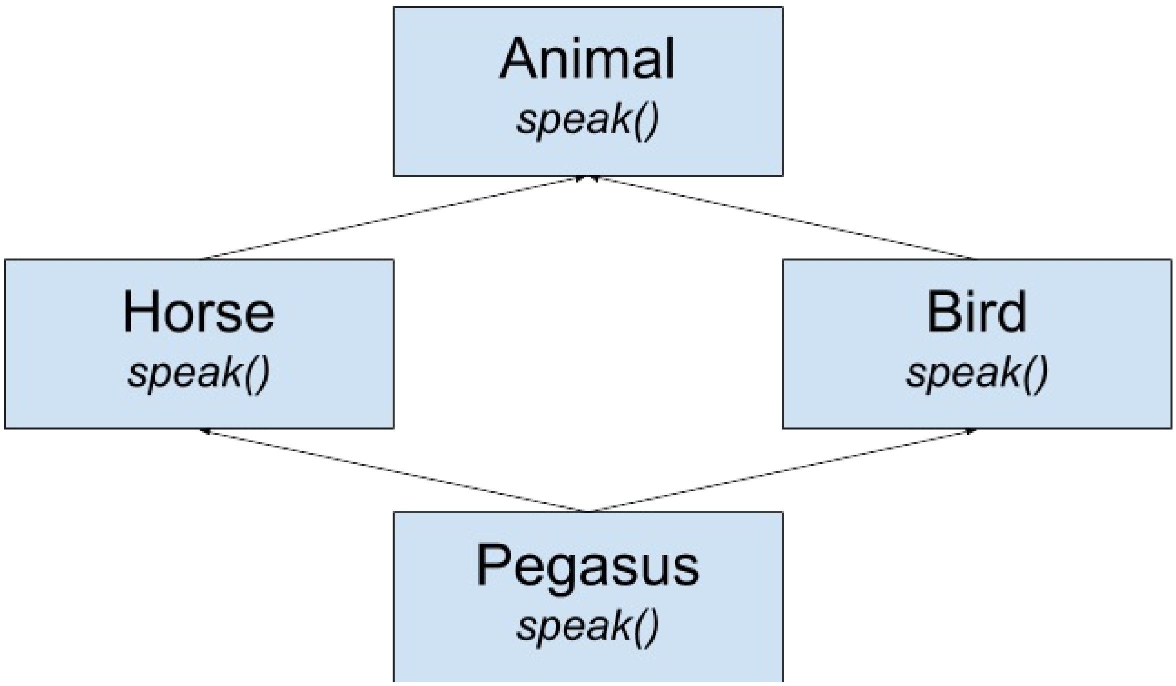


Figure 1-2. Animal Inheritance

Class `Animal` has two child classes, `Bird` and `Horse`, each of which overrides the `speak` method from `Animal`, in `Horse` to say “whinny” and in `Bird` to say “chirp”. What, then, does `Pegasus` (who multiply inherits from both `Horse` and `Bird`)[4] say? What if you have a reference of type `Animal` assigned to an instance of `Pegasus`? What then should the `speak` method return?

```
Animal animal = new Pegaus();
animal.speak(); // whinny, chirp, or other?
```

Different languages take different approaches to this problem. In C++, for example, multiple inheritance is allowed, but if a class inherits conflicting implementations, it won’t compile. In Eiffel[5], the compiler allows you to choose which implementation you want.

Java's approach was to prohibit multiple inheritance, and interfaces were introduced as a work-around. Since interfaces had only abstract methods, there were no implementations to conflict. Multiple inheritance is even allowed with interfaces, but again that works because only the method signatures are inherited.

The problem is, if you can never implement a method in an interface, you wind up with some awkward designs. For example, among the methods in the `java.util.Collection` interface are:

```
boolean isEmpty()
int     size()
```

The `isEmpty` method returns true if there are no elements in the collection, and false otherwise. The `size` method returns the number of elements in the collections. Regardless of the underlying implementation, you can immediately implement the `isEmpty` method in terms of `size`, as in Example 1-29.

Example 1-29. Implementation of isEmpty in terms of size

```
public boolean isEmpty() {
    return size() == 0;
}
```

Since `Collection` is an interface, you can't do this in the interface itself. Instead, the standard library includes an abstract class called `java.util.AbstractCollection`, which includes, among other code, exactly the implementation of `isEmpty` shown here. If you are creating your own collection implementation and you don't already have a superclass, you can extend `AbstractCollection` and you get the `isEmpty` method for free. If you already have a superclass, you have to implement the `Collection` interface instead and remember to provide your own implementation of `isEmpty` as well as `size`.

All of this is quite familiar to experienced Java developers, but as of Java 8 the situation changes. Now you can add implementations to interface methods. All you have to do is add the keyword `default` to a method and provide an implementation. The example in Example 1-30 shows an interface with both abstract and default methods.

Example 1-30. An `Employee` interface with a default method

```
public interface Employee {
    String getFirst();

    String getLast();

    void convertCaffeineToCodeForMoney();

    default String getName() {   ❶
        return String.format("%s %s", getFirst(), getLast());
    }
}
```

❶

default method

The `getName` method has the keyword `default`, and its implementation is in terms of the other, abstract, methods in the interface, `getFirst` and `getLast`.

Many of the existing interfaces in Java have been enhanced with default methods. For example, `Collection` now contains the following default methods:

```
default boolean        removeIf(Predicate<? super E> filter)
default Stream<E>            stream()
default Stream<E>            parallelStream()
default Spliterator<E> spliterator()
```

The `removeIf` method removes all elements from the collection that satisfy the `Predicate` argument, returning `true` if any elements were removed. The `stream` and `parallelStream` methods are factory methods for creating streams, and are discussed elsewhere in this book. The `spliterator` method returns an object from a class that implements the `Spliterator` interface, which is an object for traversing and partitioning elements from a source.

Default methods are used the same way any other methods are used, as

Example 1-31 shows.

Example 1-31. Using default methods

```
List<Integer> nums = Arrays.asList(3, 1, 4, 1, 5, 9);
boolean removed = nums.removeIf(n -> n <= 0);
System.out.println("Elements were " + (removed ? "" : "NOT") +

// Iterator has a default forEach method
nums.forEach(System.out::println);
```

What happens when a class implements two interfaces with the same default method? That is the subject of "Default Method Conflict", but the short answer is that if the class implements the method itself everything is fine. See "Default Method Conflict" for details.

See Also

“Default Method Conflict” shows the rules that apply when a class implements multiple interfaces with default methods.

[1] Coined by Gordon Moore, one of the co-founders of Fairchild Semiconductor and Intel, based on the observation that the number of transistors that could be packed into an integrated circuit doubled roughly every 18 months. See *https://en.wikipedia.org/wiki/Moore%27s_law* for details

[2] I mean no disrespect by treating Admiral Hopper as an object. I have no doubt she could still kick my a**, and she passed away in 1992.

[3] The temptation to add Idris Elba to the list is almost overwhelming, but no such luck as yet.

[4] “A magnificent horse, with the brain of a bird.” (Disney’s *Hercules* movie, which is fun if you pretend you know nothing about Greek mythology and never heard of Hercules)

[5] There’s an obscure reference for you, but Eiffel was one of the foundational languages of Object-Oriented Programming. See the book “Object Oriented Software Construction” (note: check ref) for details

Chapter 2. The java.util.function Package

The previous chapter discussed the basic syntax of lambda expressions and method references. One basic principle is that for either, there is always a context. Lamdba expressions and method references are always assigned to a functional interface, which provides information about the single abstract method being implemented.

While many interfaces in the Java standard library contain only a single, abstract method and are thus functional interfaces, there is a new package that was specifically designed that way. The Java SE 8 API added the `java.util.function` package, which contains a set of interfaces intended to support the use of functional programming in the rest of the library.

As the recipes in this section will make clear, the interfaces in `java.util.function` fall into four categories: consumers, suppliers, predicates, and functions. For example, for each basic interface `Consumer` contains a method that takes a single generic parameter and returns `void`. For that interface, there are also variations customized for primitive types (`IntConsumer`, `LongConsumer` and `DoubleConsumer`) and a variation that takes two arguments and returns `void` Each of the families in this package follow the same pattern.

Although by definition the interfaces in this section only contain a single abstract method, most also include additional methods that are either `static` or `default`. Becoming familiar with these methods will make your job as a developer easier.

Consume Data with java.util.function.Consumer

Problem

You want to write lambda expressions that implement `java.util.function.Consumer`, or use existing implementations of that interface.

Solution

Implement the `void accept(T t)` method using a lambda expression or a method reference.

Discussion

The `java.util.function.Consumer` interface has as its single, abstract method, `void accept(T t)`. See Example 2-1.

Example 2-1. Methods in java.util.function.Consumer

```
        void          accept(T t)   ❶
default Consumer<T> andThen(Consumer<? super T> after) ❷
```

❶

Single abstract method

❷

Default method for composition

The `accept` method takes a generic argument and returns `void`. The standard example is the default `forEach` method in `java.util.Iterable` (Example 2-2):

Example 2-2. The forEach method in Iterable

```
default void forEach(Consumer<? super T> action)
```

All linear collections implement this interface by performing the given action for each element of the collection, as in Example 2-3.

Example 2-3. Printing the elements of a collection

```
List<String> strings = Arrays.asList("this", "is", "a", "list",

strings.forEach(new Consumer<String>() { ❶
    @Override
    public void accept(String s) {
        System.out.println(s);
    }
```

```
});

strings.forEach(s -> System.out.println(s));   ❷
strings.forEach(System.out::println);          ❸
```

❶

Anonymous inner class implementation

❷

Expression lambda

❸

Method reference

The lambda expression conforms to the signature of the `accept` method, in that it takes a single argument and returns nothing. The `println` method in `PrintStream`, accessed here via `System.out`, also is compatible with `Consumer`. Therefore, either can be used as the target for an argument of type `Consumer`.

The `java.util.function` package also contains primitive variations of `Consumer<T>`, as well as a two-argument version. See Table 2-1 for details.

Table 2-1. Additional Consumer interfaces

Interface	**Single abstract method**
`IntConsumer`	`void accept(int x)`
`DoubleConsumer`	`void accept(double x)`
`LongConsumer`	`void accept(long x)`
`BiConsumer`	`void accept(T t, U u)`

Tip

Consumers are expected to operate via side effects, as shown in the printing example.

The `BiConsumer` interface has an `accept` method that takes two generic arguments, which are assumed to be of different types. The package contains three variations on `BiConsumer` where the second argument is a primitive. One is `ObjIntConsumer`, whose `accept` method takes two arguments, a generic and and an `int`. `ObjLongConsumer` and `ObjDoubleConsumer` are defined similarly.

Other uses of the `Consumer` interface in the standard library include:

- `Optional.ifPresent(Consumer<? super T> consumer)` — if a value is present, invoke the specified consumer. Otherwise do nothing.

- `Stream.forEach(Consumer<? super T> action)` — performs an action for each element of the stream (`Stream.forEachOrdered` is similar, accessing elements in encounter order)

- `Stream.peek(Consumer<? super T> action)` — returns a stream with the same elements as the existing stream, first performing the given action. This is a very useful technique for debugging (see "Debugging Streams with peek" for an example)

See Also

The `andThen` method in `Consumer` is used for composition. Function composition is discussed further in "Closure Composition". The `peek` method in `Stream` is examined in "Debugging Streams with peek".

Supply data with java.util.function.Supplier

Problem

You want to implement the `java.util.function.Supplier` interface.

Solution

Implement the `T get()` method in `java.util.function.Supplier` using a lambda expression or a method reference.

Discussion

The `java.util.function.Supplier` interface is particularly simple. It does not have any static or default methods. It contains only a single, abstract method, `T get()`.

Implementing `Supplier` means providing a method that take no arguments and returns the generic type. As stated in the Java docs, there is no requirement that a new or distinct result be returned each time the supplier is invoked.

One simple example of a supplier is the `Math.random` method, which takes no arguments and returns a `double`. That can be assigned to a `Supplier` reference and invoked at any time, as in Example 2-4.

Example 2-4. Using Math.random() as a supplier

```
Logger logger = Logger.getLogger("...");

DoubleSupplier randomSupplier = new DoubleSupplier() { ❶
    @Override
    public double getAsDouble() {
        return Math.random();
    }
};

randomSupplier = () -> Math.random(); ❷
randomSupplier = Math::random; ❸

logger.info(randomSupplier);
```

❶

Anonymous inner class implementation

❷

Expression lambda

❸

Method reference

The single abstract method in `DoubleSupplier` is `getAsDouble`, which returns a `double`. The other associated supplier interfaces in the `java.util.function` package are shown in Table 2-2.

Table 2-2. Additional Supplier interfaces

Interface	Single abstract method
`IntSupplier`	`int getAsInt()`
`DoubleSupplier`	`double getAsDouble()`
`LongSupplier`	`long getAsLong()`
`BooleanSupplier`	`boolean getAsBoolean()`

One of the primary use cases for suppliers is to support the concept of *deferred execution*, and the use in Example 2-4 shows an example. The `info` method in `java.util.logging.Logger` takes a `Supplier`, whose `get` method is only called if the log level means the message will be seen. This process of deferred execution can be used in your own code, to ensure that a value is retrieved from a supplier only when appropriate.

Another example from the standard library is the `orElseGet` method in `Optional`, which also takes a supplier. The `Optional` class is discussed in other recipes in this book, but the short explanation is that an `Optional` is returned by methods in the library that may reasonably expect to have no result.

Consider searching for a name from a collection, as shown in Example 2-5.

Example 2-5. Finding a name from a collection

```
List<String> names = Arrays.asList("Mal", "Wash", "Kaylee", "In
    "Zoë", "Jayne", "Simon", "River", "Shepherd Book");
```

```
Optional<String> first = names.stream()
    .filter(name -> name.startsWith("C"))
    .findFirst();

System.out.println(first);                  ❶
System.out.println(first.orElse("None")); ❷

System.out.println(first.orElse(String.format("No result found
    names.stream().collect(Collectors.joining(", "))))); ❸

System.out.println(first.orElseGet(() ->
    String.format("No result found in %s",
    names.stream().collect(Collectors.joining(", "))))); ❹
```

❶

Prints Optional.empty

❷

Prints the string “None”

❸

Forms the comma-separated collection, even when name is found

❹

Forms the comma-separated collection only if the optional is empty

The `findFirst` method on `Stream` returns the first element, processed sequentially. Since it’s possible that there are no elements remaining in the stream, the method returns an `Optional`. That optional either contains the desired element, or is empty. In this case, none of the names in the list pass the filter, so the result is an empty optional.

The `orElse` method on `Optional` returns either the contained element, or a specified default. That’s fine if the default is a simple string, but can be wasteful if processing is necessary to return a value.

In this case, the returned value shows the complete list of names in comma-separated form. The `orElse` method creates the complete string, whether the optional contains a value or not.

The `orElseGet` method, however, takes a `Supplier` as an argument. The advantage is that the `get` method on the `Supplier` will only be invoked when the optional is empty, so the complete name string is not formed unless it is necessary.

Other examples from the standard library that use suppliers include:

- The `orElseThrow` method in `Optional`, which takes a `Supplier<X extends Exception>`. The supplier is only executed if an exception occurs

- `Objects.requireNonNull(T obj, Supplier<String> messageSupplier)` only customizes its response if the first argument is null

- `CompletableFuture.supplyAsync(Supplier<U> supplier)` returns a `CompletableFuture` that is asynchronously completed by a task running with the value obtained by calling the given `Supplier`

- The `Logger` class has overloads for all its logging methods that takes a `Supplier<String>` rather than just a string (used as an example in [Link to Come])

See Also

Using the overloaded logging methods that take a supplier is discussed in "Logging with a Supplier".

Filter Data With Predicates

Problem

You want to implement the `java.util.function.Predicate` interface.

Solution

Implement the `boolean test(T t)` method in the `Predicate` interface using a lambda expression or a method reference.

Discussion

Predicates are used primarily to filter streams. Given a stream of items, the `filter` method in `java.util.stream.Stream` takes a `Predicate` and returns a new stream that includes only the items that match the given predicate.

The single abstract method in `Predicate` is `boolean test(T t)`, which takes a single generic argument and returns true or false. The complete set of methods in `Predicate`, including state and defaults, is given in Example 2-6.

Example 2-6. Methods in java.util.function.Predicate

```
default     Predicate<T> and(Predicate<? super T> other)
static <T> Predicate<T> isEquals(Object targetRef)
default     Predicate<T> negate()
default     Predicate<T> or(Predicate<? super T> other)
boolean     test(T t)
```

Say you have a collection of names and you want to find all the instances that have a particular length. Example 2-7 shows an example of how to use stream processing to do so.

Example 2-7. Finding strings of a given length

```
public String getNamesOfLength5(String... names) {
    return Arrays.stream(names)
        .filter(s -> s.length() == 5) ❶
        .collect(Collectors.joining(", "));
}
```

❶

Predicate for strings of length 5 only

Alternatively, perhaps you want only the names that start with a particular letter, as in Example 2-8.

Example 2-8. Finding strings that start with a given letter

```
public String getNamesStartingWithS(String... names) {
    return Arrays.stream(names)
        .filter(s -> s.startsWith("S")) ❶
        .collect(Collectors.joining(", "));
}
```

❶

Predicate to return strings starting with `s`

Both of these examples have hard-wired values for the filter. It's more likely that the condition will be specified by the client. Example 2-9 shows a method to do that.

Example 2-9. Finding strings that satisfy an arbitrary predicate

```
public String getNamesSatisfyingCondition(Predicate<String> cor
    return Arrays.stream(names)
        .filter(condition)  ❶
        .collect(Collectors.joining(", "));
}
```

❶

Filter by supplied predicate

This is quite flexible, but it may be a bit much to expect the client to write every predicate themselves. One option is to add constants to the class representing the most common cases, as in Example 2-10.

Example 2-10. Adding constants for common cases

```
public class ImplementPredicate {
    public static final Predicate<String> LENGTH_FIVE = s -> s.
    public static final Predicate<String> STARTS_WITH_S =
        s -> s.startsWith("S");

    // ... rest as before ...
}
```

The other advantage to supplying a predicate as an argument is that you can also use the default methods `and`, `or`, and `negate` to create a composite predicate from a series of individual elements.

The test case in Example 2-11 demonstrates all of these techniques.

Example 2-11. JUnit 4 test for predicate methods

```
package functionpackage;

import org.junit.Before;
import org.junit.Test;

import java.util.stream.Stream;

import static functionpackage.ImplementPredicate.*; ❶
import static org.junit.Assert.assertEquals;

public class ImplementPredicateTest {
    private ImplementPredicate demo = new ImplementPredicate();
    private String[] names;

    @Before
    public void setUp() {
        names = Stream.of("Mal", "Wash", "Kaylee", "Inara", "Zo
            "Jayne", "Simon", "River", "Shepherd Book")
            .sorted()
            .toArray(String[]::new);
    }

    @Test
    public void getNamesOfLength5() throws Exception {
        assertEquals("Inara, Jayne, River, Simon",
            demo.getNamesOfLength5(names));
    }

    @Test
    public void getNamesStartingWithS() throws Exception {
        assertEquals("Shepherd Book, Simon", demo.getNamesStart
    }

    @Test
    public void getNamesSatisfyingCondition() throws Exception
        assertEquals("Inara, Jayne, River, Simon",
            demo.getNamesSatisfyingCondition(s -> s.length() ==
        assertEquals("Shepherd Book, Simon",
            demo.getNamesSatisfyingCondition(s -> s.startsWith
            names));
```

```
        assertEquals("Inara, Jayne, River, Simon",
            demo.getNamesSatisfyingCondition(LENGTH_FIVE, names
        assertEquals("Shepherd Book, Simon",
            demo.getNamesSatisfyingCondition(STARTS_WITH_S, nar
    }

    @Test
    public void composedPredicate() throws Exception {
        assertEquals("Simon",
            demo.getNamesSatisfyingCondition(
                LENGTH_FIVE.and(STARTS_WITH_S), names));  ❷
        assertEquals("Inara, Jayne, River, Shepherd Book, Simor
            demo.getNamesSatisfyingCondition(
                LENGTH_FIVE.or(STARTS_WITH_S), names));   ❷
        assertEquals("Kaylee, Mal, Shepherd Book, Wash, Zoë",
            demo.getNamesSatisfyingCondition(LENGTH_FIVE.negate
    }
}
```

❶

static import to make using constants simpler

❷

composition

❸

negation

Other methods in the standard library that use predicates include:

- `Optional.filter(Predicate<? super T> predicate)` — if a value is present, and the value matches the given predicate, return an optional describing the value, otherwise return an empty `Optional`
- `Collection.removeIf(Predicate<? super E> filter)` — removes all elements of this collection that satisfy the predicate
- `Stream.allMatch(Predicate<? super T> predicate)` — return true if all elements of the stream satisfy the given predicate. The methods `anyMatch` and `noneMatch` work similarly

- `Collectors.partitioningBy(Predicate<? super T> predicate)` — returns a `Collector` which splits a stream into two categories: those that satisfy the predicate and those that do not

Predicates are useful whenever a stream should only return certain elements. This recipe hopefully gives you an idea where and when that might be useful.

See Also

Closure composition is also discussed in “Closure Composition”.

Implementing java.util.function.Function

Problem

You need to implement the `java.util.function.Function` interface.

Solution

Provide a lambda expression that implements the `R apply(T t)` method.

Discussion

The functional interface `java.util.function.Function` contains the single abstract method `apply`, which is invoked to transform a generic input argument of type `T` into a generic output argument of type `R`. The methods in `Function` are shown in Example 2-12.

Example 2-12. Methods in the java.util.function.Function interface

```
default <V> Function<T,V> andThen(Function<? super R,? extends
            R             apply(T t)
default <V> Function<V,R> compose(Function<? super V,? extends
static  <T> Function<T,T> identity()
```

The most common usage of `Function` is as an argument to the `Stream.map` method. For example, one way to transform a `String` into an integer would be to invoke the length method on each instance, as in Example 2-13.

