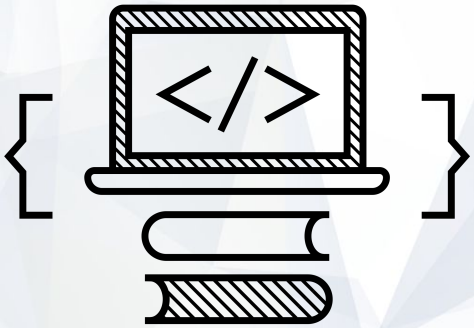
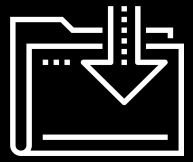


Data Structures Part 2

Java Accelerator 7

Lesson 1.5



© 2022 Trilogy Education Services, a 2U, Inc. brand. All Rights Reserved.

Learning Objectives

01 Capture and gracefully handle application errors without crashing the program.

02 Use streams and lambdas to process data.

03 Load data from the file system.

Topics

In this lesson we will cover:

Exceptions and exception handling

Lambdas and streams

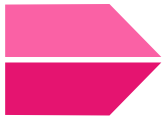
Loading data from the file system

About Exceptions

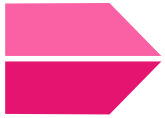
An exceptional event which occurs in the normal execution of a program that disrupts the normal flow of the program's instructions.

—Oracle

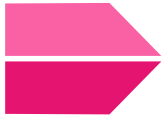
About Exceptions



Errors occur in all programs.



Every programming language has a way to handle errors.



Exceptions are the approach that Java takes.

Handling Exceptions

There are two ways to address code that might throw an exception:

We can handle them; we call this “catching the exception.”

We can pass the buck by marking that our code might throw an exception, leaving it to other code to catch the exception.

Types of Exceptions

There are two main categories of exceptions in Java:

Checked

We are required to either catch these or specify that our code might throw one of these. The compiler will enforce this requirement.

Unchecked

We do not have to catch or specify these, but we can catch them if we want to.

Unchecked exceptions are generally errors that we can't or don't want to recover from, but that is certainly not always the case (as we'll see later in this lesson).

Try/Catch/Finally

The language construct used to handle exceptions is try/catch/finally.
It looks like this:

```
try {
  // some code that might throw and exception
} catch (an-exception-type identifier) {
  // some code to handle or recover from the exception
} finally {
  // some code that runs whether there was an exception or not
 }
```

Instructor Demonstration

Exception Examples

Instructor Demonstration

Unchecked Exceptions

Activity: Unchecked Exceptions

Suggested Time:

Instructor Demonstration

Testing Exceptions

Activity: Testing Exceptions ArrayFun

Suggested Time:

Activity: Exception Exercise

Suggested Time:

Streams and Aggregate Operations

Aggregate Operations

Aggregate operations come in two types:

Intermediate Operations

Accept a stream and produce a stream.

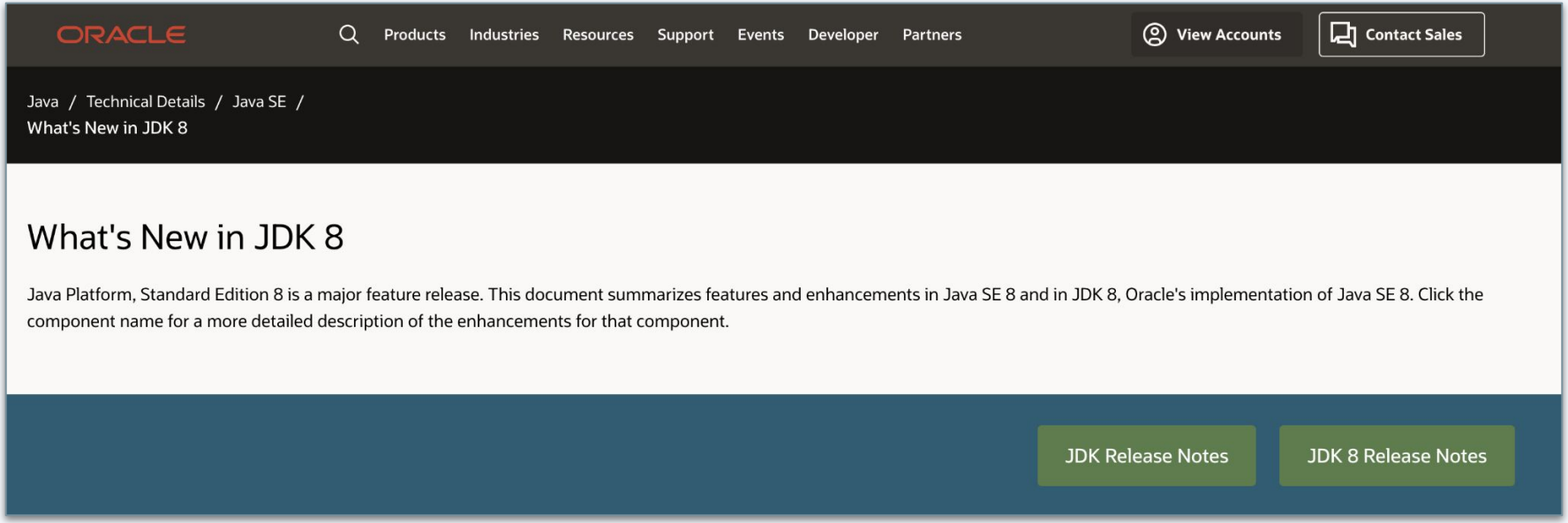
Some intermediate operations accept a stream of one type and produce a stream of another type (for example, a stream of motorcycles might be converted into a stream of ints that are the displacements for each motorcycle).