Example 2-13. Mapping strings to their lengths

```
List<String> names = Arrays.asList("Mal", "Wash", "Kaylee", "In
        "Zoë", "Jayne", "Simon", "River", "Shepherd Book");

// anonymous inner class
List<Integer> nameLengths = names.stream()
        .map(new Function<String, Integer>() {
            @Override
            public Integer apply(String s) {
                return s.length();
            }
        })
        .collect(Collectors.toList());

// lambda expression
nameLengths = names.stream()
        .map(s -> s.length())
        .collect(Collectors.toList());

// method reference
nameLengths = names.stream()
```

```
        .map(String::length)
        .collect(Collectors.toList());

System.out.printf("nameLengths = %s%n", nameLengths);
// nameLengths == [3, 4, 6, 5, 3, 5, 5, 5, 13]
```

The argument to the `map` method here could have been a `ToIntFunction`, because the return type on the method is an `int` primitive. The `Stream.mapToInt` method takes a `ToIntFunction` as an argument, and `mapToDouble` and `mapToLong` are analogous. The return types on `mapToInt`, `mapToDouble` and `mapToLong` are `IntStream`, `DoubleStream`, and `LongStream`, respectively.

The complete list of primitive variations for both the input and the output generic types are shown in Table 2-3.

Table 2-3. Additional Function interfaces

Interface	Single abstract method
`IntFunction`	`R apply(int value)`
`DoubleFunction`	`R apply(double value)`
`LongFunction`	`R apply(long value)`
`ToIntFunction`	`int applyAsInt(T value)`
`ToDoubleFunction`	`double applyAsDouble(T value)`
`ToLongFunction`	`long applyAsLong(T value)`
`DoubleToIntFunction`	`int applyAsInt(double value)`
`DoubleToLongFunction`	`long applyAsLong(double value)`

`IntToDoubleFunction`	`double applyAsDouble(int value)`
`IntToLongFunction`	`long applyAsLong(int value)`
`LongToDoubleFunction`	`double applyAsDouble(long value)`
`LongToIntFunction`	`int applyAsInt(long value)`
`BiFunction`	`void accept(T t, U u)`

What if the argument type and the return type are the same? The `java.util.function` package defines a `UnaryOperator` for that. As you might expect, there are also interfaces called `IntUnaryOperator`, `DoubleUnaryOperator`, and `LongUnaryOperator`, where the input and output arguments are `int`, `double`, and `long`, respectively. An example of a `UnaryOperator` would be the `reverse` method in `StringBuilder`, because both the input type and the output type are strings.

The `BiFunction` interface is defined for two generic input types and one generic output type, all of which are assumed to be different. If all three are the same, the package includes the `BinaryOperator` interface. An example of a binary operator would be `Math.max`, because both inputs and the output are either `int`, `double`, `float`, or `long`. Of course, the interface also defines interfaces called `IntBinaryOperator`, `DoubleBinaryOperator`, and `LongBinaryOperator` for those situations.

To complete the set, the package also has primitive variations of `BiFunction`, which are summarized in Table 2-4.

Table 2-4. Additional BiFunction interfaces

Interface	**Single abstract method**

`ToIntBiFunction`	`int applyAsInt(T t, U u)`
`ToDoubleBiFunction`	`double applyAsDouble(T t, U u)`
`ToLongBiFunction`	`long applyAsLong(T t, U u)`

(It's probably a good thing that the API doesn't also include primitive variations for each of the `BiFunction` arguments. There are enough variations on this theme already.)

While the various `Stream.map` methods are the primary usages of `Function`, they do appear in other contexts. Among them are:

- `Map.computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)` — if the specified key does not have a value, use the provided `Function` to compute one and add it to a map
- `Comparator.comparing(Function<? super T,? extends U> keyExtractor)` — discussed in the section on sorting, this method generates a `Comparator` that sorts a collection by the key generated from the given `Function`
- `Comparator.thenComparing(Function<? super T,? extends U> keyExtractor)` — an instance method, also used in sorting, that adds an additional sorting mechanism if the collection has equal values by the first sort

Functions are also used extensively in the `Collectors` utility class for grouping and downstream collectors.

The `andThen` and `compose` methods are discussed in "Closure Composition". The `identity` method is simply the lambda expression `e -> e`. One usage is shown in "Adding a Linear Collection to a Map".

See Also

See “Closure Composition” for examples of the `andThen` and `compose` methods in the `Function` interface. See “Adding a Linear Collection to a Map” for an example of `Function.identity`. See “Downstream Collectors” for examples of using functions as downstream collectors.

Chapter 3. Functional Programming in Java 8

The recipes in this chapter describe how to combine lambda expressions and method references to accomplish the new functional idioms now supported by Java. The goal is to make your code more readable, easier to write and understand, and, hopefully, parallelizable.

Default Method Conflict

Problem

You have a class that implements two interfaces, each of which that contain the same default method, but with different implementations.

Solution

Implement the method in your class. Your implementation can still use the provided defaults from the interfaces.

Discussion

Java 8 supports both static and default methods in interfaces. Default methods provide an implementation, which is then inherited by the class. This allows interfaces to add new methods without breaking existing class implementations.

Since classes can implement multiple interfaces, a class may inherit default methods that have the same signature but are implemented differently, or it may already contain its own version of a default method.

There are three possibilities when this occurs:

1. In any conflict between a method in a class and a default method in an interface, the class always wins

2. If the conflict comes between two interfaces where one is a descendent of the other, then the child wins, the same way implementations in child classes override those in their parents

3. If there is no inheritance relationship between the two defaults, the class will not compile.

In the last case, simply implement the method in the class and everything will work again. This reduces the third case to the first one.

As an example, consider the `Company` interface shown in Example 3-1 and the `Employee` interface shown in Example 3-2.

Example 3-1. The Company interface with a default method

```
public interface Company {
    default String getName() {
        return "Company Name";
    }

    // other methods
```

```
}
```

The `default` keyword indicates that the `getName` method is a default method, which provides an implementation that returns the company name.

Example 3-2. The Employee interface with a default method

```
public interface Employee {
    String getFirst();

    String getLast();

    void convertCaffeineToCodeForMoney();

    default String getName() {
        return String.format("%s %s", getFirst(), getLast());
    }
}
```

The `Employee` interface also contains a default method called `getName` with the same signature as the one in `Company`, but with a different implementation. The `CompanyEmployee` class shown in Example 3-3 implements both interfaces, causing a conflict.

Example 3-3. First attempt at CompanyEmployee (WON'T COMPILE)

```
public class CompanyEmployee implements Company, Employee {
    private String first;
    private String last;

    @Override
    public void convertCaffeineToCodeForMoney() {
        System.out.println("Coding...");
    }

    @Override
    public String getFirst() {
        return first;
    }

    @Override
    public String getLast() {
        return last;
```

```
    }
}
```

Since `CompanyEmployee` inherits unrelated defaults for `getName`, the class won't compile. To fix this, you need to add your own version of `getName` to the class, which will then override both the defaults.

You can still use the provided defaults, however, using the `super` keyword, as shown in Example 3-4.

Example 3-4. Fixed version of CompanyEmployee

```
public class CompanyEmployee implements Company, Employee {
    private String first;
    private String last;

    @Override
    public String getName() { ❶
        return String.format("%s working for %s",
            Employee.super.getName(), Company.super.getName());
    }

    @Override
    public void convertCaffeineToCodeForMoney() {
        System.out.println("Coding...");
    }

    @Override
    public String getFirst() {
        return first;
    }

    @Override
    public String getLast() {
        return last;
    }
}
```

❶

Implement `getName`

❷

Access default implementations using `super`

In this version, the `getName` method in the class builds a `String` from the default versions provided by both `Company` and `Employee`.

The best news of all is that this is as complicated as default methods ever get. You now know everything there is to know about them.

Actually, there’s one edge case to consider. If the `Company` interface contained `getName` but was not marked `default` (and didn’t have an implementation, making it abstract), would that still cause a conflict because `Employee` also had the same method? The answer is yes, interestingly enough, and you still need to provide an implementation in the `CompanyEmployee` class.

Of course, if the same method appears in both interfaces and neither is a default, then this is the pre-Java 8 situation. There’s no conflict, but the class must provide an implementation.

See Also

Iterating Over A Collection

Problem

You want to iterate over a collection or map.

Solution

Use the `forEach` method, which was added as a default method to both `Iterable` and `Map`.

Discussion

Rather than using a loop to iterate over a linear collection (i.e., a class that implements `Collection` or one of its descendents), you can use the new `forEach` method that has been added to `Iterable` as a default method.

From the Javadocs, its signature is:

```
default void forEach(Consumer<? super T> action)
```

The argument to `forEach` is of type `Consumer`, one of the new functional interfaces added to the `java.util.function` package. A `Consumer` represents an operation that takes a single generic parameter and returns no result. As the docs say, “unlike most other functional interfaces, `Consumer` is expected to operate via side-effects.”

Note

A *pure* function operates without side-effects, so applying the function with the same arguments always gives the same result. In functional programming, this is known as *referential integrity*.

Since `java.util.Collection` is a sub-iterface of `Iterable`, the `forEach` method is available on all linear collections, like `ArrayList` and `HashSet`. Iterating over each is therefore quite simple, as Example 3-5 shows.

Example 3-5. Iterating over a linear collection

```
List<Integer> integers = Arrays.asList(3, 1, 4, 1, 5, 9);

    integers.forEach(new Consumer<Integer>() {   ❶
        @Override
        public void accept(Integer integer) {
            System.out.println(integer);
        }
    });
```

```
    integers.forEach((Integer n) -> {   ❷
        System.out.println(n);
    });

    integers.forEach(n -> System.out.println(n)); ❸

    integers.forEach(System.out::println);   ❹
}
```

❶

Anonymous inner class implementation

❷

Full verbose form of a block lambda

❸

Expression lambda

❹

Method reference

The anonymous inner class version is shown simply as a reminder of the signature of the `accept` method in the `Consumer` interface. As you can see, the `accept` method takes a single argument and returns `void`. The lambda versions shown are compatible with this. Since each of the lambda versions consists of a single call to the `println` method on `System.out`, that method can be used as a method reference, as shown in the last version.

Interestingly enough, the `Map` interface also as a `forEach` method, again added as a default. In this case, the signature takes a `BiConsumer`.

```
default void forEach(BiConsumer<? super K, ? super V> action)
```

`BiConsumer` is another of the new interfaces in the `java.util.function` package. It represents a function that takes two generic arguments and returns `void`. When applied to the `forEach` method in `Map`, the arguments become the keys and values from the `Map.Entry` instances in the `entrySet`.

That means iterating over a `Map` is now as easy as iterating over a `List`, `Set`, or any other linear collection. Example 3-6 shows an example.

Example 3-6. Iterating over a map

```
Map<Long, String> map = new HashMap<>();
map.put(86L, "Don Adams (Maxwell Smart)");
map.put(99L, "Barbara Feldon");
map.put(13L, "David Ketchum");

map.forEach((num, agent) ->
    System.out.printf("Agent %d, played by %s%n", num, agent));
```

The output from the iteration is shown in Example 3-7.

Example 3-7. Map iteration output

```
Agent 99, played by Barbara Feldon
Agent 86, played by Don Adams (Maxwell Smart)
Agent 13, played by David Ketchum
```

Prior to Java 8, to iterate over a map you needed to first use the `keySet` or `entrySet` methods to acquire the `Set` of keys or `Map.Entry` instances and then iterate over that. With the new default `forEach` method, iteration is much simpler.

See Also

The functional interfaces `Consumer` and `BiConsumer` are discussed in "Consume Data with java.util.function.Consumer".

Logging with a Supplier

Problem

You want to create a log message, but only if the log level ensures it will be seen.

Solution

Use the new logging overloads in the `Logger` class that take a `Supplier`.

Discussion

The logging methods in `java.util.logging.Logger`, like `info`, `warning`, or `severe`, now have two overloaded versions: one that takes a single `String` as an argument, and one that takes a `Supplier<String>`.

For example, Example 3-8 shows the signatures of the two `info` methods.

Example 3-8. Logging methods at info level in java.util.logging.Logger

```
void info(String msg)
void info(Supplier<String> msgSupplier)
```

The other logging methods are similar. The version that takes a `String` is part of the original definition that appeared in Java 1.4 and later. The `Supplier` version is new to Java 8. If you look at the implementation of the `Supplier` version in the standard library, you see the code shown in Example 3-9.

Example 3-9. Implementation details of the Logger class

```
public void info(Supplier<String> msgSupplier) {
    log(Level.INFO, msgSupplier);
}

public void log(Level level, Supplier<String> msgSupplier) {
    if (!isLoggable(level)) {  ❶
        return;
    }
    LogRecord lr = new LogRecord(level, msgSupplier.get()); ❷
    doLog(lr);
}
```

❶

Return if the log level is such that the message will not be shown

❷

Retrieve the message from the Supplier by calling `get`

Rather than construct a message that will never be shown, the implementation checks to see if the message will be “loggable”. If the message was provided as a simple string, it would be evaluated whether it was logged or not. The version that uses a `Supplier` allows the developer to put empty parentheses and an arrow in front of the message, converting it into a `Supplier`, which will only be invoked if the log level is appropriate. Example 3-10 shows how to use both overloads.

Example 3-10. Using a Supplier in the info method

```
private Logger logger = Logger.getLogger(this.getClass().getNar
private List<String> data = new ArrayList<>();

// ... populate list with data ...

logger.info("The data is " + data.toString());        ❶
logger.info(() -> "The data is " + data.toString()); ❷
```

❶

Argument always constructed

❷

Argument only constructed if log level shows info messages

In this example, the log message is going to show the `toString` values of every object in the list, and the resulting string will be formed whether the program is using a log level that shows info messages or not. By converting the log argument to a `Supplier` by simply adding `() ->` in front of it, the implementation code will only invoke the `get` method on the `Supplier` if the message will be used.

The technique of replacing an argument with a `Supplier` of the same type is known as *deferred execution*, and can be used in any context where object creation might be expensive.

See Also

Deferred execution is one of the primary use cases for `Supplier`. Suppliers are discussed in "Supply data with java.util.function.Supplier".

Closure Composition

Problem

You want to apply a series of small, independent functions consecutively.

Solution

Use the composition methods defined as defaults in the `Function`, `Consumer`, and `Predicate` interfaces.

Discussion

One of the benefits of functional programming is that you can create a set of small, reusable functions that you can combine to solve larger problems. To support this, the functional interfaces in the `java.util.function` package include methods to make composition easy.

For example, the `Function` interface has two default methods with the signatures shown in Example 3-11.

Example 3-11. Composition methods in java.util.function.Function

```
default <V> Function<V,R>       compose(Function<? super V,? ex
default <V> Function<T,V>       andThen(Function<? super R,? ex
```

The dummy arguments names in the JavaDocs indicate what each method does. The `compose` method applies its argument *before* the original function, while the `andThen` method applies its argument *after* the original function.

To demonstrate this, consider the trivial example shown in Example 3-12.

Example 3-12. Using the compose and andThen methods

```
Function<Integer, Integer> a2 = x -> x + 2;
Function<Integer, Integer> m3 = x -> x * 3;

Function<Integer, Integer> m3a2 = a2.compose(m3); ❶
Function<Integer, Integer> a2m3 = a2.andThen(m3); ❷

System.out.println("m3a2(1): " + m3a2.apply(1));
System.out.println("a2m3(1): " + a2m3.apply(1));
```

❶

First `m3`, then `a2`

❷

First `a2`, then `m3`

The `a2` function adds two to its argument. The `m3` function multiplies its argument by three. Since `m3a2` is made using `compose`, first the `m3` function is applied and then the `a2` function, whereas for `a2m3` using the `andThen` function does the opposite.

The results of applying each composite function gives:

```
m3a2(1): 5  // == (1 * 3) + 2
a2m3(1): 9  // == (1 + 2) * 3
```

The result of the composition is a function, so this process creates new operations that can be used later. Say, for example, you receive data as part of an HTTP request, which means it is transmitted in string form. You already have a method to operate on the data, but only if it's already a number. If this happens frequently, you can compose a function that parses the string data before applying the numerical operation. For example, see Example 3-13.

Example 3-13. Parse an integer from a string, then add 2

```
Function<Integer, Integer> a2 = x -> x + 2;
Function<String, Integer> parseThenAdd2 = a2.compose(Integer::p
System.out.println(parseThenAdd2.apply("1"));
```

The new function, `parseThenAdd2`, invokes the static `Integer.parseInt` method before adding two to the result. Going the other way, you can define a function that invokes a `toString` method after a numerical operation, as in Example 3-14.

Example 3-14. Add a number, then convert to a string

```
Function<Integer, Integer> a2 = x -> x + 2;
Function<Integer, String> plus2toString = a2.andThen(Object::to
System.out.println(plus2toString.apply(1).getClass().getName())
```

This operation returns a function that takes an `Integer` argument and returns a `String`.

The `Consumer` interface also has a method used for closure composition, as shown in Example 3-15.

Example 3-15. Closure composition with consumers

```
default Consumer<T> andThen(Consumer<? super T> after)
```

The JavaDocs for `Consumer` explain that the `andThen` method returns a composed `Consumer` that performs the original operation followed by the `Consumer` argument. If either operation throws an exception, it is thrown to the caller of the composed operation.

The `Predicate` interface has three methods that can be used to compose predicates, as shown in Example 3-16.

Example 3-16. Composition methods in the Predicate interface

```
default Predicate<T>    and(Predicate<? super T> other)
default Predicate<T>    negate()
default Predicate<T>    or(Predicate<? super T> other)
```

As you might expect the `and`, `or`, and `negate` methods are used to compose predicates using a logical and, a logical or, and a logical not operation. Each returns a composed predicate.

The composition approach can be used to build up complex operations from a small library of simple functions.[1]

See Also

The functional interfaces in the `java.util.function` package are discussed in detail in Chapter 2.

Using an Extracted Method for Exception Handling

Problem

Code in a lambda expression needs to throw an exception, but you do not want to clutter a block lambda with exception handling code.

Solution

Create a separate method that does the operation, handle the exception there, and invoke the extracted method in your lambda expression.

Discussion

A lambda expression is effectively the implementation of the single abstract method in a functional interface. As with anonymous inner classes, lambda expressions can only throw exceptions declared in the abstract method signature.

If the required exception is unchecked, the situation is relatively easy. The ancestor of all unchecked exceptions is `java.lang.RuntimeException`.[2] Like any Java code, a lambda expression can throw a runtime exception without declaring it or wrapping the code in a try/catch block. The exception is then propagated to the caller.

An an example, consider a method that divides all elements of a collection by a constant value, as shown in Example 3-17.

Example 3-17. A lambda expression that may throw an unchecked exception

```
public List<Integer> div(List<Integer> values, Integer factor)
    return values.stream()
        .map(n -> n / factor)   ❶
        .collect(Collectors.toList());
}
```

❶

Can throw an `ArithmeticException`

Integer division will throw an `ArithmeticException` (an unchecked exception) if the denominator is zero.[3] This will be propagated to the caller, as shown in ???.

```
List<Integer> values = Arrays.asList(30, 10, 40, 10, 50, 90);
List<Integer> scaled = demo.div(values, 10);
System.out.println(scaled);
// prints: [3, 1, 4, 1, 5, 9]

scaled = demo.div(values, 0);
```

```
System.out.println(scaled);
// throws ArithmeticException: / by zero
```

The client code invokes the `div` method, and if the divisor is zero, the lambda expression throws an `ArithmeticException`. The client can add a try/catch block inside the `map` method in order to handle the exception, but that leads to some seriously ugly code (see Example 3-18).

Example 3-18. Lambda expression with try/catch

```
public List<Integer> div(List<Integer> values, Integer factor)
    return values.stream()
        .map( n -> {
            try {
                return n / factor;
            } catch (ArithmeticException e) {
                e.printStackTrace();
            }
        })
        .collect(Collectors.toList());
}
```

This same process works even for checked exceptions, as long as the checked exception is declared in the functional interface.

It's generally a good idea to keep stream processing code as simple as possible, with the goal of writing one line per intermediate operation. In this case, you can simplify the code by extracting the function inside `map` into a method, and the stream processing could be done by calling it, as in Example 3-19.

Example 3-19. Extracting a lambda into a method

```
private Integer divide(Integer value, Integer factor) {
    try {
        return value / factor;
    } catch (ArithmeticException e) { ❶
        e.printStackTrace();
    }
}

public List<Integer> divUsingMethod(List<Integer> values, Inte
```

```
    return values.stream()
        .map(n -> divide(n, factor))   ❷
        .collect(Collectors.toList());
}
```

❶

Handle the exception here

❷

Stream code is simplified

As an aside, if the extracted method had not needed the `factor` value, the argument to `map` could have been simplified to a method reference.

The technique of extracting the lambda to a separate method has benefits as well. You can write tests for the extracted method (using reflection if the method is private), set break points in it, or any other mechanism normally associated with methods.

See Also

Lambda expressions with checked exceptions are discussed in “Checked Exceptions And Lambdas”. Using a generic wrapper method for exceptions is in “Using a Generic Exception Wrapper”.

Checked Exceptions And Lambdas

Problem

You have a lambda expression that throws a checked exception, and the abstract method in the functional interface you are implementing does not declare that exception.

Solution

Add a try/catch block to the lambda expression, or delegate to an extracted method to handle it.

Discussion

A lambda expression is effectively the implementation of the single abstract method in a functional interface. A lambda expression can therefore only throw checked exceptions declared in the signature of the abstract method.

Say you are planning to invoke a service using a URL and you need to form a query string from a collection of string parameters. The parameters need to be encoded in a way that allows them to used in a URL. Java provides a class for this purpose called, naturally enough, `java.net.URLEncoder`, which has a static `encode` method that takes and `String` and encodes it according to a specified encoding scheme.

In this case, what you would like to write is code like Example 3-20.

Example 3-20. URL encoding a collection of strings (NOTE: DOES NOT COMPILE)

```
public List<String> encodeValues(String... values) {
    return Arrays.stream(values)
        .map(s -> URLEncoder.encode(s, "UTF-8")))   ❶
        .collect(Collectors.toList());
}
```

❶

Throws `UnsupportedEncodingException`, which must be handled

The method takes a variable argument list of strings and tries to run each of them through the `UREncoder.encode` method under the recommended `UTF-8` encoding. Unfortunately, since that method throws a (checked) `UnsupportedEncodingException`, the code does not compile.

You might be tempted to simply declare that the `encodeValues` method throws that exception, but that doesn't work (see Example 3-21).

Example 3-21. Declaring the exception (ALSO DOES NOT COMPILE)

```
public List<String> encodeValues(String... values)
    throws UnsupportedEncodingException {   ❶
    return Arrays.stream(values)
        .map(s -> URLEncoder.encode(s, "UTF-8")))
        .collect(Collectors.toList());
}
```

❶

Throwing the exception from the surrounding method also DOES NOT COMPILE

The problem is that throwing an exception from a lambda is like building an entirely separate class with a method and throwing the exception from there. It helps to think of the lambda as the implementation of an anonymous inner class, because then it becomes clear that throwing the exception in the inner object still needs to be handled or declared there, not in the surrounding object. Code like that is shown in Example 3-22, which shows both the anonymous inner class version and the lambda expression version.

Example 3-22. URL encoding using try/catch (CORRECT)

```
// Anonymous inner class version
public List<String> encodeValuesAnonInnerClass(String... values
    return Arrays.stream(values)
        .map(new Function<String, String>() { // Anonymous inne
            @Override
            public String apply(String s) {   ❶
                try {
                    return URLEncoder.encode(s, "UTF-8");
                } catch (UnsupportedEncodingException e) {
                    e.printStackTrace();
                    return "";
                }
            }
        })
        .collect(Collectors.toList());
}

// Lambda expression version
public List<String> encodeValues(String... values) {
    return Arrays.stream(values)
        .map(s -> {   // lambda expression with try/catch
            try {
```

```
                return URLEncoder.encode(s, "UTF-8");
            } catch (UnsupportedEncodingException e) {
                e.printStackTrace();
                return "";
            }
        })
        .collect(Collectors.toList());
}
```

❶

Exception must be handled in the `apply` method

Here is the version that uses an extracted method for the encoding.

Example 3-23. URL encoding delegating to a method

```
private String encodeString(String s) { ❶
    try {
        return URLEncoder.encode(s, "UTF-8");
    } catch (UnsupportedEncodingException e) {
        throw new RuntimeException(e);
    }
}

public List<String> encodeValuesUsingMethod(String... values) ·
    return Arrays.stream(values)
        .map(this::encodeString)   ❷
        .collect(Collectors.toList());
}
```

❶

Extracted method for exception handling

❷

Method reference to the extracted method

This works, and is simple to implement. It also gives you a method that you can test and/or debug separately. The only downside is that you need to extract a method for each operation that may throw an exception.

See Also

Using an extracted method to handle exceptions in lambdas is covered in "Using an Extracted Method for Exception Handling". Using a generic wrapper for exceptions is in "Using a Generic Exception Wrapper".

Using a Generic Exception Wrapper

Problem

You have a lambda expression that throws an exception, but you wish to use a generic wrapper to catches all checked exceptions and re-throws them as unchecked.

Solution

Create special exception classes and add a generic method to accept them and return lambdas without exceptions.

Discussion

Both “Using an Extracted Method for Exception Handling” and “Checked Exceptions And Lambdas” show how to delegate to a separate method to handle exceptions thrown from lambda expressions. Unfortunately, you need to define a private method for each operation that may throw an exception. This can be made more versatile using a *generic wrapper*.

For this approach, define a separate functional interface with a method that declares it throws `Exception`, and use a wrapper method to connect it to your code.

For example, the `map` method on `Stream` requires a `Function`, but the `apply` method in `Function` does not declare any checked exceptions. If you want to use a lambda expression in `map` that may throw a checked exception, start by creating a separate functional interface that declares that it throws `Exception`, as in Example 3-24.

Example 3-24. A functional interface based on Function that throws Exception

```
@FunctionalInterface
public interface FunctionWithException<T, R, E extends Exceptio
    R apply(T t) throws E;
}
```

Now you can add a wrapper method that takes a `FunctionWithException` and returns a `Function` by wrapping the `apply` method in a try/catch block, as shown in Example 3-25.

Example 3-25. A wrapper method to deal with exceptions

```
private static <T, R, E extends Exception>
    Function<T, R> wrapper(FunctionWithException<T, R, E> fe)
        return arg -> {
            try {
                return fe.apply(arg);
            } catch (Exception e) {
```

```
                throw new RuntimeException(e);
            }
        };
}
```

The `wrapper` method accepts code that throws any `Exception` and builds in the necessary try/catch block, while delegating to the `apply` method. In this case the `wrapper` method was made `static`, but that isn't required. The result is that you can invoke the wrapper with any `Function` that throws an exception, as in Example 3-26.

Example 3-26. Using a generic static wrapper method

```
public List<String> encodeValuesWithWrapper(String... values) {
    return Arrays.stream(values)
        .map(wrapper(s -> URLEncoder.encode(s, "UTF-8")))  ❶
        .collect(Collectors.toList());
}
```

❶

Using the `wrapper` method

Now you can write code in your `map` operation that throws any exception, and the `wrapper` method will re-throw it as unchecked. The downside to this approach is that you need a separate generic wrapper, like `ConsumerWithException`, `SupplierWithException`, and so on, for each functional interface you plan to use.

It's complications like this that make it clear why some Java frameworks (like Spring and Hibernate), and even entire languages (like Groovy and Kotlin), catch all checked exceptions and re-throw them as unchecked.

See Also

Lambda expression with checked exceptions are discussed in "Checked Exceptions And Lambdas". Extracting to a method is dicussed in "Using an Extracted Method for Exception Handling".

[1] The Unix operating system is based on this idea, with similar advantages.

[2] Isn't that just about the worst-named class in the entire Java API? All exceptions are thrown at runtime; otherwise they're compiler errors. Shouldn't that class have been called `UncheckedException` all along? To emphasize how silly the situation can get, Java 8 also adds a new class called `java.io.UncheckedIOException` just to avoid some of the issues discussed in this recipe.

[3] Interestingly enough, if the values and the divisor are changed to `Double` instead of `Integer`, you don't get an exception at all, even if the divisor is 0.0. Instead you get a result where all the elements are "Infinity". This, believe it or not, is the correct behavior according to the IEEE 754 specification for handling floating point values in a binary computer.

Chapter 4. Streams

Java 8 introduces a new streaming metaphor to support functional programming. A stream is a sequence of elements that does not save the elements and does not modify the original source. Functional programming in Java often involves generating a stream from some source of data, passing the elements through a series of intermediate operations (called a *pipeline*), and completing the process with a *terminal expression*.

Streams can only be used once. After a stream has passed through zero or more intermediate operations and reached a terminal operation, it is finished. To process the values again, you need to make a new stream.

Streams are also lazy. A stream will only process as much data as is necessary to reach the terminal condition. "Lazy Streams" shows this in action.

The recipes in this section demonstrate various typical stream operations.

Creating Streams

Problem

You want to create a stream.

Solution

Use the static factory methods in the `Stream` interface, or the `stream` methods on `Iterable` or `Arrays`.

Discussion

The new `java.util.stream.Stream` interface in Java 8 provides several static methods for creating streams. Specifically, you can use the static methods `Stream.of`, `Stream.iterate`, and `Stream.generate`.

The `Stream.of` method takes a variable argument list of elements:

```
static <T> Stream<T> of(T... values)
```

The implementation of the `of` method in the standard library actually delegates to the `stream` method in the `Arrays` class, shown in Example 4-1.

Example 4-1. Reference implementation of Stream.of

```
@SafeVarargs
public static<T> Stream<T> of(T... values) {
    return Arrays.stream(values);
}
```

Tip

The `@SafeVarargs` annotation is part of Java generics. It comes up when you have an array as an argument, because it is possible to assign a typed array to an `Object` array and then violate type safety with an added element. The `@SafeVarargs` annotation tells the compiler that the developer promises not to do that. See [Link to Come] for additional details.

As a trivial example, see Example 4-2.

Note

Since streams do not process any data until a terminal expression is reached, each of the examples in this section will add a terminal method like `collect` or `forEach` at the end.

Example 4-2. Creating a stream using Stream.of

```
String names = Stream.of("Gomez", "Morticia", "Wednesday", "Pug
    .collect(Collectors.joining(","));
System.out.println(names);
// prints Gomez,Morticia,Wednesday,Pugsley
```

The API also includes an overloaded `of` method that takes a single element `T t`. This method returns a singleton sequential stream containing a single element.

Speaking of the `Arrays.stream` method, Example 4-3 shows an example.

Example 4-3. Creating a stream using Arrays.stream

```
String[] munsters = { "Herman", "Lily", "Eddie", "Marilyn", "Gr
names = Arrays.stream(munsters)
    .collect(Collectors.joining(","));
System.out.println(names);
// prints Herman,Lily,Eddie,Marilyn,Grandpa
```

Since you have to create an array ahead of time, this approach is less convenient, but works well for variable argument lists. The API includes overloads of `Arrays.stream` for arrays of `int`, `long`, and `double`, as well as the generic type used here.

Another static factory method in the `Stream` interface is `iterate`. The signature of the `iterate` method is:

```
static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
```

According to the JavaDocs, this method "returns an *infinite* (emphasis added) sequential ordered `Stream` produced by iterative application of a function `f` to an initial element seed". Recall that a `UnaryOperator` is a function whose single input and output types are the same (discussed in "Implementing java.util.function.Function"). This is useful when you have a way to produce the next value of the stream from the current value, as in Example 4-4.

Example 4-4. Creating a stream using Stream.iterate

```
List<BigDecimal> nums =
    Stream.iterate(BigDecimal.ONE, n -> n.add(BigDecimal.ONE) )
        .limit(10)
        .collect(Collectors.toList());
System.out.println(nums);
// prints [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Stream.iterate(LocalDate.now(), ld -> ld.plusDays(1L))
    .limit(10)
    .forEach(System.out::println)
// prints 10 days starting from today
```

The first example counts from one using `BigDecimal` instances. The second uses the new `LocalDate` class in `java.time` and adds one day to it repeatedly. Since the resulting streams are both unbounded, the intermediate operation `limit` is needed.

The other factory method in the `Stream` class is `generate`, whose signature is:

```
static <T> Stream<T> generate(Supplier<T> s)
```

This method produces a sequential, unordered stream by repeatedly invoking the `Supplier`. A simple example of a `Supplier` in the standard library (a method that takes no arguments but produces a return value) is the `Math.random` method, which is used in Example 4-5.

Example 4-5. Creating a stream of random doubles

```
long count = Stream.generate(Math::random)
    .limit(10)
    .forEach(System.out::println)
```

If you already have a collection, you can take advantage of the `default` method `stream` that has been added to the `Collection` interface, as in Example 4-6.[1]

Example 4-6. Creating a stream from a collection

```
List<String> bradyBunch = Arrays.asList("Greg", "Marcia", "Pete
names = bradyBunch.stream()
    .collect(Collectors.joining(","));
```

```
System.out.println(names);
// prints Greg,Marcia,Peter,Jan,Bobby,Cindy
```

There are three child interfaces of `Stream` specifically for working with primitives: `IntStream`, `LongStream`, and `DoubleStream`. `IntStream` and `LongStream` each have two additional factory methods for creating streams, `range` and `rangeClosed`. Their method signatures from `IntStream` are (`LongStream` is similar):

```
static IntStream  range(int startInclusive, int endExclusive)
static IntStream  rangeClosed(int startInclusive, int endInclus
static LongStream range(long startInclusive, long endExclusive)
static LongStream rangeClosed(long startInclusive, long endIncl
```

The arguments show the difference between the two: `rangeClosed` includes the end value, and `range` doesn't. Each returns a sequential, ordered stream that starts and the first argument and increments by one after that. An example of each is shown in Example 4-7.

Example 4-7. The range and rangeClosed methods

```
List<Integer> ints = IntStream.range(10, 15)
    .boxed()   ❶
    .collect(Collectors.toList());
System.out.println(ints);
// prints [10, 11, 12, 13, 14]

List<Long> longs = LongStream.rangeClosed(10, 15)
    .boxed()   ❶
    .collect(Collectors.toList());
System.out.println(longs);
// prints [10, 11, 12, 13, 14, 15]
```

❶

Necessary for Collectors to convert primitives to `List<T>`

The only quirk in that example is the use of the `boxed` method to convert the `int` values to `Integer` instances, which is discussed further in "Boxed Streams".

To summarize, here are the methods to create streams:

- `Stream.of(T... values)` and `Stream.of(T t)`
- `Arrays.stream(T[] array)`, with overloads for `int[]`, `double[]`, and `long[]`
- `Stream.iterate(T seed, UnaryOperator<T> f)`
- `Stream.generate(Supplier<T> s)`
- `Collection.stream()`
- Using `range` and `rangeClosed`:
    - `IntStream.range(int startInclusive, int endExclusive)`
    - `IntStream.rangeClosed(int startInclusive, int endInclusive)`
    - `LongStream.range(long startInclusive, long endExclusive)`
    - `LongStream.rangeClosed(long startInclusive, long endInclusive)`

See Also

Streams are used throughout this book. The process of converting streams of primitives to wrapper instances is discussed in "Boxed Streams".

Boxed Streams

Problem

You want to create a collection from a primitive stream.

Solution

Use the `boxed` method on `Stream` to wrap the elements. Alternatively, map the values using the appropriate wrapper class, or use the three-argument form of the `collect` method.

Discussion

When dealing with streams of objects, you can convert from a stream to a collection using one of the static methods in the `Collectors` class. For example, given a stream of strings, you can create a `List<String>` using the code in Example 4-8.

Example 4-8. Converting a stream of strings to a list

```
List<String> strings = Stream.of("this", "is", "a", "list", "o
    .collect(Collectors.toList());
```

The same process doesn’t work on streams of primitives, however. The code in Example 4-9 does not compile.

Example 4-9. Converting a stream of int to a list of Integer (DOES NOT COMPILE)

```
IntStream.of(3, 1, 4, 1, 5, 9)
    .collect(Collectors.toList());  // does not compile
```

You have three alternatives available as work-arounds. First, use the `boxed` method on `Stream` to convert the `IntStream` to a `Stream<Integer>`, as shown in Example 4-10.

Example 4-10. Using the boxed method

```
List<Integer> ints = IntStream.of(3, 1, 4, 1, 5, 9)
    .boxed() ❶
    .collect(Collectors.toList());
```

❶

Converts `int` to `Integer`

One alternative is to use the `mapToObj` method to convert each element from a primitive to an instance of the wrapper class, as in [Link to Come].

Example 4-11. Using the mapToObj method

```
List<Integer> ints = IntStream.of(3, 1, 4, 1, 5, 9)
    .mapToObj(Integer::valueOf)
    .collect(Collectors.toList())
```

Just as `mapToInt`, `mapToLong`, and `mapToDouble` parses streams of objects into the associated primitives, the `mapToObj` method from `IntStream`, `LongStream`, and `DoubleStream` converts primitives to instances of the associated wrapper classes. The argument to `mapToObj` in this example uses the `Integer` constructor.

Warning

In JDK 9, the `Integer(int val)` constructor is deprecated for performance reasons. The recommendation is to use `Integer.valueOf(int)` instead.

Another alternative is to use the three-argument version of `collect`, whose signature is:

```
<R> R collect(Supplier<R> supplier,
              ObjIntConsumer<R> accumulator,
              BiConsumer<R,R> combiner)
```

Example 4-12 shows how to use this method.

Example 4-12. Using the three-argument version of collect

```
List<Integer> ints = IntStream.of(3, 1, 4, 1, 5, 9)
    .collect(ArrayList<Integer>::new, ArrayList::add, ArrayList
```

In this version of `collect`, the supplier is the constructor for `ArrayList<Integer>`, the accumulator is the `add` method, which represents how to add a single element to a list, and the combiner (which is only used during parallel operations) is `addAll`, which combines two lists into one. Using the three-argument version of `collect` is not very common, but understanding how it works is a useful skill.

Any of these approaches work, so the choice is just a matter of style.

Incidentally, if you want to convert to an array rather than a list, then the `toArray` method works just as well if not better. See Example 4-13.

Example 4-13. Convert an IntStream to an int array

```
int[] intArray = IntStream.of(3, 1, 4, 1, 5, 9).toArray();

// or

int[] intArray = IntStream.of(3, 1, 4, 1, 5, 9).toArray(int[]::
```

The first demo uses the default form of `toArray`, which returns `Object[]`. The second uses an `IntFunction<int[]>` as a generator, which creates an `int[]` of the proper size and populates it.

The fact that any of these approaches is necessary is yet another consequence of the original decision in Java to treat primitives differently from objects, complicated by the introduction of generics. Still, using `boxed` or `mapToObj` is easy enough once you know to look for them.

See Also

Collectors are discussed in Chapter 5. Constructor references are covered in “Constructor References”.

Reduction Operations Using Reduce

Problem

You want to produce a customized single value from stream operations.

Solution

Use the `reduce` method to accumulate calculations on each element.

Discussion

The functional paradigm in Java often uses a process known as map-filter-reduce. The mapping operation transforms a stream of one type of variable (like a `String`) into another (like an int). Then a filter is applied to produce a new stream with only the desired elements in it (e.g., strings with length below a certain threshold). Finally, you may wish to provide a terminal operation that generates a single value from the stream (like a sum or average of the lengths).

Built-in Reduction Operations

The primitive streams `IntStream`, `LongStream`, and `DoubleStream` have several reduction operations built into the API.

For example, Table 4-1 shows the reduction operations from the `IntStream` class.

Table 4-1. Reduction operations in the IntStream class

Method	Return Type
`average`	`OptionalDouble`
`count`	`long`
`max`	`OptionalInt`
`min`	`OptionalInt`
`sum`	`int`
`summaryStatistics`	`IntSummaryStatistics`

`collect(Supplier<R> supplier, ObjIntConsumer<R> accumulator, BiConsumer<R,R> combiner)`	`R`
`reduce`	`int`, `OptionalInt`

As a simple demonstration, consider generating a stream of random numbers and checking their statistics, as shown in Example 4-14.

Example 4-14. Summary statistics from doubles

```
DoubleSummaryStatistics stats = DoubleStream.generate(Math::ran
    .limit(1_000_000)
    .summaryStatistics();
System.out.println(stats);
```

Note

A cool feature introduced in Java 7 is that you can use an underscore inside numeric literals as a readable delimiter which is ignored by the compiler.

The `DoubleSummaryStatistics` class has methods to return the count, sum, average, max, and min values individually, but simply printing the result shows an output similar to that shown in Example 4-15.

Example 4-15. Output of summary statistics calculation

```
DoubleSummaryStatistics{count=10000, sum=4983.859228, min=0.00
    average=0.498386, max=0.999998}
```

Other reduction operations like `sum`, `count`, `max`, `min`, and `average` do what you would expect. The only interesting part is that some of them return optionals, because if there are no elements in the stream (perhaps after a filtering operation) the result is undefined or null.

The last two operations in the table, `collect` and `reduce`, bear further discussion. The `collect` method is used throughout this book to convert a stream into a collection, usually in combination with one of the static helper methods in the `Collectors` class, like `toList` or `toSet`. That version of `collect` does not exist on the primitive streams. The three-argument version shown here takes a collection to populate, a way to add a single element to that collection, and a way to add multiple elements to the collection. An example is shown in "Boxed Streams".

Basic `reduce` implementations

The behavior of the `reduce` method, however, is not necessarily intuitive until you've seen it in action.

There are two overloaded versions of the `reduce` method in `IntStream`.

```
OptionalInt reduce(IntBinaryOperator op)
int         reduce(int identity, IntBinaryOperator op)
```

The first takes an `IntBinaryOperator` and returns an `OptionalInt`. The second asks you to supply an `int` called `identity` along with the `IntBinaryOperator`.

Recall that a `java.util.function.BiFunction` takes two arguments and returns a single value, all three of which can be different types. If the both input types and return type are all the same, the function is a `BinaryOperator` (for example, `Math.max`). An `IntBinaryOperator` is a `BinaryOperator` where the both inputs and the output type are all ints.

One way, therefore, to sum a series of numbers[2] would be to use the `reduce` code shown in Example 4-16.

Example 4-16. Summing numbers using reduce

```
int sum = IntStream.rangeClosed(1, 10)
    .reduce((x, y) -> x + y).orElse(0);

// sum == 55
```

The `IntBinaryOperator` here is supplied by the lambda expression that takes two ints and returns their sum. Since it is conceivable that the stream could be empty if we had added a filter, the result is an `OptionalInt`. Chaining the `orElse` method indicates that if there are no elements in the stream, the return value should be zero.

In the lambda expression, you can think of the first argument of the binary operator as an accumulator, and the second argument as the value of each element in the stream. This is made clear if you print each one as it goes by, as shown in Example 4-17.

Example 4-17. Printing the values of x and y

```
int sum = IntStream.rangeClosed(1, 10)
    .reduce((x, y) -> {
        System.out.printf("x=%d, y=%d%n", x, y);
        return x + y;
    }).orElse(0);
```

The output is shown in Example 4-18.

Example 4-18. The output of printing each value as it passes

```
x=1, y=2
x=3, y=3
x=6, y=4
x=10, y=5
x=15, y=6
x=21, y=7
x=28, y=8
x=36, y=9
x=45, y=10

sum=55
```

As the output shows, the initial values of `x` and `y` are the first two values of the range. The value returned by the binary operator becomes the value of `x` (i.e., the accumulator) on the next iteration, while `y` takes on each value in the stream.

This is fine, but what if you wanted to process each number before summing them? Say, for example, you wanted to double all the numbers before summing them.[3] A naïve approach would be simply to try the code shown in Example 4-19.

Example 4-19. Doubling the values during the sum (NOTE: NOT CORRECT)

```
int doubleSum = IntStream.rangeClosed(1, 10)
    .reduce((x, y) -> x + 2 * y).orElse(0);

// doubleSum == 109 (oops! off by one!)
```

Since the sum of the integers from 1 to 10 is 55, the resulting sum should be 110, but this calculation produces 109. The reason is that in the lambda expression in the `reduce` method, the initial values of `x` and `y` are 1 and 2 (the first two values of the stream), so that first value of the stream doesn't get doubled.