Terminal Operations

Accept a stream and produce a non-stream result (which could be null).

Aggregate Operations

Lambdas got a lot of the attention when the features of Java 8 were announced because they introduced **functional programming** to Java.



oracle.com

The real power for everyday tasks is when streams and aggregate operations are used with lambdas.

Streams and Aggregate Operations Used with Lambdas

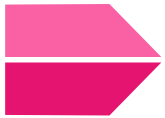
This combination gives developers some great options for processing collections of data. For example:

- Get all the motorcycles made by a particular manufacturer.
- Get all the motorcycles with power under 100 horsepower.
- Get all the motorcycles made by a particular manufacturer under 100 horsepower.
- Calculate the average displacement of the motorcycles in the collection.
- Return the value of the highest horsepower in the collection.
- Group the motorcycles by manufacturer and return them in a map where the key is the name of the manufacturer and the value is a list of motorcycles from that manufacturer.

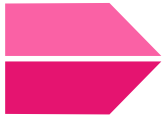
A **pipeline** is a sequence of aggregate operations. Pipelines allow for more complex processing of a group of objects in a collection.

Pipelines

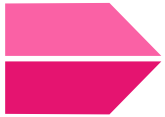
A pipeline consists of:



A source of data (a collection, for example)



Zero or more intermediate aggregate operations



One terminal operation

Iterators and Loops vs. Stream Processing

Processing objects in a collection using loops/iterators and processing them using streams and aggregate operations have a lot in common, but there are some key differences:

Iterators and Loops		Stream Processing
Loops/iterators work directly on objects from the collection	vs.	Aggregate operations work on objects from a stream
The developer must be in control of how and when objects in the collection are processed when using a loop or an iterator.	vs.	All of the iteration and control of how and when objects are acted upon are internal to the operation.

The parameters for aggregate operations are lambda expressions.

Lambdas and Aggregate Operations

So far, all of our parameters to methods have been data of some kind: either primitives or objects.

Lambdas allow us to pass in methods as parameters, which is a really powerful feature that allows us to pass in or define functions on the fly.

This allows us to easily create code to process objects in collections. Each aggregate operation takes a lambda expression as a parameter.

Aggregate Operations

`filter`	An intermediate operation that filters the object in a stream based on the logic in the given lambda expression.
`forEach`	A terminal operation that acts on each object in a stream according to the logic in the given lambda expression.
`collect`	A terminal operation that returns a collection of objects according to its parameter.

Aggregate Operations

`average`	A terminal operation that returns the average of the given input stream. The average is returned as an `OptionalDouble`.

`mapToInt`
`mapToDouble`
`mapToLong`
	A family of intermediate operations that map an input stream of one type into a stream of another type (integer, double, or long) based on the logic in the given lambda expression. For example, `mapToInt` would map an incoming stream of objects to a stream of integers.

`getAsInt`
`getasDouble`
`getAsLong`
	`getAsInt` is a method of `OptionalInt` that returns an integer from an `OptionalInt`. In the same vein, `getAsDouble` returns a double from an `OptionalDouble`; `getAsLong` returns a long from an `OptionalLong`.

Lambda Syntax

Traditional Method Declaration

```
boolean filterByMake(Motorcycle moto) {
  return moto.getMake().equals("Suzuki");
}
```

Verbose Lambda Declaration

This format can be used no matter what.

We can take some more shortcuts under certain circumstances.

```
(Motorcycle moto) -> {
  return moto.getMake().equals("Suzuki");
}
```

More Concise Lambda Declaration

- If we have only one parameter, we can lose the parentheses.
- We can also lose the parameter type—the compiler will infer the type.
- We can take further shortcuts under certain conditions.

```
moto -> {
  return moto.getMake().equals("Suzuki");
}
```

Even More Concise Lambda Declaration

- This particular method declaration works with the `filter` aggregate operation (which, as you can infer, expects a boolean return).
- In this case, we don't have to use the `return` keyword. The compiler will infer that the last statement in the method should be returned.
- If there is only one statement in the body of the method, we don't need to use curly braces.
- Developers generally try to use the most concise form they can.
- We'll see several examples of different formats below.

```
moto -> moto.getMake().equals("Suzuki")
```

Time to Code

Lambdas and Streams

Suggested Time:

30 minutes

Activity: Lambda Stream Exercise

Suggested Time:


Recap

Recap: Exceptions

You should now be able to do the following:

- Define exception.
- Compare and contrast checked and unchecked exceptions.
- Explain the “Catch or Specify” requirement in Java.
- Write code that satisfies the “Catch or Specify” requirement in Java.
- Explain the `finally` block.
- Use a `finally` block in a program.
- Write code that catches and handles exceptions.

Lambdas and Streams

You should now be able to do the following:

- Define lambdas.
- Describe the concise syntax of a lambda function.
- Explain the aggregate operations for collections.
- Compare and contrast stream processing and iterators.

Use the following aggregate operations (with lambda expressions) in a program:

`filter`	`forEach`	`collect`
`average`	`mapToXxx`	`getAsXxx`