That's why there's an overloaded version of `reduce` that takes an initial value for the accumulator. The resulting code is shown in Example 4-20.

Example 4-20. Doubling the values during the sum (WORKS)

```
int doubleSum = IntStream.rangeClosed(1, 10)
    .reduce(0, (x, y) -> x + 2 * y);

// doubleSum == 110 (yay!)
```

By providing the initial value of zero for the accumulator `x`, the value of `y` is assigned to each of the elements in the stream, doubling them all. The values of `x` and `y` during each iteration are shown in Example 4-21.

Example 4-21. The values of the lambda parameters during each iteration

```
Acc=0, n=1
Acc=2, n=2
Acc=6, n=3
Acc=12, n=4
Acc=20, n=5
Acc=30, n=6
```

```
Acc=42, n=7
Acc=56, n=8
Acc=72, n=9
Acc=90, n=10

sum=110
```

Note also that when you use the version of `reduce` with an initial value for the accumulator, the return type is `int` rather than `OptionalInt`.

Identity values and Binary Operators

The demonstrations used in this section referred to the first argument as an initial value for the accumulator, even though the method signature called it `identity`. The word `identity` means that you should supply a value to the binary operator that, when combined with any other value, returns the other value. For addition, the identity is zero. For multiplication, the identity is 1.

For the summing operation demonstrated here, the result is the same, but it's worth keeping in mind that the actual requirement for the first argument of `reduce` is the identity value for whatever operation you are planning to use in the binary operator. Internally this becomes the initial value of the accumulator.

The standard library provides many reduction methods, but if none of them directly apply to your problem, the two forms of the `reduce` method shown here can be very helpful.

Binary Operators in the Library

A few methods have been added to the standard library that make reduction operations particularly simple. For example, `Integer`, `Long`, and `Double` all have a `sum` method that does exactly what you would expect. The implementation of the `sum` method in `Integer` is:

```
public static int sum(int a, int b) {
    return a + b;
}
```

Why bother creating a method just to add two integers, as done here? The `sum` method is a `BinaryOperator` (more specifically, an `IntBinaryOperator`) and can therefore be used easily in a `reduce` operation, as in Example 4-22.

Example 4-22. Performing a reduce with a binary operator

```
int sum = IntStream.rangeClosed(1, 10)
                   .reduce(0, Integer::sum);
System.out.println(sum);
```

The result is the same as using the `sum` method on `IntStream`. The `Integer` class also has a `max` and a `min` method, both of which are also binary operators and can be used the same way, as in Example 4-23.

Example 4-23. Finding the max using reduce

```
Integer max = Stream.of(3, 1, 4, 1, 5, 9)
        .reduce(Integer.MIN_VALUE, Integer::max);
System.out.println("The max value is " + max);

// The max value is 9
```

Another interesting example is the `concat` method in `String`, which doesn't actually look like a `BinaryOperator`:

```
String concat(String str)
```

You can use this in a `reduce` method anyway, as shown in Example 4-24.

Example 4-24. Concatenating strings from a stream using reduce

```
String s = Stream.of("this", "is", "a", "list")
        .reduce("", String::concat);
System.out.println(s);
// thisisalist
```

The reason this works is that when you use a method reference via the class name (as in `String::concat`), the first value is the target of the `concat` method and the second value is the argument to `concat`. Since the result returns a `String`, the target, argument, and return type are all of the same type and once again you can treat this as a binary operator for the `reduce` method.

This technique can greatly reduce[4] the size of your code, so keep in mind when you’re browsing the API.

Using a Collector

While using `concat` this way works, it is inefficient because `String` concatenation creates and destroys objects. Better would be to use the `collect` method with a `Collector`.

One overload of the `collect` method on `Stream` takes a `Supplier` for the collection, a `BiConsumer` that adds a single element to the collection, and a `BiConsumer` that combines two collections. With strings, the natural accumulator would be a `StringBuilder`. The corresponding `collect` implementation would look like Example 4-25.

Example 4-25. Collecting strings using a StringBuilder

```
String s = Stream.of("this", "is", "a", "list")
        .collect(() -> new StringBuilder(),      ❶
                (sb, str) -> sb.append(str),     ❷
                (sb1, sb2) -> sb1.append(sb2))   ❸
        .toString();
```

❶

Result supplier

❷

Add a single value to the result

❸

Combine two results

This approach can be more simply expressed using method references, as in <sb_method_refs>>.

Example 4-26. Collecting strings, with method references

```
String s = Stream.of("this", "is", "a", "list")
        .collect(StringBuilder::new,
                StringBuilder::append,
                StringBuilder::append)
        .toString();
```

Simplest of all, however, would be to use one the `joining` method in the `Collectors` utility class, as in Example 4-27.

Example 4-27. Joining strings using Collectors

```
String s = Stream.of("this", "is", "a", "list")
        .collect(Collectors.joining());
```

The `joining` method is overloaded to also take a string delimiter. It’s hard to beat that for simplicity.

For more details and examples, see “Converting a Stream into a Collection”.

The most general form of `reduce`

The third form of the `reduce` method is:

```
<U> U reduce(U identity,
             BiFunction<U,? super T,U> accumulator,
             BinaryOperator<U> combiner)
```

This is a bit more complicated, and there are normally easier ways to accomplish the same goal, but an example of how to use it might be useful. Consider a `Book` class with simply an integer id and a string title, as in Example 4-28.

Example 4-28. A simple Book class

```
public class Book {
    private Integer id;
    private String title;

    // constructors, getters and setters, toString, equals, has
}
```

Say you have a list of books and you want to add them to a map, where the keys are the ids and the values are the books themselves. One way to accomplish that is shown in Example 4-29.

Example 4-29. Accumulating Books into a Map

```
HashMap<Integer, Book> bookMap = books.stream()
    .reduce(new HashMap<Integer, Book>(),      ❶
           (map, book) -> {                    ❷
                map.put(book.getId(), book);
                return map;
            },
            (map1, map2) -> {                  ❸
                map1.putAll(map2);
                return map1;
            });

bookMap.forEach((k,v) -> System.out.println(k + ": " + v));
// prints
// 1: Book{id=1, title='Modern Java Recipes'}
// 2: Book{id=2, title='Making Java Groovy'}
// 3: Book{id=3, title='Gradle Recipes for Android'}
```

❶

Identity value for `putAll`

❷

Accumulate a single book into map using `put`

❸

Combine multiple maps using `putAll`

It’s easiest to examine the arguments to the `reduce` method in reverse order.

The last argument is a `combiner`, which is required to be a `BinaryOperator`, meaning the two input arguments and the return value are all of the same type. In this case, the provided lambda expression takes two maps and copies all the keys from the second map into the first one and returns it. The lambda expression would be simpler if the `putAll` method returned the map, but no such luck. The combiner is only relevant if the `reduce` operation is done in parallel, because then you need to combine maps produced from each portion of the range.

The second argument is a function that adds a single book to a map. This too would be simpler if the `put` method on `Map` returned the `Map` after the new entry was added.

Finally, the first argument to the `reduce` method is the identity value for the `combiner` function, meaning that if `combiner` is invoked with the identity and an argument, it should return the argument. In this case, the identity value is an empty map, because that combined with any other map returns the other map.

Reduction operations are fundamental to the functional programming idiom. In many common cases, the `Stream` interfaces provide a built-in method for you, like `sum` or `collect(Collectors.joining(','))`. If you need to write your own, however, this recipe shows how to use the `reduce` operation directly.

The best news is that once you understand how to use `reduce` in Java 8, you know how to use the same operation in other languages, even if it goes by different names (like `inject` in Groovy or `fold` in Scala). They all work the same way.

See Also

A much simpler way to turn a list of POJOs into a map is shown in “Adding a Linear Collection to a Map”. Summary statistics are discussed in [Link to Come]. Sorting operations are covered in [Link to Come].

Check Sorting Using Reduce

Problem

You want to check that a sort is correct.

Solution

Use the `reduce` method to check each pair of elements.

Discussion

The `reduce` method on `Stream` takes a `BinaryOperator` as an argument:

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

A `BinaryOperator` is a `Function` where the types of both input arguments and the output value are all the same. As shown in the previous recipe, the first element in the `BinaryOperator` is normally an accumulator, while the second element takes each value of the stream, as in Example 4-30:

Example 4-30. Summing BigDecimals with Reduce

```
BigDecimal total = Stream.iterate(BigDecimal.ONE, n -> n.add(B
        .limit(10)
        .reduce(BigDecimal.ZERO, (acc, val) -> acc.add(val)); ❶
System.out.println("The total is " + total);
```

❶

```
BinaryOperator<BigDecimal>
```

As usual, whatever is returned by the lambda expression becomes the next value of the `acc` variable on the next iteration. In this way, the calculation accumulates the values of the first 10 `BigDecimal` instances.

This is the most typical way of using the `reduce` method, but just because `acc` here is used as an accumulator doesn't mean it has to be handled that way. Consider sorting strings instead, using the approach discussed in "Sorting Using A Comparator". The code snippet shown in Example 4-31 sorts strings by length.

Example 4-31. Sorting strings by length

```
List<String> strings = Arrays.asList("this", "is", "a", "list",

List<String> sorted = strings.stream()
    .sorted(Comparator.comparingInt(String::length))
```

```
    .collect(toList());

// result is ["a", "is", "of", "this", "list", "strings"]
```

The question is, how do you test this? Each adjacent pair of strings has to be compared by length to make sure the first is equal to or shorter than the second. The `reduce` method here works well, however, as Example 4-32 shows (part of a JUnit test case).

Example 4-32. Testing that strings are sorted properly

```
strings.stream()
    .reduce((prev, curr) -> {
        assertTrue(prev.length() <= curr.length());   ❶
        return curr;                                  ❷
    });
```

❶

Check each pair is sorted properly

❷

`curr` becomes the next value of `prev`

For each consecutive pair, the previous and current parameters are assigned to variables `prev` and `curr`. The assertion tests that the previous length is less than or equal to the current length. The important part is that the argument to `reduce`, here a `BinaryOperator`, returns the value of the current string, which becomes the value of `prev` on the next iteration.

The only thing required to make this work is for the stream to be sequential and ordered, as done here.

See Also

Debugging Streams with peek

Problem

You want to see the individual elements of a stream as they are processed.

Solution

Invoke the `peek` intermediate operation wherever you need it in a stream pipeline.

Discussion

Newcomers to Java 8 sometimes find the sequence of intermediate operations on a stream confusing, because they have trouble visualizing the stream values as they are processed.

Consider a simple method that accepts a start and end range for a stream of integers, doubles each number, and then sums up only the resulting values divisible by 3, as shown in Example 4-33.

Example 4-33. Doubling integers, filtering, and summing

```
public int sumDoublesDivisibleBy3(int start, int end) {
    return IntStream.rangeClosed(start, end)
        .map(n -> n * 2)
        .filter(n -> n % 3 == 0)
        .sum();
}
```

A simple test could prove that this is working properly.

```
@Test
public void sumDoublesDivisibleBy3() throws Exception {
    assertEquals(1554, demo.sumDoublesDivisibleBy3(100, 120));
}
```

That’s helpful, but doesn’t deliver a lot of insight. If the code wasn’t working, it would be very difficult to figure out where the problem lay.

Imagine that you added a `map` operation to the pipeline that took each value, printed it, and then returned the value again, as in Example 4-34.

Example 4-34. Adding an identity map for printing

```
public int sumDoublesDivisibleBy3(int start, int end) {
    return IntStream.rangeClosed(start, end)
        .map(n -> { ❶
            System.out.println(n);
```

```
            return n;
        })
        .map(n -> n * 2)
        .filter(n -> n % 3 == 0)
        .sum();
}
```

❶

Identity map that prints stream elements

The result prints the numbers from `start` to `end`, inclusive, with one number per line. While you might not want this in production code, it gives you a look inside the stream processing without interfering with it.

This behavior is exactly how the `peek` method in `Stream` works. The declaration of the `peek` method is:

```
Stream<T> peek(Consumer<? super T> action)
```

According to the JavaDocs, the `peek` method “returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as they are consumed from the resulting stream.” Recall that a `Consumer` takes a single input but returns nothing, so any provided `Consumer` will not corrupt each value as it streams by.

Since `peek` is an intermediate operation, the `peek` method can be added multiple times if you wish, as in Example 4-35.

Example 4-35. Using multiple peek methods

```
public int sumDoublesDivisibleBy3(int start, int end) {
    return IntStream.rangeClosed(start, end)
        .peek(n -> System.out.printf("original: %d%n", n)) ❶
        .map(n -> n * 2)
        .peek(n -> System.out.printf("doubled : %d%n", n)) ❷
        .filter(n -> n % 3 == 0)
        .peek(n -> System.out.printf("filtered: %d%n", n)) ❸
        .sum();
}
```

❶

print value before doubling

❷

print value before filtering

❸

print value before summing

The result will show each element in its original form, then after it has been doubled, and finally only if it passes the filter.

Unfortunately, there's no easy way to make the `peek` code optional, so this is a convenient step to use for debugging but should be removed for production code.

See Also

Converting Strings to Streams and Back

Problem

Rather than loops over individual characters of a `String`, you would like to use the idiomatic `Stream` processing techniques.

Solution

Use the default methods `chars` and `codePoints` from the `java.lang.CharSequence` interface to convert a `String` into an `IntStream`. To convert back to a `String`, use the overload of the `collect` method on `IntStream` that takes a `Supplier`, a `BiConsumer` representing an accumulator, and a `BiConsumer` representing a combiner.

Discussion

Strings are collections of characters, so in principle it should be as easy to convert a string into a stream as it is any other collection or array. Unfortunately, `String` is not part of the collections framework, and therefore does not implement `Iterator`, so there is no `stream` factory method to convert one into a `Stream`. The other option would be the static `stream` methods in the `java.util.Arrays` class, but while there are versions of `Arrays.stream` for `int[]`, `long[]`, `double[]`, and even `T[]`, there isn't one for `char[]`. It's almost as if the designers of the API didn't want you to process a `String` using stream techniques.

Still, there is an approach that works. The `String` class implements the `CharSequence` interface, and that interface contains two new methods that produce an `IntStream`. Both methods are default methods in the interface, so they have an implementation. The signatures are in Example 4-36.

Example 4-36. Stream methods in java.lang.CharSequence

```
default IntStream chars()
default IntStream codePoints()
```

The difference between the two methods has to do with how Java handles UTF-16 encoded characters as opposed to the full Unicode set of code points. If you're interested, the differences are explained in the JavaDocs for `java.lang.Character`. For the methods shown here, the difference is only in the type of integers returned. The former returns a `IntStream` consisting of `char` values from this sequence, while the latter returns an `IntStream` of Unicode code points.

The opposite question is how to convert a stream of characters back into a `String`. The `Stream.collect` method is used to perform a mutable reduction on the elements of a stream to produce a collection. The version of `collect` that takes a `Collector` is most commonly used, because the `Collectors` utility class provides many static methods (like `toList`, `toSet`, `toMap`, `joining`, and many others discussed in this book) that produce the desired

`Collector`.

Conspicuous by its absence, however, is a `Collector` that will take a stream of characters and assemble it into a `String`. Fortunately, that code isn't difficult to write, using the other overload of `collect`, which takes a `Supplier` and two `BiConsumer` arguments, one as an accumulator and one as a combiner.

This all sounds a lot more complicated than it is in practice. Consider writing a method to check if a string is a palindrome. Palindromes are not case sensitive, and they remove all punctuation before checking whether the resulting string is the same forward as backwards. In Java 7 or earlier, Example 4-37 shows a simple way to write a method that tests them.

Example 4-37. Checking for palindromes in Java 7 or earlier

```
public boolean isPalindrome(String s) {
    StringBuilder sb = new StringBuilder();
    for (char c : s.toCharArray()) {
        if (Character.isLetterOrDigit(c)) {
            sb.append(c);
        }
    }
    String forward = sb.toString().toLowerCase();
    String backward = sb.reverse().toString().toLowerCase();
    return forward.equals(backward);
}
```

As is typical in code written in a non-functional style, the method declares a separate object with mutable state (the `StringBuilder` instance), then iterates over a collection (the `char[]` returned by the `toCharArray` method in `String`), using an `if` condition to decide whether to append a value to the buffer. The `StringBuilder` class also has a `reverse` method to make checking for palindromes easier, while the `String` class does not. This combination of mutable state, iteration, and decision statements cries out for an alternative stream-based approach.

That stream-based alternative is shown in Example 4-38.

Example 4-38. Checking for palindromes using Java 8 streams

```
public boolean isPalindrome(String s) {
    String forward = s.toLowerCase().codePoints()
        .filter(Character::isLetterOrDigit)
        .collect(StringBuilder::new,
                 StringBuilder::appendCodePoint,
                 StringBuilder::append)
        .toString();

  String backward = new StringBuilder(forward).reverse().toStr:
  return forward.equals(backward);
}
```

The `codePoints` method returns an `IntStream`, which can then be filtered using the same condition as above. The interesting part is in the `collect` method, whose signature is:

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R,? super T> accumulator,
              BiConsumer<R,R> combiner)
```

The arguments are:

- a `Supplier` which produces the resulting reduced object, in this case a `StringBuilder`
- a `BiConsumer` used to accumulate each element of the stream into the resulting data structure; this example uses the `appendCodePoint` method
- a `BiConsumer` representing a combiner, which is a "non-interfering, stateless function" for combining two values which must be compatible with the accumulator; in this case, the `append` method. Note that the combiner is only used if the operation is done in parallel.

That sounds like a lot, but the advantage in this case is that the code doesn't have to make a distinction between characters and integers, which is often an issue when working with elements of strings.

Example 4-39 shows a simple test of the method.

Example 4-39. Testing the palindrome checker

```
import org.junit.Test;
import java.util.stream.Stream;

import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;

public class PalindromeEvaluatorTest {
    private PalindromeEvaluator demo = new PalindromeEvaluator

    @Test
    public void isPalindrome() throws Exception {
        assertTrue(
            Stream.of("Madam, in Eden, I'm Adam",
                "Go hang a salami; I'm a lasagna hog",
                "Flee to me, remote elf!",
                "A Santa pets rats as Pat taps a star step at N
                .allMatch(demo::isPalindrome));

        assertFalse(demo.isPalindrome("This is NOT a palindrome
    }
}
```

Viewing strings as arrays of characters doesn't quite fit the new functional idioms in Java 8, but the mechanisms in this recipe hopefully show how they can be make to work.

See Also

Collectors are discussed further in Chapter 5.

Counting Elements

Problem

You want to know how many elements are in a stream.

Solution

Use either the `Stream.count` or `Collectors.counting` methods.

Discussion

This recipe is almost too easy, but does serve to demonstrate a technique that will be revisited later in the section on downstream collectors.

The `Stream` interface has a default method called `count` that returns a `long`, which is demonstrated in Example 4-40.

Example 4-40. Counting elements in a stream

```
long count = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5).count();
System.out.printf("There are %d elements in the stream%n", coun
//
// There are 9 elements in the stream
```

One interesting feature of the `count` method is that the JavaDocs show how it is implemented. The docs say, “this is a special case of a reduction and is equivalent to:”

```
return mapToLong(e -> 1L).sum();
```

First every element in the stream is mapped to `1` as a `long`. Then the `mapToLong` method produces a `LongStream`, which has a `sum` method. In other words, map all the elements to ones and add them up. Nice and simple.

An alternative is to notice that the `Collectors` class has a similar method, called `counting`, shown in Example 4-41.

Example 4-41. Counting the elements using Collectors.counting

```
count = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
    .collect(Collectors.counting());
System.out.printf("There are %d elements in the stream%n", coun
```

The result is the same. The question is, why do this? Why not use the `count` method on `Stream` instead?

You can, of course, and arguably should. Where this becomes useful, however, is as a *downstream collector*, discussed more extensively in "Downstream Collectors". As a spoiler, consider the following example, Example 5-23, repeated from that section.

Example 4-42. Counting string partitioned by length

```
Map<Boolean, Long> numberLengthMap = strings.stream()
    .collect(Collectors.partitioningBy(
        s -> s.length() % 2 == 0,  ❶
        Collectors.counting()));   ❷

numberLengthMap.forEach((k,v) -> System.out.printf("%5s: %d%n",
//
// false: 4
//  true: 8
```

❶

predicate

❷

downstream collector

The first argument to `partitioningBy` is a `Predicate`, used to separate the strings into two categories: those that satisfy the predicate, and those that don't. If that was the only argument to `partitioningBy`, the result would be a `Map<Boolean, List<String>>`, where the keys would be the values `true` and `false`, and the values would be a list of even-length strings and a list of odd-length strings.

The two-argument overload of `partitioningBy` used here takes a `Predicate` followed by a `Collector`, called a downstream collector, which post-processes each list of strings returned. This is the use case for the `Collectors.counting` method. The output now is a `Map<Boolean, Long>` where the values are the number of even- and odd-length strings in the stream.

Several other methods in `Stream` have analogs in `Collectors` methods, which are discussed in that section. In each case, if you are working directly with a stream, use the `Stream` methods. The `Collectors` methods are intended for downstream post-processing of a `partitioningBy` or `groupingBy` operation.

See Also

Downstream collectors are discussed in “Downstream Collectors”. Collectors in general are discussed in several recipes in the section on Collectors, including “Partitioning and Grouping”.

Summary Statistics

Problem

You want the count, sum, min, max, and average of a stream of numerical values.

Solution

Use the `summaryStatistics` method in `IntStream`, `DoubleStream`, and `LongStream`.

Discussion

The primitive streams `IntStream`, `DoubleStream`, and `LongStream` add methods to the `Stream` interface that work for primitive types. One of those methods is `summaryStatistics`, shown in Example 4-43.

Example 4-43. SummaryStatistics

```
DoubleSummaryStatistics stats = DoubleStream.generate(Math::rar
    .limit(1_000_000)
    .summaryStatistics();

System.out.println(stats);  ❶

System.out.println("count: " + stats.getCount());
System.out.println("min  : " + stats.getMin());
System.out.println("max  : " + stats.getMax());
System.out.println("sum  : " + stats.getSum());
System.out.println("ave  : " + stats.getAverage());
```

Tip

Java 7 added the capability to use underscores in numerical literals, as in 1_000_000.

A typical run yields:

```
DoubleSummaryStatistics{count=1000000, sum=499608.317465, min=(
count: 1000000
min  : 1.3938598313334438E-6
max  : 0.9999988915490642
sum  : 499608.31746475823
ave  : 0.49960831746475826
```

The `toString` implementation of `DoubleSummaryStatistics` shows all the values, but the class also has getter methods for the individual quantities. With one million doubles, it's not surprising that the minimum is close to zero, the maximum is close to 1, the sum is approximately 500,000, and the average is nearly 0.5.

This is essentially a “poor developer’s” approach to statistics. It’s limited, but if all you need is the available set of basic quantities, it’s nice to know the library provides them automatically.

See Also

Summary statistics is a special form of a reduction operation. Others appear in "Reduction Operations Using Reduce".

Finding The First Element In A Stream

Problem

You wish to find the first element in a stream that satisfies a particular condition.

Solution

Use the `findFirst` or `findAny` method after applying a filter.

Discussion

The `findFirst` and `findAny` methods in `java.util.stream.Stream` return an `Optional` describing the first element of a stream. Neither takes an argument, implying that any mapping or filtering operations have already been done.

For example, given a list of integers, to find the first even number, apply an even-number filter and then use `findFirst`, as in Example 4-44.

Example 4-44. Finding the first even integer

```
Optional<Integer> firstEven = Stream.of(3, 1, 4, 1, 5, 9, 2, 6,
    .filter(n -> n % 2 == 0)
    .findFirst();

System.out.println(firstEven);
// prints: Optional[4]
```

If the stream is empty, the return value is an empty `Optional` (see Example 4-45).

Example 4-45. Using findFirst on an empty stream

```
Optional<Integer> firstEvenGT10 = Stream.of(3, 1, 4, 1, 5, 9, 2
    .filter(n -> n > 10)
    .filter(n -> n % 2 == 0)
    .findFirst();

System.out.println(firstEvenGT10);
// prints: Optional.empty
```

Since the code returns the first element after applying the filter, you might think that it involves a lot of wasted work. Why apply a modulus operation to all the elements and then pick just the first one? Stream elements are actually processed one by one, so this isn't a problem. This is discussed in "Lazy Streams".

If the stream has no encounter order, then any element may be returned. In the current example, the stream does have an encounter order, so the “first” even number (in the original example) is always 4, *whether we do the search using a sequential or a parallel stream*. See Example 4-46.

Example 4-46. Using firstEven in parallel

```
firstEven = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
     .parallel()
     .filter(n -> n % 2 == 0)
     .findFirst();

System.out.println(firstEven);
// Always prints Optional[4]
```

That feels bizarre at first. Why would you get the same value back even though several numbers are being processed at the same time. The answer lies in the notion of *encounter order*.

The API defines encounter order as the order in which the source itself makes its elements available. A `List` and an array have an encounter order, but a `Set` does not.

There is also a method called `unordered` in `BaseStream` (which `Stream` extends) that (optionally!) returns an unordered stream as an intermediate operation, though it may not.

Sets and Encounter Order

`HashSet` instances have no defined encounter order, but if you initialize one with the same data repeatedly you will get the same order of elements each time. That means using `findFirst` will give the same result each time as well. The method documentation says that `findFirst` *may* give a different result on unordered streams, but the current implementation doesn’t change its behavior just because the stream is unordered.

To get a `Set` with a different encounter order, you can add and remove enough elements to force a rehash. For example:

```
List<String> wordList = Arrays.asList(
    "this", "is", "a", "stream", "of", "strings");
Set<String> words = new HashSet<>(wordList);
Set<String> words2 = new HashSet<>(words);

// Now add and remove enough elements to force a rehash
IntStream.rangeClosed(0, 50).forEachOrdered(i -> words2.add(St
words2.retainAll(wordList);

// The sets are equal, but have different element ordering
System.out.println(words.equals(words2));
System.out.println("Before: " + words);
System.out.println("After : " + words2);
```

The outputs will be something like:

```
true
Before: [a, strings, stream, of, this, is]
After : [this, is, strings, stream, of, a]
```

The ordering is different, so the result of `findFirst` will be different.

In Java 9, the new immutable sets (and maps) are randomized, so their iteration orders will change from run to run, even if they are initialized the same way every time.[5]

The `findAny` method returns an `Optional` describing some element of the stream, or an empty `Optional` if the stream is empty. In this case, the behavior of the operation is *explicitly nondeterministic*, meaning it is free to select any element of the stream. This allows optimization in parallel operations.

To demonstrate this, consider returning any element from an unordered, parallel stream of integers. Example 4-47 introduces an artificial delay by mapping each element to itself after a random delay of up to 100 milliseconds.

Example 4-47. Using findAny in parallel after a random delay

```
public Integer delay(Integer n) {
    try {
        Thread.sleep((long) (Math.random() * 100));
    } catch (InterruptedException ignored) { ❶
    }
```

```
    return n;
}

// ...

Optional<Integer> any = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
    .unordered() ❷
    .parallel()  ❸
    .map(this::delay) ❹
    .findAny();  ❺

System.out.println("Any: " + any);
```

❶

The only exception in Java that it is okay to catch and ignore

❷

We don’t care about order

❸

Use the common fork-join pool in parallel

❹

Introduce a random delay

❺

Return the first element, regardless of encounter order

The output now could be any of the given numbers, depending on which thread gets there first.

Both `findFirst` and `findAny` are *short-circuiting*, *terminal* operations. A short-circuiting operation may produce a finite stream when presented with an infinite one. A terminal operation is short-circuiting if it may terminate in finite time even when presented with infinite input.

Note that the examples used in this section demonstrate that sometimes parallelization can hurt rather than help performance. Streams are lazy, meaning they will only process as many elements as are necessary to satisfy the pipeline. In this case, since the requirement is simply to return the first element, firing up a fork-join pool is overkill. See Example 4-48.

Example 4-48. Using findAny on sequential and parallel streams

```
Optional<Integer> any = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
    .unordered()
    .map(this::delay)
    .findAny(); ❶

System.out.println("Sequential Any: " + any);

any = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
    .unordered()
    .parallel()
    .map(this::delay)
    .findAny(); ❷

System.out.println("Parallel Any: " + any);
```

❶

Sequential stream (by default)

❷

Parallel stream

(This demo assumes that the delay method has been modified to print the name of the current thread along with the value it is processing.)

Typical output looks like the following (on an eight-core machine, which therefore uses a fork-join pool with eight threads by default).

```
main // sequential, so only one thread
Sequential Any: Optional[3]

ForkJoinPool.commonPool-worker-1
ForkJoinPool.commonPool-worker-5
ForkJoinPool.commonPool-worker-3
ForkJoinPool.commonPool-worker-6
ForkJoinPool.commonPool-worker-7
main
ForkJoinPool.commonPool-worker-2
```

```
ForkJoinPool.commonPool-worker-4
Parallel Any: Optional[1]
```

The sequential stream only needs to access one element, which it then returns, short circuiting the process. The parallel stream fires up eight different threads, finds one element, and shuts them all down. The parallel stream therefore accesses many values it doesn’t need.

Again, the key concept is that of encounter order with streams. If the stream has an encounter order, then `findFirst` will always return the same value. The `findAny` method is allowed to return any element, making it more appropriate for parallel operations.

See Also

Lazy streams are discussed in “Lazy Streams”. Parallel streams are in [Link to Come].

Using anyMatch, allMatch, and noneMatch

Problem

You wish to determine if any elements in a stream match a `Predicate`, or if all match, or if none match.

Solution

Use the methods `anyMatch`, `allMatch`, and `noneMatch` on the `Stream` interface, each of which return a boolean.

Discussion

The signatures of the `anyMatch`, `allMatch`, and `noneMatch` methods on `Stream` are:

```
boolean anyMatch(Predicate<? super T> predicate)
boolean allMatch(Predicate<? super T> predicate)
boolean noneMatch(Predicate<? super T> predicate)
```

Each does exactly what it sounds like. As an example, consider a prime number calculator. A number is prime if none of the integers from 2 up to the value (minus 1) evenly divide into it.

A trivial way to check if a number is prime is to compute the modulus of the number from every number from two up to its square root, rounded up, as in Example 4-49.

Example 4-49. Prime number check

```
public boolean isPrime(int num) {
    int limit = (int) (Math.sqrt(num) + 1);          ❶
    return num == 2 || num > 1 && IntStream.range(2, limit)
        .noneMatch(divisor -> num % divisor == 0); ❷
}
```

❶

Upper limit for check

❷

Using `noneMatch`

The `noneMatch` method makes the calculation particularly simple.

BigInteger and Primes

Interestingly enough, the `java.math.BigInteger` class has a method called `isProbablyPrime` with the following signature:

```
boolean isProbablyPrime(int certainty)
```

If the method returns `false`, the value is definitely composite. For `true`, however, the `certainty` argument comes into play.

The value of `certainty` represents the amount of uncertainty that the caller is willing to tolerate. If the method returns `true`, the probability that the number is actually prime exceeds `1 - 1/2^{certainty}`, so a `certainty` of 2 implies a probability of 0.5, a `certainty` of 3 implies 0.75, 4 implies 0.875, 5 implies 0.9375, and so on.

Asking for greater values of `certainty` makes the algorithm take longer.

Two ways to test the calculation are shown in Example 4-50.

Example 4-50. Tests for the prime calculation

```
private Primes calculator = new Primes();

@Test ❶
public void testIsPrimeUsingAllMatch() throws Exception {
    assertTrue(IntStream.of(2, 3, 5, 7, 11, 13, 17, 19)
        .allMatch(calculator::isPrime));
}

@Test ❷
public void testIsPrimeWithComposites() throws Exception {
    assertFalse(Stream.of(4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 2
        .anyMatch(calculator::isPrime));
}
```

❶

Use `allMatch` for simplicity

❷

Test with composites

The first test invokes the `allMatch` method, whose argument is a `Predicate`, on a stream of known primes and returns `true` only if all the values are prime.

The second test uses `anyMatch` with a collection of composite (non-prime) numbers, and asserts that none of them satisfy the predicate.

The `anyMatch`, `allMatch`, and `noneMatch` methods are convenient ways to check a stream of values against a particular condition.

See Also

Stream flatMap vs map

Problem

You have a stream and you need to transform the elements in some way, but you’re not sure whether to use `map` or `flatMap`.

Solution

Use `map` if you each element is transformed into a single value. Use `flatMap` if each element will be transformed to multiple values and the resulting stream needs to be “flattened”.

Discussion

Both the `map` and the `flatMap` methods on `Stream` take a `Function` as an argument. The signature for `map` is:

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
```

A `Function` takes a single input and transforms it into a single output. In the case of `map`, a single input of type `T` is transformed into a single output of type `R`. Examples of this have been used throughout this book. Example 4-51 shows a stream of strings being mapped to a stream of integers, and a stream of `Person` instances being mapped to a stream of strings representing their names.

Example 4-51. Map Examples

```
List<Integer> sizes = Stream.of("This", "is", "a", "stream", "o
    .map(String::length)
    .collect(Collectors.toList());

System.out.println(sizes);
// prints: [4, 2, 1, 6, 2, 7]

List<String> names = Stream.of(new Person("Steve"), new Person
    new Person("Thor"), new Person("Natasha"),
    new Person("Bruce"), new Person("Clint"))
    .map(Person::getName)
    .collect(Collectors.toList());

System.out.println(names);
// prints: [Steve, Tony, Thor, Natasha, Bruce, Clint]
```

Nothing surprising there. The `flatMap` method, on the other hand, takes a function that can produce multiple output values for each input value. Since Java methods have only a single return type, the `Function` argument for `flatMap` produces a `Stream` of the output values.

The signature of the `flatMap` method is:

```
<R> Stream<R> flatMap(Function<? super T,? extends Stream<? ext
```

For each generic argument `T`, the function produces a `Stream<R>` rather than just an `R`. The `flatMap` method then “flattens” the resulting stream by removing each element from the individual streams and adding them to the output.

To demonstrate `flatMap`, consider the idea of counting how many times each word appears in a passage of text[6]. Given a file of text, this means the code needs to read each line of text, and from each line, split it into words.

Consider a file like that shown in Example 4-52.

Example 4-52. A simple file with lines of text

```
This is a
very simple file
with some text

in it and this has

some simple duplicates
with the text
```

The new static method `Files.lines(Path)` takes a `Path` as an argument and returns a `Stream<String>` containing each line of the file. Then, to split each line into words, you can use the `split` method in `String`, which takes a regular expression. Note that this particular file has empty lines in it, which we don’t want in the resulting map of word counts.

The `flatMap` method takes each element of a given stream and applies a function that can produce a stream of results. Since you don’t want the empty words to count, the process will be done in a ternary operator.

```
line -> line.length() == 0 ? Stream.empty() : Stream.of(line.sp
```

Example 4-53 puts everything together. Note the use of a downstream collector to create the map of words to the sizes of the resulting lists.

Example 4-53. Using both map and flatMap to create a map of word counts

```
import java.io.IOException;
import java.nio.file.Files;
```

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Map;
import java.util.function.Function;
import java.util.stream.Stream;

import static java.util.stream.Collectors.counting;
import static java.util.stream.Collectors.groupingBy;

public class WordMap {
    private Path resourceDir = Paths.get("src/main/resources");
    private String fileName = "simple_file.txt";

    public Map<String, Long> createMap() {
        try (Stream<String> lines =
            Files.lines(resourceDir.resolve(fileName))) { ❶
            return lines.flatMap(line ->  ❷
                line.length() == 0 ? Stream.empty() :
                    Stream.of(line.split("\\W+")))
                .map(String::toLowerCase)  ❸
                .collect(groupingBy(Function.identity(), count:
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

❶

Produce a stream of lines

❷

For each line, either produce a stream of words or an empty stream

❸

Simple map method to convert to lowercase

❹

Downstream collector to return just the sizes of the lists of words

The idea is to convert each line into a stream of words, and each empty line into an empty stream. The `flatMap` method then takes the resulting “stream of streams” and reduces it to a single stream of words, removing the empty streams in the process.

For this file, the resulting map is:

```
duplicates: 1
some: 2
very: 1
a: 1
in: 1
this: 2
simple: 2
is: 1
it: 1
the: 1
with: 2
file: 1
and: 1
has: 1
text: 2
```

The two key concepts for `flatMap` are:

1. The `Function` argument to `flatMap` produces a stream of output values, and

2. The resulting stream of streams is then flattened into a single stream of results.

If you keep those ideas in mind, you should find the `flatMap` method quite helpful.

As a final note, the `Optional` class also has a `map` method and a `flatMap` method. See “Mapping Optionals” and “Optional flatMap vs map” for details.

See Also

The `flatMap` method is also demonstrated in “Mapping Optionals”. `flatMap` in `Optional` is discussed in “Optional flatMap vs map”. Downstream collectors are discussed in “Downstream Collectors”. The `Files.lines` method is also used in [Link to Come].

Concatenating Streams

Problem

You want to combine two or more streams into a single one.

Solution

The `concat` method on `Stream` combines two streams, which works if the number of streams is small. Otherwise use `flatMap`.

Discussion

Say you acquire data from several locations, and you want to process every element in all of them using streams. One mechanism you can use is the `concat` method in `Stream`, whose signature is:

```
static <T> Stream<T> concat(Stream<? extends T> a, Stream<? ext
```

This method creates a lazily concatenated stream which accesses all the elements of the first stream, followed by all the elements of the second stream. As the Javadocs say, the resulting stream is ordered if the input streams are ordered, and the resulting stream is parallel if *either* of the input streams are parallel. Closing the returned stream also closes the underlying input streams.

NOTE

Both input streams must hold elements of the same type.

As a simple example of concatenating streams, see Example 4-54.

Example 4-54. Concatenating two

```
@Test
public void concat() throws Exception {
    Stream<String> first = Stream.of("a", "b", "c").parallel();
    Stream<String> second = Stream.of("X", "Y", "Z");
    List<String> strings = Stream.concat(first, second)  ❶
            .collect(Collectors.toList());
    List<String> stringList = Arrays.asList("a", "b", "c", "X",
    assertEquals(stringList, strings);
}
```

❶

First elements followed by second elements

If you want to add a third stream to the mix, you can next the concatenations, Example 4-55.

Example 4-55. Concatenating multiple streams

```
@Test
public void concatThree() throws Exception {
    Stream<String> first = Stream.of("a", "b", "c").parallel();
    Stream<String> second = Stream.of("X", "Y", "Z");
    Stream<String> third = Stream.of("alpha", "beta", "gamma");

    List<String> strings = Stream.concat(Stream.concat(first, s
            .collect(Collectors.toList());
    List<String> stringList = Arrays.asList("a", "b", "c",
        "X", "Y", "Z", "alpha", "beta", "gamma");
    assertEquals(stringList, strings);
}
```

This nesting approach works, but the Javadocs contain a note about this:

> Use caution when constructing streams from repeated concatenation. Accessing an element of a deeply concatenated stream can result in deep call chains, or even `StackOverflowException`

The idea is that the `concat` method essentially builds a binary tree of streams, which can grow unwieldy if too many are used.

An alternative approach is to use the `reduce` method to perform multiple concatenations, as in Example 4-56.

Example 4-56. Concatenating with reduce

```
@Test
public void reduce() throws Exception {
    Stream<String> first = Stream.of("a", "b", "c").parallel();
    Stream<String> second = Stream.of("X", "Y", "Z");
    Stream<String> third = Stream.of("alpha", "beta", "gamma");
    Stream<String> fourth = Stream.empty();

    List<String> strings = Stream.of(first, second, third, fou
            .reduce(Stream.empty(), Stream::concat)   ❶
            .collect(Collectors.toList());

    List<String> stringList = Arrays.asList("a", "b", "c",
        "X", "Y", "Z", "alpha", "beta", "gamma");
    assertEquals(stringList, strings);
```

```
}
```

❶

Using `reduce` with an empty stream and a binary operator

This works because the `concat` method when used as a method reference is a binary operator. Note this is simpler code, but doesn't fix the potential stack overflow problem.

Instead, when combining streams, the `flatMap` method is a natural solution, as in Example 4-57.

Example 4-57. Using flatMap to concatenate streams

```
@Test
public void flatMap() throws Exception {
    Stream<String> first = Stream.of("a", "b", "c").parallel();
    Stream<String> second = Stream.of("X", "Y", "Z");
    Stream<String> third = Stream.of("alpha", "beta", "gamma");
    Stream<String> fourth = Stream.empty();

    List<String> strings = Stream.of(first, second, third, fou
            .flatMap(Function.identity())
            .collect(Collectors.toList());
    List<String> stringList = Arrays.asList("a", "b", "c",
        "X", "Y", "Z", "alpha", "beta", "gamma");
    assertEquals(stringList, strings);
}
```

This approach works, but also has its quirks. Using `concat` creates a parallel stream if any of the input streams are parallel, but `flatMap` does not (Example 4-58).

Example 4-58. Parallel or not?

```
@Test
public void concatParallel() throws Exception {
    Stream<String> first = Stream.of("a", "b", "c").parallel();
    Stream<String> second = Stream.of("X", "Y", "Z");
    Stream<String> third = Stream.of("alpha", "beta", "gamma");
```

```
    Stream<String> total = Stream.concat(Stream.concat(first, s

    assertTrue(total.isParallel());
}

@Test
public void flatMapNotParallel() throws Exception {
    Stream<String> first = Stream.of("a", "b", "c").parallel();
    Stream<String> second = Stream.of("X", "Y", "Z");
    Stream<String> third = Stream.of("alpha", "beta", "gamma");
    Stream<String> fourth = Stream.empty();

    Stream<String> total = Stream.of(first, second, third, fou
            .flatMap(Function.identity());
    assertFalse(total.isParallel());
}
```

Still, you can always make the resulting parallel if you want by calling the `parallel` method, as long as you have not yet processed the data (Example 4-59).

Example 4-59. Making a flatMap stream parallel

```
@Test
public void flatMapParallel() throws Exception {
    Stream<String> first = Stream.of("a", "b", "c").parallel();
    Stream<String> second = Stream.of("X", "Y", "Z");
    Stream<String> third = Stream.of("alpha", "beta", "gamma");
    Stream<String> fourth = Stream.empty();

    Stream<String> total = Stream.of(first, second, third, fou
            .flatMap(Function.identity());
    assertFalse(total.isParallel());

    total = total.parallel();
    assertTrue(total.isParallel());
}
```

Since `flatMap` is an intermediate operation, the stream can still be modified using the `parallel` method, as shown.

In short, the `concat` method is effective for two streams, and can be used as part of a general reduction operation, but `flatMap` is a natural alternative.

See Also

See the excellent blog post online at *https://www.techempower.com/blog/2016/10/19/efficient-multiple-stream-concatenation-in-java/* for details, performance considerations, and more.

Lazy Streams

Problem

You want to process the minimum number of stream elements necessary to satisfy a condition.

Solution

Streams are already lazy and do not process elements until a terminal condition is reached. Then each elements is processed individually. If there is a short-circuiting operation at the end, the stream processing will terminate whenever all the conditions are satisfied.

Discussion

When you first encounter stream processing, it's tempting to think that much more effort is being expended than necessary. For example, consider taking a range of numbers between 100 and 200, doubling each of them, and then finding the first value that evenly divisible by three, as in Example 4-60.[7]

Example 4-60. A simple pipeline

```
// Find first even double of the numbers betwen 100 and 200 tha
OptionalInt firstEvenDoubleDivBy3 = IntStream.range(100, 200)
    .map(n -> n * 2)
    .filter(n -> n % 3 == 0)
    .findFirst();
System.out.println(firstEvenDoubleDivBy3);
// prints: Optional[204]
```

If you didn't know better, you might think a lot of wasted effort was expended:

- The range of numbers from 100 to 199 is created (100 operations)
- Each number is doubled (100 operations)
- Each number is checked for divisibility (100 operations)
- The first element of the resulting stream is returned (1 operation)

Since the first value that satisfies the stream requirements is 204, why process all the other numbers?

Fortunately, stream processing doesn't work that way. Streams are *lazy*, in that no work is done until the terminal condition is reached, and then each element is processed through the pipeline individually. To demonstrate this, Example 4-61 shows the same code, but refactored to show each element as it passes through the pipeline.

Example 4-61. Explicit processing of each stream element

```
public int multByTwo(int n) { ❶
    System.out.printf("Inside multByTwo with arg %d%n", n);
    return n * 2;
}

public boolean divByThree(int n) {  ❷
    System.out.printf("Inside divByThree with arg %d%n", n);
    return n % 3 == 0;
}

// ...

firstEvenDoubleDivBy3 = IntStream.range(100, 200)
    .map(this::multByTwo) ❶
    .filter(this::divByThree) ❷
    .findFirst();
```

❶

Method reference for multiply by two, with print

❷

Method reference for modulus 3, with print

The output this time is:

```
Inside multByTwo with arg 100
Inside divByThree with arg 200
Inside multByTwo with arg 101
Inside divByThree with arg 202
Inside multByTwo with arg 102
Inside divByThree with arg 204
First even divisible by 3 is Optional[204]
```

The value 100 goes through the `map` to produce 200, but does not pass the filter, so the stream moves to the value 101. That is mapped to 202, which also doesn’t pass the filter. Then the next value, 102, is mapped to 204, but that is divisible by three, so it passes. The stream processing terminates *after processing only three values*, using six operations.

This is one of the great advantages of stream processing over working with collections directly. With a collection, all of the operations would have to be performed before moving to the next step. With streams, the intermediate operations form a pipeline, but nothing happens until the terminal operation is reached. Then the stream processes only as many values as are necessary.

This isn't always relevant — if any of the operations are stateful, like sorting or adding them all together, then all the values are going to have to be processed anyway. But when you have stateless operations followed by a short-circuiting, terminal operation, the advantage is clear.

See Also

The differences between `findFirst` and `findAny` are discussed in "Finding The First Element In A Stream".

[1] Hopefully it doesn't destroy my credibility entirely to admit that I was able to recall the names of all six Brady Bunch kids without looking them up. Believe me, I'm as horrified as you are.

[2] Pretend, for the moment, that you didn't think to use the `sum` method.

[3] There are many ways to solve this problem, including just doubling the value returned by the `sum` method. The approach taken here illustrates how to use the two-argument form of `reduce`.

[4] Sorry about the pun

[5] Thanks to Stuart Marks for this explanation

[6] Technically this is known as a *concordance*.

[7] Thanks to the inimitable Venkat Subramaniam for the basis of this example.

Chapter 5. Comparators and Collectors

Java 8 enhances the `Comparator` interface with several static and default methods that make sorting operations much simpler. It’s now possible to sort a collection of POJOs by one property, then equal first properties by a second, then by a third, and so on, just with a series of library calls.

Java 8 also adds a new utility class called `java.util.stream.Collectors`, which provides static methods to convert from streams back into various types of collections. The collectors can also be applied “downstream”, meaning that they can post-process a grouping or partitioning operation.

The recipes in this section illustrate all these concepts.

Sorting Using A Comparator

Problem

You want to sort objects.

Solution

Use the `sorted` method on `Stream` with a `Comparator`, either implemented with a lambda expression or generated by one of the static `compare` methods on the `Comparator` interface.

Discussion

The `sorted` method on `Stream` produces a new, sorted stream using the natural ordering for the class. The natural ordering is specified by implementing the `java.util.Comparable` interface.

For example, consider sorting a collection of strings, as shown in Example 5-1.

Example 5-1. Sorting strings lexicographically.

```
private List<String> sampleStrings =
    Arrays.asList("this", "is", "a", "list", "of", "strings");

// Default sort from Java 7-
public List<String> defaultSort() {
    Collections.sort(sampleStrings);
    return sampleStrings;
}

// Default sort from Java 8+
public List<String> defaultSortUsingStreams() {
    return sampleStrings.stream()
        .sorted()
        .collect(Collectors.toList());
}
```

Java has had a utility class called `Collections` ever since the collections framework was added back in version 1.2. The static `sort` method on `Collections` takes a `List` as an argument, but returns `void`. The sort is destructive, modifying the supplied collection. This approach does not follow the functional principles supported by Java 8, which emphasize immutability.

Java 8 uses the `sorted` method on streams to do the same sorting, but produces a new stream rather than modifying the original collection. In this example, after sorting the collection, the returned list is sorted according to the natural ordering of the class. For strings, the natural ordering is lexicographical, which reduces to alphabetical when all the strings are lowercase, as in this example.

If you want to sort the strings in a different way, then there is an overloaded `sorted` method that takes a `Comparator` as an argument.

Example 5-2 shows a length sort for strings in two different ways.

Example 5-2. Sorting strings by length

```
// Sort by length with sorted
public List<String> lengthSortUsingSorted() {
    return sampleStrings.stream()
        .sorted((s1, s2) -> s1.length() - s2.length()) ❶
        .collect(toList());
}

// Length sort with comparingInt
public List<String> lengthSortUsingComparator() {
    return sampleStrings.stream()
        .sorted(Comparator.comparingInt(String::length)) ❷
        .collect(toList());
}
```

❶

Using a lambda for the Comparator

❷

Generating a Comparator using the comparingInt method

The argument to the `sorted` method is a `java.util.Comparator`, which is a functional interface. In `lengthSortUsingSorted`, a lambda expression is provided to implement the `compare` method in `Comparator`. In Java 7 and earlier, the implementation would normally be provided by an anonymous inner class, but here a lambda expression is all that is required.

Note

Java 8 added `sort(Comparator)` as a `default` instance method on `List`, equivalent to the `static sort(List, Comparator)` method on

`Collections`. Both are destructive sorts that return `void`, so the `sorted(Comparator)` approach on streams discussed here (which returns a new, sorted stream) is still preferred.

The second method, `lengthSortUsingComparator`, takes advantage of one of the static methods added to the `Comparator` interface. The `comparingInt` method takes an argument of type `ToIntFunction` that transforms the string into an int, called a *keyExtractor* in the docs, and generates a `Comparator` that sorts the collection using that key.

The added default methods in `Comparator` are extremely useful. While you can write a `Comparator` that sorts by length pretty easily, when you want to sort by more than one field that can get complicated. Consider sorting the strings by length, then equal length strings alphabetically. Using the default and static methods in `Comparator`, that becomes almost trivial, as shown in Example 5-3.

Example 5-3. Sorting by length, then equal lengths lexicographically

```
import static java.util.Comparator.comparing;
import static java.util.Comparator.naturalOrder;
import static java.util.Comparator.reverseOrder;
import static java.util.stream.Collectors.toList;

// class definition...

// Sort by length then alpha using sorted
public List<String> lengthSortThenAlphaSort() {
    return sampleStrings.stream()
        .sorted(comparing(String::length)
                    .thenComparing(naturalOrder()))
        .collect(toList());
}

// Sort by length then reverse alpha using sorted
public List<String> lengthSortThenReverseAlphaSort() {
    return sampleStrings.stream()
        .sorted(comparing(String::length)
                    .thenComparing(reverseOrder()))
        .collect(toList());
}
```

`Comparator` provides a `default` method called `thenComparing`. Just like `comparing`, it also takes a `Function` as an argument, known a key extractor. Chaining this to the `comparing` method returns a `Comparator` that compares by the first quantity, then equal first by the second, and so on.

Notice the static imports in this case that make the code easier to read. Once you get used to the static methods in both `Comparator` and `Collectors`, this becomes an easy way to simplify the code.

This approach works on any class, even if it does not implement `Comparable`. Consider the `Golfer` class shown in Example 5-4.

Example 5-4. A class for golfers

```
public class Golfer {
    private String first;
    private String last;
    private int score;

    // constructors ...

    // getters and setters ...

    // toString, equals, hashCode ...
}
```

To create a leader board at a tournament, it makes sense to sort by score, then by last name, and then by first name. Example 5-5 shows how to do that.

Example 5-5. Sorting golfers

```
private List<Golfer> golfers = Arrays.asList(
    new Golfer("Jack", "Nicklaus", 68),
    new Golfer("Tiger", "Woods", 70),
    new Golfer("Tom", "Watson", 70),
    new Golfer("Ty", "Webb", 68),
    new Golfer("Bubba", "Watson", 70)
);

public List<Golfer> sortByScoreThenLastThenFirst() {
    golfers.stream()
        .sorted(comparingInt(Golfer::getScore)
```

```
                    .thenComparing(Golfer::getLast)
                    .thenComparing(Golfer::getFirst))
        .collect(toList());
}
```

The output from calling `sortByScoreThenLastThenFirst` is shown in Example 5-6.

Example 5-6. Sorted golfers

```
Golfer{first='Jack', last='Nicklaus', score=68}
Golfer{first='Ty', last='Webb', score=68}
Golfer{first='Bubba', last='Watson', score=70}
Golfer{first='Tom', last='Watson', score=70}
Golfer{first='Tiger', last='Woods', score=70}
```

The golfers are sorted by score, so Nicklaus and Webb come before Woods and both Watsons.[1] Then equal scores are sorted by last name, putting Nicklaus before Webb and Watson before Woods. Finally, equal scores and last names are sorted by first name, putting Bubba Watson before Tom Watson.

The default and static methods in `Comparator`, along with the new `sorted` method on `Stream`, makes generating complex sorts easy.

See Also

Converting a Stream into a Collection

Problem

After stream processing, you want to convert to a `List`, `Set`, or other linear collection.

Solution

Use the `toList`, `toSet`, or `toCollection` methods in the `Collectors` utility class.

Discussion

Idiomatic Java 8 often involves passing elements of a stream through a pipeline of intermediate operations, finishing with a terminal operation. One terminal operation is the `collect` method, which is used to convert a `Stream` into a collection.

The `collect` method in `Stream` has two overloaded versions, as shown in Example 5-7.

Example 5-7. The collect method in Stream<T>

```
<R,A> R collect(Collector<? super T,A,R> collector)
<R>   R collect(Supplier<R> supplier,
                BiConsumer<R,? super T> accumulator,
                BiConsumer<R,R> combiner)
```

This recipe deals with the first version, which takes a `Collector` as an argument. Collectors perform a “mutable reduction operation” that accumulates elements into a result container. Here the result will be a collection.

`Collector` is an interface, so it can’t be instantiated. The interface contains a static `of` method for producing them, but there is often a better, or at least easier, way.

Tip

The Java 8 API frequently uses a static method called `of` as a factory method.

In this recipe, the static methods in the `Collectors` class will be used to produce `Collector` instances, which are used as the argument to `Stream.collect` to populate a collection.

A simple example that creates a `List` is shown in Example 5-8.[2]

Example 5-8. Creating a list

```
public List<String> createList() {
    return Stream.of("Mr. Furious", "The Blue Raja", "The Shove
        "The Bowler", "Invisible Boy", "The Spleen", "The Sphin
        .collect(Collectors.toList()); ❶
}
```

❶

Collect to a `List`

This method creates and populates an `ArrayList` with the given stream elements. Creating a `Set` is just as easy, as in Example 5-9.

Example 5-9. Creating a set

```
public Set<String> createSet() {
    return Stream.of("Casanova Frankenstein", "The Disco Boys",
        "The Not-So-Goodie Mob", "The Suits", "The Suzies",
        "The Furriers", "The Furriers")  ❶
        .collect(Collectors.toSet());    ❷
}
```

❶

Duplicate name, removed when converting to a `Set`

❷

Collect to a `Set`

This method creates an instance of `HashSet` and populates it, leaving out any duplicates.

Both these examples used the default data structures.[3] If you wish to specify a particular data structure, use the `Collectors.toCollection` method, which takes a `Supplier` as an argument. See Example 5-10 for an example.

Example 5-10. Creating a linked list

```
public List<String> createDeque() {
    return Stream.of("Hank Azaria", "Janeane Garofalo", "Willia
        "Paul Reubens", "Ben Stiller", "Kel Mitchell", "Wes Stu
```

```
        .collect(Collectors.toCollection(LinkedList::new)); ❶
}
```

❶

Collect to a `LinkedList`

This method instantiates a `LinkedList` and populates it with the given names. Note the use of the constructor reference as an argument of type `Supplier`.

The `Collectors` class also contains a method to create an array of objects. There are two overloads of the `toArray` method:

```
    Object[] toArray();
<A> A[]      toArray(IntFunction<A[]> generator);
```

The former returns an array containing the elements of this stream, but without specifying the type. The latter takes a function that produces a new array of desired type with length equal to the size of the stream, and is easiest to use with an array constructor reference as shown in Example 5-11.

Example 5-11. Creating an array

```
public String[] createArray() {
    return Stream.of("The Waffler", "Reverse Psychologist", "PM
        .toArray(String[]::new); ❶
}
```

❶

Array constructor reference as a `Supplier`

The returned array is of the specified type, whose length matches the number of elements in the stream.

To transform into a `Map`, the `Collectors.toMap` method requires two `Function` instances — one for the keys and one for the values.

Consider an `Actor` POJO, which wraps a `name` and a `role`. If you have a `Set` of `Actor` instances from a given movie, the code in Example 5-12 creates a `Map` from them.

Example 5-12. Creating a map

```
Set<Actor> actors = mysteryMen.getActors();

Map<String, String> actorMap = actors.stream()
    .collect(Collectors.toMap(Actor::getName, Actor::getRole));

actorMap.forEach((key,value) ->
    System.out.printf("%s played %s%n", key, value));
```

❶

Functions to produce keys and values

The output is

```
Janeane Garofalo played The Bowler
Greg Kinnear played Captain Amazing
William H. Macy played The Shoveler
Paul Reubens played The Spleen
Wes Studi played The Sphinx
Kel Mitchell played Invisible Boy
Geoffrey Rush played Casanova Frankenstein
Ben Stiller played Mr. Furious
Hank Azaria played The Blue Raja
```

Similar code works for `ConcurrentMap` using the `toConcurrentMap` method.

See Also

Adding a Linear Collection to a Map

Problem

You want to add a collection of objects to a map, where the key is one of the object properties and the value is the object itself.

Solution

Use the `toMap` method of `Collectors`, along with `Function.identity`.

Discussion

This is a short, very focused use case, but when it comes up in practice the solution here can be quite convenient.

Say you had a `List` of `Book` instances, where `Book` is a simple POJO that has an id, a name, and a price. An abbreviated form of the `Book` class is shown in Example 5-13.

Example 5-13. A simple POJO representing a book

```
public class Book {
    private int id;
    private String name;
    private double price;

    // constructors

    // getters and setters

    // equals, hashCode, and toString
}
```

Now assume you have a collection of `Book` instances, as shown in Example 5-14.

Example 5-14. A collection of books

```
List<Book> books = Arrays.asList(
    new Book(1, "Modern Java Recipes", 49.99),
    new Book(2, "Java 8 in Action", 49.99),
    new Book(3, "Java SE8 for the Really Impatient", 39.99),
    new Book(4, "Functional Programming in Java", 27.64),
    new Book(5, "Making Java Groovy", 45.99),
    new Book(6, "Head First Java", 26.97),
    new Book(7, "Effective Java", 35.47),
    new Book(8, "Gradle Recipes for Android", 23.76),
    new Book(9, "Spring Boot in Action", 39.97)
);
```

In many situations, instead of a `List` you might want a `Map`, where the keys are the book ids and the values are the books themselves. This is really easy to accomplish using the `toMap` method in `Collectors`, as shown two different ways in Example 5-15.

Example 5-15. Adding the books to a map

```
Map<Integer, Book> bookMap = books.stream()
    .collect(Collectors.toMap(Book::getId, b -> b)); ❶

bookMap = books.stream()
    .collect(Collectors.toMap(Book::getId, Function.identity())
❷
```

❶

identity lambda, given `b` return `b`

❷

static `identity` method in `Function` does the same thing

The `toMap` method in `Collectors` takes two `Function` instances as arguments, the first of which generates a key and the second of which generates the value from the provided object. In this case, the key is mapped by the `getId` method in `Book`, and the value is the book itself.

The first `toMap` in Example 5-15 uses the `getId` method to map to the key and an explicit lambda expression that returns its argument to map the value. The second example uses the static `identity` method in `Function` to do perform the same operation.

The two static `identity` methods

The static `identity` method in `Function` has the signature

```
static <T> Function<T,T>        identity()
```

The implementation in the standard library is shown in Example 5-16.

Example 5-16. The static identity method in Function

```
static <T> Function<T, T> identity() {
    return t -> t;
}
```

The `UnaryOperator` class extends `Function`, but you can't override a static method. In the JavaDocs, it also declares a static `identity` method.

```
static <T> UnaryOperator<T>     identity()
```

Its implementation in the standard library is essentially the same, as shown in Example 5-17.

Example 5-17. The static identity method in UnaryOperator

```
static <T> UnaryOperator<T> identity() {
    return t -> t;
}
```

The differences are only in the way you call them (from the two interface names) and the corresponding return types. In this case, it doesn't matter which one you use, but it's interesting to see that they're both there.

Whether you decide to supply an explicit lambda or use the static method is merely a matter of style. Either way, it is easy to add collection values to a map where the key is a property of the object and the value is the object itself.

See Also

Sorting Maps

Problem

You want to sort a map by key or by value.

Solution

Use the new static methods in the `Map.Entry` interface.

Discussion

The `Map` interface has always contained a public, static, inner interface called `Map.Entry`, which represents a key-value pair. The `Map.entrySet` method returns a `Set` of `Map.Entry` elements. Prior to Java 8, the primary methods used in this interface were `getKey` and `getValue`, which do what you'd expect.

In Java 8, the static methods in Table 5-1 have been added:

Table 5-1. Static methods in `Map.Entry` (from Java 8 docs)

`comparingByKey()`	Returns a comparator that compares `Map.Entry` in natural order on key
`comparingByKey(Comparator<? super K> cmp)`	Returns a comparator that compares `Map.Entry` by key using the given `Comparator`
`comparingByValue()`	Returns a comparator that compares `Map.Entry` in natural order on value
`comparingByValue(Comparator<? super V> cmp)`	Returns a comparator that compares `Map.Entry` by value using the given `Comparator`

To demonstrate how to use them, Example 5-18 generates a map of word lengths to number of words in a dictionary. Every Unix system contains a file in the `usr/share/dict` directory holding the contents of Webster's 2nd edition dictionary, with one word per line. The `Files.lines` method can be used to read a file and produce a stream of strings containing those lines. In this case, the stream will contain each word from the dictionary.

Example 5-18. Reading the dictionary file into a map

```
System.out.println("\nNumber of words of each length:");
try (Stream<String> lines = Files.lines(dictionary)) {
    lines.filter(s -> s.length() > 20)
        .collect(Collectors.groupingBy(
            String::length, Collectors.counting()))
        .forEach((len, num) -> System.out.printf("%d: %d%n", le
} catch (IOException e) {
    e.printStackTrace();
}
```

This example is discussed in a recipe in the I/O section, but to summarize:

- The file is read inside a *try-with-resources* block. `Stream` implements `AutoCloseable`, so when the try block exits, Java calls the `close` method on `Stream` which then calls the `close` method on `File`

- The filter restricts further processing to only words of at least 20 characters in length

- The `groupingBy` method of `Collectors` takes a `Function` as the first argument, representing the classifier. Here, the classifier is the length of each string. If you only provide one argument, the result is a `Map` where the keys are the values of the classifier and the values are lists of elements that match the classifier. In this case, `groupingBy(String::length)` would have a produced a `Map<Integer,List<String>>` where the keys are the word lengths and the values are lists of words of that length

- In this case, the two-argument version of `groupingBy` lets you supply another `Collector`, called a *downstream* collector, that post processes the lists of words. In this case, the return type is `Map<Integer,Long>`, where the keys are the word lengths and the values are the number of words of that length in the dictionary.

The result is:

```
Number of words of each length:
21: 82
22: 41
```

```
23: 17
24: 5
```

In other words, there are 82 words of length 21, 41 words of length 22, 17 words of length 23, and 5 words of length 24[4].

The results show that the map is printed in ascending order of word length. In order to see it in descending order, use `Map.Entry.comparingByKey` as in Example 5-19.

Example 5-19. Sorting the map by key

```
System.out.println("\nNumber of words of each length (desc orde
try (Stream<String> lines = Files.lines(dictionary)) {
    Map<Integer, Long> map = lines.filter(s -> s.length() > 20)
        .collect(Collectors.groupingBy(
            String::length, Collectors.counting()));

    map.entrySet().stream()
        .sorted(Map.Entry.comparingByKey(Comparator.reverseOrde
        .forEach(e -> System.out.printf("Length %d: %2d words%r
            e.getKey(), e.getValue()));
} catch (IOException e) {
    e.printStackTrace();
}
```

After computing the `Map<Integer,Long>`, this operation extracts the `entrySet` and produces a stream. The `sorted` method on stream is used to produce a sorted stream using the provided comparator.

In this case, `Map.Entry.comparingByKey` generates a comparator that sorts by the keys, and using the overload that takes a comparator allows the code to specify that we want it in reverse order.

Note

The `sorted` method on `Stream` produces a new, sorted stream that does not modify the source. The original `Map` is unaffected.

The result is:

```
Number of words of each length (desc order):
Length 24:  5 words
Length 23: 17 words
Length 22: 41 words
Length 21: 82 words
```

The other sorting methods listed in Table 5-1 are used similarly.

See Also

An additional example of sorting a map by keys or values is shown in [Link to Come]. Downstream comparators are discussed in “Downstream Collectors”.

Partitioning and Grouping

Problem

You want to divide a collection of elements into categories.

Solution

The `Collectors.partitioningBy` method splits elements into those that satisfy a `Predicate` and those that do not. The `Collectors.groupingBy` method produces a `Map` of categories, where the values are the elements in each category.

Discussion

Say you have a collection of strings. If you want to split them into those that have even lengths and those that have odd lengths, you can use `Collectors.partitioningBy`, as in Example 5-20.

Example 5-20. Partitioning strings by even or odd lengths

```
List<String> strings = Arrays.asList("this", "is", "a", "long",
        "strings", "to", "use", "as", "a", "demo");

Map<Boolean, List<String>> lengthMap = strings.stream()
    .collect(Collectors.partitioningBy(s -> s.length() % 2 == (

lengthMap.forEach((key,value) -> System.out.printf("%5s: %s%n",
//
// false: [a, strings, use, a]
//  true: [this, is, long, list, of, to, as, demo]
```

The signature of the two `partitioningBy` methods are:

```
static <T> Collector<T,?,Map<Boolean,List<T>>>  partitioningBy
static <T,D,A> Collector<T,?,Map<Boolean,D>>    partitioningBy
```

The first `partitioningBy` method takes a `Predicate` as an argument. It divides the elements into those that satisfy the `Predicate` and those that don’t. You always get a `Map` as a result with exactly two entries: one for the values that satisfy the `Predicate`, and one for the elements that do not.

The overloaded version of the method takes a second argument of type `Collector`, called a *downstream collector*. This allows you to post-process the lists returned by the partition, and is discussed in “Downstream Collectors”.

The `groupingBy` method performs an operation like a “group by” statement in SQL. It returns a map where the keys are the groups and the values are lists of elements in each group.

Note

If you are getting your data from a database, by all means do any grouping operations there. The new API methods are convenience methods for data in memory.

The signature for the `groupingBy` method is:

```
static <T,K> Collector<T,?,Map<K,List<T>>>      groupingBy(Func
```

The `Function` argument takes each element of the stream and extracts a property to group by. This time, rather than simply partition the strings into two categories, consider separating them by length, as in Example 5-21.

Example 5-21. Grouping strings by length

```
List<String> strings = Arrays.asList("this", "is", "a", "long",
        "strings", "to", "use", "as", "a", "demo");

Map<Integer, List<String>> lengthMap = strings.stream()
    .collect(Collectors.groupingBy(String::length));

lengthMap.forEach((k,v) -> System.out.printf("%d: %s%n", k, v))
//
// 1: [a, a]
// 2: [is, of, to, as]
// 3: [use]
// 4: [this, long, list, demo]
// 7: [strings]
```

The keys in the resulting maps are the lengths of the strings (1, 2, 3, 4, and 7) and the values are lists of strings of each length.

See Also

“Downstream Collectors” shows how to post-process the lists returned by a `groupingBy` or `partitioningBy` operation.

Downstream Collectors

Problem

You want to post-process the collections returned by a `groupingBy` or `partitioningBy` operation.

Solution

Use one of the static utility methods from the `java.util.stream.Collectors` class.

Discussion

“Partitioning and Grouping” showed how to separate elements into multiple categories. The `partitioningBy` and `groupingBy` methods return a `Map` where the keys were the categories (simply booleans `true` and `false` for `partitioningBy`, but objects for `groupingBy`) and the values were lists of elements that satisfied each category. Recall the example partitioning strings by even and odd lengths, show in Example 5-20 but repeated here for convenience.

Example 5-22. Partitioning strings by even or odd lengths

```
List<String> strings = Arrays.asList("this", "is", "a", "long",
        "strings", "to", "use", "as", "a", "demo");

Map<Boolean, List<String>> lengthMap = strings.stream()
    .collect(Collectors.partitioningBy(s -> s.length() % 2 == (

lengthMap.forEach((key,value) -> System.out.printf("%5s: %s%n",
//
// false: [a, strings, use, a]
//  true: [this, is, long, list, of, to, as, demo]
```

Rather than the actual lists, you may be interested in how many fall into each category. In other words, instead of producing a `Map` whose values are `List<String>`, you might want just the numbers of element in each of the lists. The `partitioningBy` method has an overloaded version whose second argument is of type `Collector`:

```
static <T,D,A> Collector<T,?,Map<Boolean,D>>    partitioningBy
```

This is where the static `Collectors.counting` method becomes useful. Example 5-23 shows how it works.

Example 5-23. Counting the partitioned strings

```
Map<Boolean, Long> numberLengthMap = strings.stream()
    .collect(Collectors.partitioningBy(s -> s.length() % 2 == (
```

```
                    Collectors.counting()));   ❶

numberLengthMap.forEach((k,v) -> System.out.printf("%5s: %d%n",
//
// false: 4
//  true: 8
```

❶

downstream collector

This is called a *downstream collector*, because it is post-processing the resulting lists downstream, i.e., after the partitioning operation is completed.

The `groupingBy` method also has an overload that takes a downstream collector.

```
static <T,K,A,D> Collector<T,?,Map<K,D>>        groupingBy(Func
```

(Aren't Java generics fun? Of course the chosen letters are arbitrary, but presumably here `T` is the type of the element in the collection, `K` is the key type for the resulting map, `A` is an accumulator, and `D` is the type of the downstream collector. The `?` represents "unknown". See [Link to Come] for details.)

Several methods in `Stream` have analogs in the `Collectors` class. Table 5-2 shows how they align.

Table 5-2. Collectors methods similar to Stream methods

Stream	Collectors
`count`	`counting`
`map`	`mapping`
`min`	`minBy`
`max`	`maxBy`

IntStream.sum	summingInt
DoubleStream.sum	summingDouble
LongStream.sum	summingLong
IntStream.summarizing	summarizingInt
DoubleStream.summarizing	summarizingDouble
LongStream.summarizing	summarizingLong

See Also

[Link to Come] shows an example of a downstream collector when determining the longest words in a dictionary. "Partitioning and Grouping" discusses the `partitionBy` and `groupingBy` methods in more detail.

Finding Max and Min Values

Problem

You want to determine the maximum or minimum value in a stream.

Solution

You have several choices: the `maxBy` and `minBy` methods on `BinaryOperator`, the `max` and `min` methods on `Stream`, or the `maxBy` and `minBy` utility methods on `Collectors`.

Discussion

A `BinaryOperator` is one of the new functional interfaces in the `java.util.function` package. It extends `BiFunction` and applies when both arguments to the function and the return type are all from the same class.

The `BinaryOperator` interface adds two static methods:

```
static <T> BinaryOperator<T> maxBy(Comparator<? super T> compa
static <T> BinaryOperator<T> minBy(Comparator<? super T> compa
```

Each of these returns a `BinaryOperator` which uses the supplied `Comparator`.

To demonstrate the various ways to get the maximum value from a stream, consider a POJO called `Employee` that holds three attributes: `name`, `salary`, and `department`, as in Example 5-24.

Example 5-24. Employee POJO

```
public class Employee {
    private String name;
    private Integer salary;
    private String department;

    // ... constructors, getters and setters,
    //     overrides for toString, equals, hashCode ...
}

List<Employee> employees = Arrays.asList(                    ❶
        new Employee("Cersei",     250_000, "Lannister"),
        new Employee("Jamie",      150_000, "Lannister"),
        new Employee("Tyrion",       1_000, "Lannister"),
        new Employee("Tywin",    1_000_000, "Lannister"),
        new Employee("Jon Snow",    75_000, "Stark"),
        new Employee("Robb",       120_000, "Stark"),
        new Employee("Eddard",     125_000, "Stark"),
        new Employee("Sansa",            0, "Stark"),
        new Employee("Arya",         1_000, "Stark"));
```

```
Employee defaultEmployee =                                                ❷
    new Employee("A man (or woman) has no name", 0, "Black and
```

❶

Collection of employees

❷

Default for when the stream is empty

Given a collection of employees, you can use the `reduce` method on `Stream`, which takes a `BinaryOperator` as an argument. The snippet in Example 5-25 shows how to get the employee with the largest salary.

Example 5-25. Using BinaryOperator.maxBy

```
Optional<Employee> optionalEmp = employees.stream()
    .reduce(BinaryOperator.maxBy(Comparator.comparingInt(Emplo

System.out.println("Emp with max salary: " +
    optionalEmp.orElse(defaultEmployee));
```

The `reduce` method requires a `BinaryOperator`. The static `maxBy` method produces that `BinaryOperator` based on the supplied `Comparator`, which in this case compares employees by salary.

This works, but there's actually a convenience method called `max` that can be applied directly to the stream.

```
Optional<T> max(Comparator<? super T> comparator)
```

Using that method directly is shown in Example 5-26.

Example 5-26. Using Stream.max

```
optionalEmp = employees.stream()
        .max(Comparator.comparingInt(Employee::getSalary));
```

The result is the same.

Note that there is also a method called `max` on the primitive streams (`IntStream`, `LongStream`, and `DoubleStream`) that takes no arguments. Example 5-27 shows that method in action.

Example 5-27. Finding the highest salary

```
OptionalInt maxSalary = employees.stream()
        .mapToInt(Employee::getSalary)
        .max();
System.out.println("The max salary is " + maxSalary);
```

In this case, the `mapToInt` method is used to convert the stream of employees into a stream of integers by invoking the `getSalary` method, and the returned stream is an `IntStream`. The `max` method then returns an `OptionalInt`.

There is also a static method called `maxBy` in the `Collectors` utility class. You can use it directly here, as in Example 5-28.

Example 5-28. Using Collectors.maxBy

```
optionalEmp = employees.stream()
    .collect(Collectors.maxBy(Comparator.comparingInt(Employee:
```

This is awkward, however, and can be replaced by the `max` method on stream, as shown in the preceding example. The `maxBy` method on `Collectors` is helpful when used as a downstream collector, i.e., when post-processing a grouping or partitioning operation. The code in Example 5-29 uses `groupingBy` on stream to create a map of departments to lists of employees, but then determines the employee with the greatest salary in each department.

Example 5-29. Using Collectors.maxBy as a downstream collector

```
Map<String, Optional<Employee>> map = employees.stream()
    .collect(Collectors.groupingBy(Employee::getDepartment,
          Collectors.maxBy(Comparator.comparingInt(Employee::ge

map.forEach((house, emp) ->
        System.out.println(house + ": " + emp.orElse(defaultEmp
```

The `minBy` method in each of these classes works the same way.

See Also

Creating Immutable Collections

Problem

You want to create an immutable list, set, or map using the `Stream` API.

Solution

Use the new static method `collectingAndThen` in the `Collectors` class.

Discussion

With its focus on parallelization and clarity, functional programming favors using immutable objects wherever possible. The Collections framework, added in Java 1.2, has always had methods to create immutable collections from existing ones, though in a somewhat awkward fashion.

The `Collections` utility class has methods `unmodifiableList`, `unmodifiableSet`, and `unmodifiableMap` (along with a few other methods with the same `unmodifiable` prefix), as shown in Example 5-30.

Example 5-30. Unmodifiable methods in the Collections class

```
static <T> List<T>    unmodifiableList(List<? extends T> list)
static <T> Set<T>     unmodifiableSet(Set<? extends T> s)
static <K,V> Map<K,V> unmodifiableMap(Map<? extends K,? extends
```

In each case, the argument to the method is an existing list, set, or map, and the resulting list, set, or map has the same elements as the argument, but with an important difference: all the methods that could modify the collection, like `add` or `remove`, now throw `java.lang.UnsupportedOperationException`.

Prior to Java 8, if you received the individual values as an argument, using a variable argument list, you produced an unmodifiable list or set as shown in Example 5-31.

Example 5-31. Creating unmodifiable lists or sets prior to Java 8

```
@SafeVarargs  ❶
public final <T> List<T> createImmutableListJava7(T... elements
    return Collections.unmodifiableList(Arrays.asList(elements)
}

@SafeVarargs  ❶
public final <T> Set<T> createImmutableSetJava7(T... elements)
    return Collections.unmodifiableSet(new HashSet<>(Arrays.asL
}
```

❶

> You promise not to corrupt the input array type. See [Link to Come] for details

The idea in each case is to start by taking the incoming values and convert them into a `List`. Then you can wrap the resulting list using `unmodifiableList`, or, in the case of a `Set`, use the list as the argument to a set constructor before using the `unmodifiableSet` method.

In Java 8, with the new stream API, you can take advantage of the static `Collectors.collectingAndThen` method instead, as in Example 5-32.

Example 5-32. Creating unmodifiable lists or sets in Java 8

```
import static java.util.stream.Collectors.collectingAndThen;
import static java.util.stream.Collectors.toList;
import static java.util.stream.Collectors.toSet;

// ... define a class with the following methods ...

@SafeVarargs
public final <T> List<T> createImmutableList(T... elements) {
    return Arrays.stream(elements)
        .collect(collectingAndThen(toList(),
                     Collections::unmodifiableList));
}

@SafeVarargs
public final <T> Set<T> createImmutableSet(T... elements) {
    return Arrays.stream(elements)
        .collect(collectingAndThen(toSet(), Collections::unmod:
}
```

The `Collectors.collectingAndThen` method takes two arguments: a downstream `Collector` and a `Function` called a *finisher*. The idea is to stream the input elements and then collect them into a list or set, but specify the unmodifiable function that wraps the resulting collection.

Converting a series of input elements into an unmodifiable `Map` isn't as clear, partly because it's not obvious which of the input elements would be assumed to be keys and which would be values. The code shown in Example 5-33[5] creates an immutable map is very awkward way, using an instance initializer.

Example 5-33. Creating an immutable map

```
Map<String, Integer> map = Collections.unmodifiableMap(
  new HashMap<String, Integer>() {{
    put("have", 1);
    put("the", 2);
    put("high", 3);
    put("ground", 4);
}});
```

Readers who are familiar with Java 9, however, already know that this entire recipe can be replaced with a very simple set of factory methods, `List.of`, `Set.of`, and `Map.of`.

See Also

[Link to Come] shows the new factory methods in Java 9 that automatically create immutable collections.

Implementing the Collector Interface

Problem

You need to implement `java.util.stream.Collector` manually, because none of the factory methods in the `java.util.stream.Collectors` class give you exactly what you need.

Solution

Provide lambda expressions or method references for the supplier, accumulator, combiner, and finisher functions used by the `Collector.of` factory methods, along with any desired characteristics.

Discussion

The utility class `java.util.stream.Collectors` has several convenient static methods whose return type is `Collector`. Examples are `toList`, `toSet`, `toMap`, and even `toCollection`, each of which is illustrated elsewhere in this book. Instances of classes that implement `Collector` are sent as arguments to the `collect` method on `Stream`. For example, in Example 5-34, the method accepts string arguments and returns a `List` containing only those whose length is even.

Example 5-34. Using collect to return a List

```
public List<String> evenLengthStrings(String... strings) {
    return Stream.of(strings)
        .filter(s -> s.length() % 2 == 0)
        .collect(Collectors.toList());  ❶
}
```

❶

Collect even length strings into a list

If you need to write your own collectors, however, the procedure is a bit more complicated. Collectors use five functions that work together to accumulate entries into a mutable container and optionally transform the result. The four functions are called `supplier`, `accumulator`, `combiner`, `finisher`, and `characteristics`.

Taking the `characteristics` function first, it represents an immutable `Set` of elements of an `enum` type `Collector.Characteristics`. The possible values are `CONCURRENT`, `IDENTITY_FINISH`, and `UNORDERED`. `CONCURRENT` means that the result container can support the accumulator function being called concurrently on the result container from multiple threads. `UNORDERED` says that the collection operation does not need to preserve the encounter order of the elements. `IDENTITY_FINISH` means that the finishing function returns its argument without any changes.

The purpose of each of the required methods is:

`supplier()`

: create the accumulator container using a `Supplier<A>`

`accumulator()`

: add a single new data element to the accumulator container using a `BiConsumer<A,T>`

`combiner()`

: merge two accumulator containers using a `BinaryOperator<A>`

`finisher()`

: transform the accumulator container into the result container using a `Function<A,R>`

`characteristics()`

: a `Set<Collector.Characteristics>` chosen from the enum values

As usual, an understanding of the functional interfaces defined in the `java.util.function` package makes everything clearer. A `Supplier` is used to create the container used to accumulate temporary results. A `BiConsumer` adds a single element to the accumulator container. A `BinaryOperator` means that both input types and the output type are the same, so here the idea is to combine two accumulator containers into one. A `Function` finally transforms the accumulator container into the desired result container.

Each of these methods is invoked during the collection process, which is triggered by (for example) the `collect` method on `Stream`. Conceptually, the collection process is equivalent to the (generic) code shown in Example 5-35, taken from the JavaDocs.

Example 5-35. How the Collector methods are used

```
R container = collector.supplier.get(); ❶
```

```
for (T t : data) {
    collector.accumulator().accept(container, t); ❷
}
return collector.finisher().apply(container); ❸
```

❶

Create the accumulator container

❷

Add each element to the accumulator container

❸

Convert the accumulator container to the result container using the finisher

Conspicuous by its absense is any mention of the `combiner` function. If your stream is sequential, you don’t need it — the algorithm proceeds as described. If, however, you are operating on a parallel stream, then the work is divided into multiple regions, each of which produces an accumulator container. The combiner is then used during the join process to merge the accumulator containers together into a single one before applying the finisher function.

An example, similar to that shown in Example 5-34, is given in Example 5-36.

Example 5-36. Using collect to return an unmodifiable SortedSet

```
public SortedSet<String> oddLengthStringSet(String... strings)
        Collector<String, ?, SortedSet<String>> intoSet =
                Collector.of(TreeSet<String>::new,          ❶
                        SortedSet::add,                      ❷
                        (left, right) -> {                   ❸
                              left.addAll(right);
                              return left;
                        },
                        Collections::unmodifiableSortedSet); ❹
        return Stream.of(strings)
                .filter(s -> s.length() % 2 != 0)
                .collect(intoSet);
    }
```

❶

Supplier to create a new `TreeSet`

❷

BiConsumer to add each string to the `TreeSet`

❸

BinaryOperator to combine two `SortedSet` instances into one

❹

Finisher function to create an unmodifiable set

The result will be a sorted, unmodifiable set of strings, ordered lexicographically.

This example used one of the two overloaded versions of the `static` `of` method for producing collectors, whose signatures are:

```
static <T,A,R> Collector<T,A,R> of(Supplier<A> supplier,
    BiConsumer<A,T> accumulator,
    BinaryOperator<A> combiner,
    Function<A,R> finisher,
    Collector.Characteristics... characteristics)
static <T,R> Collector<T,R,R>   of(Supplier<R> supplier,
    BiConsumer<R,T> accumulator,
    BinaryOperator<R> combiner,
    Collector.Characteristics... characteristics)
```

Given the convenience methods in the `Collectors` class that produce collectors for you, you rarely need to make one of your own this way. Still, it's a useful skill to have, and once again illustrates how the functional interfaces in the `java.util.function` package come together to create interesting objects.

See Also

The finisher function is an example of a downstream collector, discussed further in "Downstream Collectors". The `Supplier`, `Function`, and `BinaryOperator` functional interfaces are discussed in various recipes in Chapter 2. The static utility methods in `Collectors` are discussed in "Converting a Stream into a Collection".

[1] Ty Webb, of course, is from the movie *Caddyshack*. Judge Smails: "Ty, what did you shoot today?" Ty Webb: "Oh, Judge, I don't keep score." Smails: "Then how do you measure yourself with other golfers?" Webb: "By height." Adding a sort by height is left to the reader as an easy exercise.

[2] The names in this recipe come from *Mystery Men*, one of the great overlooked movies of the 90s. (Mr. Furious: "Lance Hunt *is* Captain Amazing." The Shoveler: "Lance Hunt wears glasses. Captain Amazing *doesn't* wear glasses." Mr. Furious: "He takes them off when he transforms." The Shoveler: "That doesn't make any sense! He wouldn't be able to *see*!")

[3] `ArrayList` for `List` and `HashSet` for `Set`

[4] For the record, those five longest words are formaldehydesulphoxylate, pathologicopsychological, scientificophilosophical, tetraiodophenolphthalein, and thyroparathyroidectomize. Good luck with that, spell checker.

[5] From the blog post "Java 9's Immutable Collections Are Easier To Create But Use With Caution" by Carl Martensen, *http://carlmartensen.com/immutability-made-easy-in-java-9*

Chapter 6. The Optional Type

> Sigh, why does everything related to `Optional` have to take 300 messages?
>
> Brian Goetz, *lambda-libs-spec-experts mailing list (23 Oct 2013)*

The Java 8 API introduces a new class called `java.util.Optional<T>`. While many developers assume that the goal of `Optional` is to remove `NullPointerExceptions` from your code, that's not its real purpose. Instead, `Optional` is designed to communicate to the user when a returned value may legitimately be null. This situation can arise whenever a stream of values is filtered by some condition that happens to leave no elements remaining.

In the `Stream` API, the following methods return an `Optional` if no elements remain in the stream:

- `reduce`
- `min`
- `max`
- `findFirst`
- `findAny`

An instance of `Optional` can be in one of two states: a reference to an instance of type `T`, or empty. The former case is called **present**, and the latter is known as **empty** (as opposed to `null`).

Warning

While `Optional` is a reference type, it should never be assigned a value of `null`. Doing so is a serious error.

This section looks at the idiomatic ways to use `Optional`. While the proper use of `Optional` is likely to be a lively source of discussions in your company[1], the good news is that there are standard recommendations for its proper use. Following these principles should help keep your intentions clear and maintainable.

Creating An Optional

Problem

You need to return an `Optional` from an existing value.

Solution

Use `Optional.of`, `Optional.ofNullable`, or `Optional.empty`.

Discussion

Like many other new classes in the Java 8 API, instances of `Optional` are immutable. The API refers to `Optional` as a *value-based class*, meaning instances:

- are final and immutable (though they may contain references to mutable objects)[2],
- have no public constructors, and thus must be instantiated by factory methods,
- have implementations of `equals`, `hashCode`, and `toString` that are based only on their state

`Optional` and Immutability

Instances of `Optional` are immutable, but the objects they wrap may not be. If you create an `Optional` that contains an instance of a mutable object, you can still modify the instance. See, for example, Example 6-1.

Example 6-1.

```
AtomicInteger counter = new AtomicInteger();
Optional<AtomicInteger> opt = Optional.ofNullable(counter);

System.out.println(optional);      // Optional[0]

counter.incrementAndGet();         ❶
System.out.println(optional);      // Optional[1]

optional.get().incrementAndGet(); ❷
System.out.println(optional);      // Optional[2]

optional = Optional.ofNullable(new AtomicInteger()); ❸
```

❶

Increment using counter directly

❷

Retrieve contained value and increment

❸

Optional reference can be re-assigned

You can modify the contained value either with the original reference, or one retrieved by calling `get` on the `Optional`. You can even re-assign the reference itself, which basically says that immutable is not the same thing as `final`. What you can't do is modify the `Optional` instance itself, because there are no methods available to do so.

This idea of the word "immutable" being something of a grey area is pretty common in Java, which doesn't have a good, built-in way of creating classes that only produce objects that can't be changed.

The static factory methods to create an `Optional` are `empty`, `of`, and `ofNullable`, whose signatures are:

```
static <T> Optional<T>  empty()
static <T> Optional<T>  of(T value)
static <T> Optional<T>  ofNullable(T value)
```

The `empty` method returns, naturally enough, an empty optional. The `of` method returns an optional that wraps the specified value or throws an exception if the argument is `null`. The expected way to use it is as shown in Example 6-2.

Example 6-2. Creating an Optional with "of"

```
public static <T> Optional<T> createOptionalTheHardWay(T value)
    return value == null ? Optional.empty() : Optional.of(value
}
```

The description of the method in Example 6-2 is called "TheHardWay" not because it's particularly difficult, but because there's an easier way, which is to use the `ofNullable` method, as in Example 6-3.

Example 6-3. Creating an Optional with “ofNullable”

```
public static <T> Optional<T> createOptionalTheEasyWay(T value)
    return Optional.ofNullable(value);
}
```

In fact, the implementation of `ofNullable` in the reference implementation of Java 8 is the line shown in `createOptionalTheHardWay`.

Incidentally, the classes `OptionalInt`, `OptionalLong`, and `OptionalDouble` wrap primitives which can never be null, so they only have an `of` method.

```
static OptionalInt    of(int value)
static OptionalLong   of(long value)
static OptionalDouble of(double value)
```

Instead of `get`, the getter methods on those classes are `getAsInt`, `getAsLong`, and `getAsDouble`.

See Also

Other recipes in this section, like "Mapping Optionals" and "Optional flatMap vs map", also create `Optional` values, but from provided collections. "Optional in Getters and Setters" uses the methods in this recipe to wrap provided values.

Retrieving Values From An `Optional`

Problem

You want to extract a contained value from an `Optional`.

Solution

Use the `get` method, but only if you're sure a value exists inside the `Optional`. Otherwise use one of the variations of `orElse`. You can also use `ifPresent` if you only want to execute a `Consumer` when a value is present.

Discussion

If you invoke a method that returns an `Optional`, you can retrieve the value contained inside by invoking the `get` method. If the `Optional` is empty, however, then the `get` method throws a `NoSuchElementException`.

Consider a method that returns the first even length string from a stream of them, Example 6-4.

Example 6-4. Retrieving the first even-length string

```
Optional<String> firstEven =
    Stream.of("five", "even", "length", "string", "values")
        .filter(s -> s.length() % 2 == 0)
        .findFirst();
```

The `findFirst` method returns an `Optional<String>`, because it's possible that none of the strings in the stream will pass the filter. You could print the returned value by calling `get` on the `Optional`.

```
System.out.println(firstEven).get()   // Please don't do this
```

The problem is that while this will work here, you should never call `get` on an `Optional` unless you're sure it contains a value or you risk throwing the exception, as in Example 6-5.

Example 6-5. Retrieving the first odd-length string

```
Optional<String> firstOdd =
    Stream.of("five", "even", "length", "string", "values")
        .filter(s -> s.length() % 2 != 0)
        .findFirst();

System.out.println(firstOdd.get()); // throws NoSuchElementExce
```

How do you get around this? You have several options. The first is to check that the `Optional` contains a value before retrieving it, Example 6-6.

Example 6-6. Retrieving the first even-length string with a protected `get`

```
Optional<String> firstEven =
    Optional<String> first =
      Stream.of("five", "even", "length", "string", "values")
        .filter(s -> s.length() % 2 == 0)
        .findFirst();

System.out.println(first.isPresent() ? first.get() : "No even l
```

While this works, you've only traded null checking for `isPresent` checking, which doesn't feel like much of an improvement.

Fortunately, there's a good alternative, which is to use the very convenient `orElse` method, Example 6-7.

Example 6-7. Using `orElse`

```
Optional<String> firstOdd =
    Stream.of("five", "even", "length", "string", "values")
        .filter(s -> s.length() % 2 != 0)
        .findFirst();

System.out.println(firstOdd.orElse("No odd length strings"));
```

The `orElse` method returns the contained value if one is present, or a supplied default otherwise. It's therefore a convenient method to use if you have a convenient fallback value in mind.

There are a few variations of `orElse`:

- `orElse(T other)` returns the value if present, otherwise it returns the default value, `other`
- `orElseGet(Supplier<? extends T> other)` returns the value if present, otherwise it invokes the `Supplier` and returns the result
- `orElseThrow(Supplier<? extends X> exceptionSupplier)` returns the value if present, otherwise throws the exception created by the `Supplier`

The difference between `orElse` and `orElseGet` is that the former returns a string that is always created, whether the value exists in the `Optional` or not, while the latter uses a `Supplier` which is only executed if the `Optional` is empty.

In this case, the value is a simple string, so the difference is pretty minimal. If, however, the argument to `orElse` is a complex object, the `orElseGet` method with a `Supplier` ensures the object is only created when needed.

```
Optional<ComplexObject> val =              // ...findFirst()

val.orElse(new ComplexObject());           // always creates the
val.orElseGet(() -> new ComplexObject()) // only creates object
```

Note

Using a `Supplier` as a method argument is an example of *deferred* or *lazy execution*. It allows you to avoid invoking the `get` method on the `Supplier` until necessary.[3]

Here is one more example to emphasize that point. Instead of returning a default string directly, Example 6-8 moves the evaluation of that string to a method call.

Example 6-8. Using `orElse` when element exists

```
private String getDefault() {
    System.out.println("inside getDefault()");
    return "No matching string found";
}

// ... then, inside main, ...

Optional<String> first =
    Stream.of("five", "even", "length", "string", "values")
        .filter(s -> s.length() % 2 == 0)
        .findFirst();

System.out.println(first.orElse(getDefault()));
```

The output of that example is:

```
inside getDefault()
five
```

In other words, the `getDefault()` method is invoked, even though the `Optional` contained a value. If all you're planning to do it returned a hard-wired default string (or some other simple type), the cost of evaluating the default may not be an issue. If you want to avoid it entirely, however, use the `orElseGet` method instead, as in Example 6-9.

Example 6-9. Using `orElseGet` to avoid evaluation of the default argument

```
private String getDefault() {
    System.out.println("inside getDefault()");
    return "No matching string found";
}

// ... same as before ...

System.out.println(first.orElseGet(() -> getDefault()));
// Equivalently, first.orElseGet(this::getDefault)
```

Now the output is just "five". The `Supplier` argument is evaluated, as all method arguments are, but its `get` method is not executed unless necessary.

The implementation of `orElseGet` in the library is Example 6-10.

Example 6-10. Implementation of `Optional.orElseGet` in the JDK

```
public T orElseGet(Supplier<? extends T> other) {
    return value != null ? value : other.get();
    // Note: "value" is a final attribute of type T in Optional
}
```

For the record, the `orElseThrow` method also takes a supplier. From the API, the method signature is:

```
<X extends Throwable> T orElseThrow(Supplier<? extends X> excep
```

Therefore, in Example 6-11, the constructor reference used as the `Supplier` argument isn't executed when the `Optional` contains a value.

Example 6-11. Using `orElseThrow` as a `Supplier`

```
Optional<String> first =
    Stream.of("five", "even", "length", "string", "values")
        .filter(s -> s.length() % 2 == 0)
        .findFirst();

System.out.println(first.orElseThrow(NoSuchElementException::ne
```

Finally, the `ifPresent` method allows you to provide a `Consumer` that is only executed when the `Optional` contains a value, as in Example 6-12.

Example 6-12. Using the `ifPresent` method

```
Optional<String> first =
    Stream.of("five", "even", "length", "string", "values")
        .filter(s -> s.length() % 2 == 0)
        .findFirst();

first.ifPresent(val -> System.out.println("Found an even-lengt

first = Stream.of("five", "even", "length", "string", "values")
    .filter(s -> s.length() % 2 != 0)
    .findFirst();

first.ifPresent(val -> System.out.println("Found an odd-length
```

In this case, only the message “Found an even-length string” will be printed.

See Also

Optional in Getters and Setters

Problem

You wish to use `Optional` in accessors and mutators.

Solution

Wrap the result of getter methods in optionals, but do not do the same for setters, and especially not for attributes.

Discussion

The `Optional` data type communicates to a user that the result of an operation may legitimately be null, without throwing a `NullPointerException`. The `Optional` class, however, was deliberately designed NOT to be serializable, so you don’t want to use it to wrap fields in a class.

Consequently, the preferred mechanism for adding optionals in getters and setters is to wrap nullable attributes in them when returned from getter methods, but not to do the same in setters, as in Example 6-13.

Example 6-13. Using Optional in a DAO layer

```
import java.util.Optional;

public class Department {
    private Manager boss;

    public Optional<Manager> getBoss() {
        return Optional.ofNullable(boss);
    }

    public void setBoss(Manager boss) {
        this.boss = boss;
    }
}
```

The `Manager` attribute is considered nullable[4]. You might be tempted to make the attribute of type `Optional<Manager>`, but because `Optional` is not serializable, neither is `Department`.

The approach here is not to require the user to wrap a value in an `Optional` in order to call a setter method, which is what would be required if the `setBoss` method took an `Optional<Manager>` as an argument. The purpose of an `Optional` is to indicate a value that may legitimately be null, and the client already knows whether or not the value is null, and the internal implementation here doesn’t care.

Finally, returning an `Optional<Manager>` in the getter method accomplishes the goal of telling the caller that the department may or may not have a boss at the moment and that's okay.

The downside to this approach is that for years the "JavaBeans" convention defines getters and setters in parallel, based on the attribute. In fact, the definition of a *property* in Java (as opposed to simply an attribute) is that you have getters and setters that follow the standard pattern. The approach in this recipe violates that pattern. The getter and the setter are no longer symmetrical.

It's (partly) for this reason that some developers say that `Optional` should not appear in your getters and setters at all. Instead, they treat it as an internal implementation detail that shouldn't be exposed to the client.

The approach used here is popular among open source developers who use Object-Relational Mapping (ORM) tools like Hibernate, however. The overriding consideration there is communicating to the client that you've got a nullable database column backing this particular field, without forcing the client to wrap a reference in the setter as well.

That seems a reasonable compromise, but, as they say, your mileage may vary.

See Also

“Mapping Optionals” uses this DAO example to convert a collection of IDs into a collection of employees. “Creating An Optional” discusses wrapping values in an `Optional`.

Optional flatMap vs map

Problem

You want to avoid wrapping an `Optional` inside another `Optional`.

Solution

Use the `flatMap` method in `Optional`.

Discussion

The `map` and `flatMap` methods in `Stream` are discussed in “Stream flatMap vs map”. The concept of `flatMap` is a general one, however, and can also be applied to `Optional`.

The signature of the `flatMap` method in `Optional` is:

```
<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)
```

This is similar to `map` from `Stream`, in that the `Function` argument is applied to each element and produces a single result, in this case of type `Optional<U>`. More specifically, if the argument `T` exists, `flatMap` applies the function to it and returns an optional. If the argument is not present, the method returns an empty optional. Like `flatMap` in `Stream`, however, the mapping does not wrap an existing `Optional` inside another `Optional`.

Consider a DAO (data access object) layer with getter methods as shown in Example 6-14.

Example 6-14. Part of a DAO layer with Optionals

```
public class Manager {
    private String name; ❶

    public String getName() {
        return name;
    }

    // ...
}

public class Department {
    private Manager boss; ❷

    public Optional<Manager> getBoss() {
        return Optional.ofNullable(boss); ❷
    }
```

```
    // ...
}
```

❶

Assumed not null

❷

Might be null, so wrap return in an `Optional`

This technique of using `Optional` in a DAO layer is discussed in "Optional in Getters and Setters", and the sample classes are (partially) repeated here for convenience. A `Manager` is assumed to have a (non-null, i.e., required) name. A `Department` may or may not have a `Manager`.

If the client calls the `getBoss` method on `Department`, the result is wrapped in an `Optional`. See Example 6-15.

Example 6-15. Returning an Optional

```
Manager mrSlate = new Manager("Mr. Slate");

// Department with a manager
Department d = new Department();
d.setBoss(mrSlate);
System.out.println("Boss: " + d.getBoss());
// prints: Boss: Optional[Manager{name='Mr. Slate'}]

// Department without a manager
Department d1 = new Department();
System.out.println("Boss: " + d1.getBoss());
// prints: Boss: Optional.empty
```

So far, so good. If the `Department` has a `Manager`, the getter method returns it, wrapped in an `Optional`, and if not, the method returns an empty optional.

The problem is, if you want the name of the `Manager`, you can't call `getName` on an `Optional`. You either have to get the contained value out of the `Optional`, or use the `map` method (Example 6-16).

Example 6-16. Extract a name from an Optional manager

```
// Can't call getName on an Optional, so either...
System.out.println("Name: " +
        d.getBoss().orElse(new Manager("Unknown")).getName());
// prints: Name: Mr. Slate

System.out.println("Name: " +
        d1.getBoss().orElse(new Manager("Unknown")).getName());
// prints: Name: Unknown

// or...
System.out.println("Name: " + d.getBoss().map(Manager::getName)
// prints: Name: Optional[Mr. Slate]

System.out.println("Name: " + d1.getBoss().map(Manager::getName
// prints: Name: Optional.empty
```

The `map` method (discussed further in “Mapping Optionals”) applies the given function only if the `Optional` it’s called on is not empty, so that’s the simpler approach here.

Life gets more complicated if the optionals might be chained. Say a `Company` might have a `Department` (only one, just to keep the code simple), as in Example 6-17.

Example 6-17. A company may have a department (only one, for simplicity)

```
public class Company {
    private Department department;

    public Optional<Department> getDepartment() {
        return Optional.ofNullable(department);
    }

    // ...
}
```

If you call `getDepartment` on a `Company`, the result is wrapped in an `Optional`. If you then want the manager, the solution would appear to be to use the `map` method as above. But that leads to a problem, because the result is an optional wrapped inside an optional (Example 6-18).

Example 6-18. An Optional wrapped inside an Optional

```
Company co = new Company();
co.setDepartment(d);

System.out.println("Company Dept: " + co.getDepartment());
// prints: Company Dept: Optional[Department{boss=Manager{name=

System.out.println("Company Dept Manager: " + co.getDepartment
    .map(Department::getBoss));
// prints: Company Dept Manager: Optional[Optional[Manager{name
```

This is where `flatMap` comes in. Using `flatMap` flattens the structure, so that you only get a single optional. See Example 6-19.

Example 6-19. Using flatMap

```
System.out.println(
    co.getDepartment()
        .flatMap(Department::getBoss)
        .map(Manager::getName));
// prints: Optional[Mr. Slate]

Optional<Company> company = Optional.of(co);
System.out.println(
    company
        .flatMap(Company::getDepartment)
        .flatMap(Department::getBoss)
        .map(Manager::getName)
);
// prints: Optional[Mr. Slate]
```

As the example shows, you can even wrap the company in an optional, then just use `flatMap` repeatedly to get to whatever property you want, finishing with a `map` operation.

See Also

Wrapping a value inside an `Optional` is discussed in "Creating An Optional". The `flatMap` method in `Stream` is discussed in "Stream flatMap vs map". Using `Optional` in a DAO layer is in "Optional in Getters and Setters". The `map` method in `Optional` is in "Mapping Optionals".

Mapping Optionals

Problem

You want to apply a function to a collection of `Optional` instances, but only if they contain a value.

Solution

Use the `map` method in `Optional`.

Discussion

Say you have a list of employee ID values and you want to retrieve a collection of the corresponding employee instances. If the `findEmployeeById` method has the signature

```
public Optional<Employee> findEmployeeById(int id)
```

Then searching for all the employees will return a collection of `Optional` instances, some of which may be empty. This is actually a natural usage of both `map` and `flatMap`, as shown in Example 6-20.

Example 6-20. Finding Employees by ID

```
public List<Employee> findEmployeesByIds(List<Integer> ids) {
    return ids.stream()
        .map(this::findEmployeeById) // returns Stream<Optiona
        .flatMap(optional -> optional.isPresent() ?   ❶
            Stream.of(optional.get()) :
            Stream.empty())
        .collect(Collectors.toList());
}
```

❶

Check `isPresent` before invoking `get`

The result of the `map` operation is a stream of optionals, each of which either contains an employee or is empty. To extract the contained value, the natural idea is to invoke the `get` method, but you're never supposed to call `get` unless you're sure a value is present. The ternary operator in the listing does exactly that.

The result is a stream of streams, where any optionals containing employees are now single-element streams containing employees, and any empty optionals are now empty streams[5]. The `flatMap` operation then flattens the stream of streams down to a single stream, which can be collected into a list.

That was fun, but the code can be simplified using the `Optional.map` method, whose signature is:

```
<U> Optional<U> map(Function<? super T,? extends U> mapper)
```

The `map` method in `Optional` takes a `Function` as an argument, and returns either an `Optional` with the result or an empty `Optional`.

Now the code in Example 6-20 can be simplified (slightly) to the version in Example 6-21.

Example 6-21. Using Optional.map

```
public List<Employee> findEmployeesByIds(List<Integer> ids) {
    return ids.stream()
        .map(this::findEmployeeById) // returns Stream<Optional
        .flatMap(optional -> optional.map(Stream::of).orElseGet
        .collect(Collectors.toList());
}
```

Each `Optional<Employee>` is transformed into a `Stream<Employee>`, and each empty optional is transformed into an empty stream. The subsequent `flatMap` operation flattens the resulting stream of streams into a single stream.

The process is illustrated in Figure 6-1.

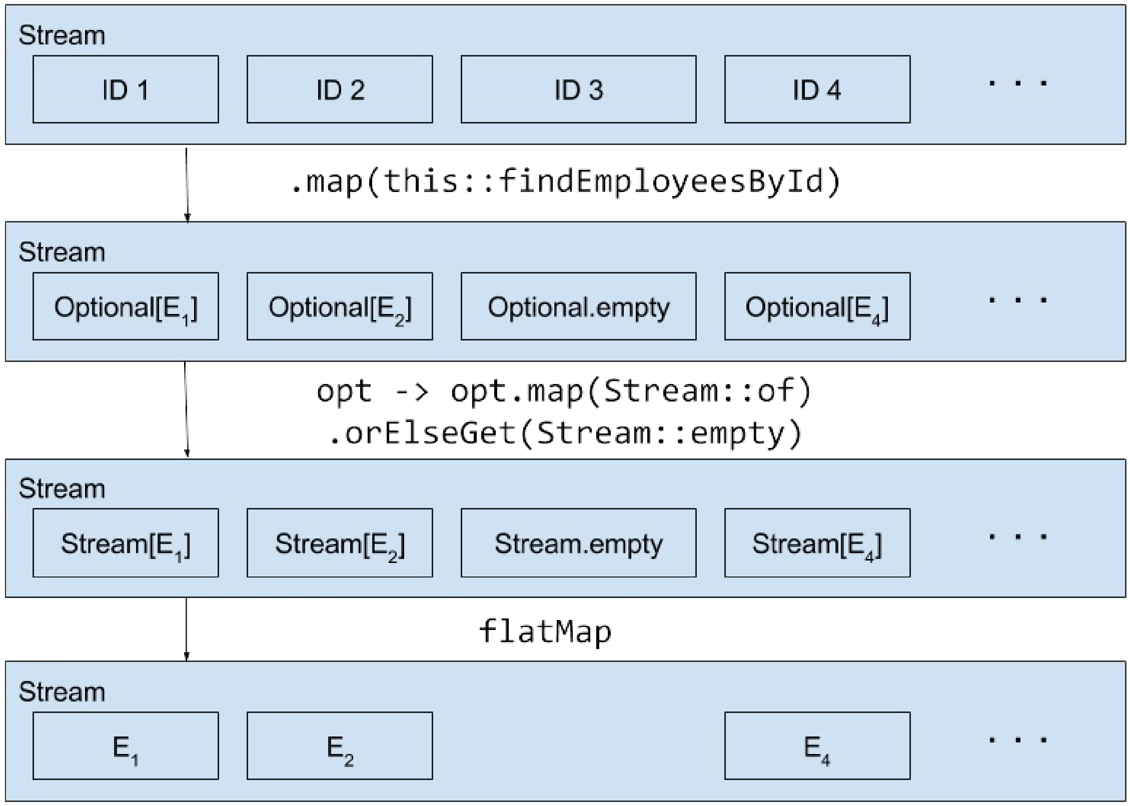


Figure 6-1. Optional map and flatMap

The `Optional.map` method is a convenience method for (hopefully) simplifying stream processing code.

Incidentally, another way of solving the same problem is to simply `filter` on `isPresent`, as in Example 6-22.

Example 6-22. Filtering with Optional.isPresent

```
public List<Employee> findEmployeesByIds(List<Integer> ids) {
    return ids.stream()
        .map(this::findEmployeeById) // returns Stream<Optiona
        .filter(Optional::isPresent)
        .map(Optional::get)
        .collect(Collectors.toList());
}
```

The `filter` produces a new stream with only the `Optional` instances that contain an employee. Then the `map` operation converts those to employees. Arguably this is more intuitive, especially for developers unaccustomed to `flatMap` operations, but the result is the same.

See Also

“Optional in Getters and Setters” illstrates how to use `Optional` in a DAO (data access object) layer. “Stream flatMap vs map” discusses the `flatMap` method.

[1] I’m being diplomatic here.

[2] See the sidebar about immutability

[3] See chapter 6 of Venkat Subramaniam’s book *Functional Programming in Java* (Pragmatic Programmers, 2014) for a detailed explanation.

[4] Perhaps this is just wishful thinking, but an appealing idea, nonetheless.

[5] Insert your own *Inception* joke here