Intermediate Java

WELCOME!

Develop Intelligence

Simon Roberts

WELCOME

Develop Intelligence

Mark Your Attendance:

- Go to the Course Calendar Invite
- Click on the Course Event Link
- Press Check-In

Join Us in Making Learning Technology Easier

Develop Intelligence

Our mission...

Over 16 years ago, we embarked on a journey to improve the world by making learning technology easy and accessible to everyone.

...impacts everyone daily.

And it's working. Today, we're known for delivering customized tech learning programs that drive innovation and transform organizations.

In fact, when you talk on the phone, watch a movie, connect with friends on social media, drive a car, fly on a plane, shop online, and order a latte with your mobile app, you are experiencing the impact of our solutions.

Upskilling and Reskilling Offerings

Develop Intelligence

Intimately customized learning experiences just for your teams.

- **Workshop** — 2-3 day upskilling experiences
- **Fast Track** — 5-day reskilling experiences
- **Learning Spike** — 1-day technology overviews
- **Target Topics** — 90-minute instructor-led micro-learnings
- **Hack-a-thon** — Learn and build an MVP in 2-3 days

World Class Practitioners

Note About Virtual Trainings

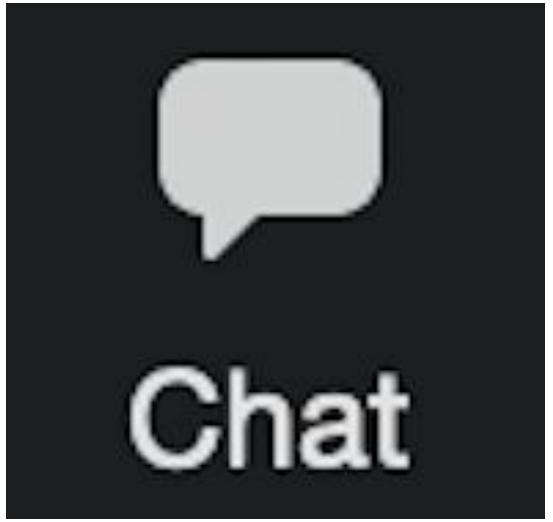
What we want

...what we've got

Virtual Training Expectations for You

Arrive on time / return on time

Mute unless speaking



Use chat or ask questions verbally

Virtual Training Expectations for Me

I pledge to:

- Make this as interesting and interactive as possible
- Ask questions in order to stimulate discussion
- Use whatever resources I have at hand to explain the material
- Try my best to manage verbal responses so that everyone who wants to speak can do so
- Use an on-screen timer for breaks so you know when to be back

Prerequisites

- You should have written code in Java and be familiar with the majority of syntax that exists in Java 7.

- This course will introduce some of the features that are less well understood from Java 7 onwards.

Objectives

At the end of this course you will be able to:

- Use try with resources and multi-catch constructs to handle exceptions.
- Use inner and nested classes to implement appropriate design patterns.
- Make good use of collections and streams apis.
- Use the java.time api.
- Create generic classes and methods
- Use Java's functional features including writing and using lambda expressions.
- Create annotations and write code that identifies and uses annotations at runtime.
- Use key features of the java.util.concurrent apis.

About you

In 90 seconds!

- What you hope to learn
- What your background level is

Agenda

- Investigating the try-with-resources mechanism
- Reviewing Java syntax, design patterns, and using inner classes.
- Investigating collections and introducing functional ideas
- Building more functional syntax
- Using streams
- Building generic types and methods
- Working with annotations
- Using the concurrency api

THANK YOU

Develop
Intelligence


Develop
Intelligence

Exception handling in Java 7

Develop Intelligence

Java 7 added:

- Try-with-resources
- multi-catch

Try-with-resources

Develop Intelligence

Prior to this feature Java's try/finally had significant issues when trying to ensure reliable closure of O/S resources (such as files).

- Variable scope rules made access to the resource in the finally block cumbersome
- The definite initialization rule requires null-checking every resource
- Many close() methods throw a checked exception, mandating try/catch in the finally block
- Perhaps worse only one live exception is possible per-thread in the JVM, which means significant exceptions can be lost if a close throws an exception.

Try-with-resources

Develop Intelligence

Java 7 try-with-resources addresses all of these problems:

```
try ( // declare+initialize AutoCloseable resources
  BufferedReader in =
    Files.newBufferedReader(Path.of("input.txt"));
  BufferedWriter out = Files.newBufferedWriter(
    Path.of("out.txt", StandardOpenOption.CREATE));
) {
  // regular try body.
} // No "finally" needed, resources automatically closed
```

Try-with-resources

- Multiple resources can be listed in the resources block
- Separate with semicolon
- "Terminating" semicolon permitted but not required
- Since Java 9, a final variable that is declared above the try may be listed, allowing greater scope, but still guaranteeing closure

Design Concepts

Develop Intelligence

If a change from design A to design B is easy, and both satisfy the functional requirement, but a change from B to A is relatively hard, prefer design A unless there are other reasons to choose between them

- Make classes final until there is a determined need for inheritance, and the consequences are understood
- Prefer factories or builders over constructors
  - Constructors can only produce a new object of the exact type, or an exception
  - Any method can do this, but can do much more too
  - Removing constructors breaks clients, changing the implementation details of a method does not.

Design concepts

Develop Intelligence

Consider preferring immutable data and structures

- This can avoid "that can't happen" moments where other users of data altered them unexpectedly
- Immutable data can be shared, perhaps reducing the need for copying and compensating for the need to make new data to describe variations on old data
- Sometimes an immutable proxy (such as Collections.unmodifiableXxx) serves well to prevent clients changing something while allowing the "owner" to change it.

Design concepts

Develop Intelligence

Ensure that a business domain object is always "valid"

- Avoids unexpected states breaking client code
- Ensure any/all mutable fields are private
- Ensure that validity is verified prior to completing construction and before all changes
- Represent adjunct concepts (such as wire transfer and database storage) in secondary classes (e.g. a Data Transfer Object). Ensure the secondary class is dependent on the primary but not the other way round.

Design concepts

Develop Intelligence

Keep in mind that a Set is intended to reject duplicates, and might be preferable to a List in some situations. However:

- Sets reject duplicates simply by returning false from their add methods. This is easy to overlook.
- Sets require objects that implement equals/hashcode or ordering, and it's not always clear which is required if you simply have a java.util.Set.
- Most (but not all) core Java objects do not implement equals/hashcode unless they're immutable (String, primitive wrappers, java.time classes, for example)

Separation of concerns

Develop Intelligence

Take care to separate unrelated concerns, and also concerns that change independently, for example:

To print a subset of a list involves 3 independent concerns:

- processing the list
- distinguishing which items are wanted
- printing the result

Command pattern

Develop Intelligence

Defining the selection mechanism is done (in object oriented languages) by defining a behavior (method) in an object, and passing that object as an argument for the purpose of the behavior it contains

Passing the behavior as in argument object is known (in the Gang of Four pattern catalog) as the "command" pattern

The concept is also also known as a higher order function in "functional programming"

Lambda expressions

Develop Intelligence

A lambda expression in Java defines an object in a context.

- The context must require an implementation of an interface
- The interface must declare EXACTLY ONE abstract method
- We must only want to implement that one abstract method
- We provide a modified method argument list and body with an "arrow" between them:

```
(Student s) -> {
  // function body
}
```

- The argument list and return type must conform to the abstract method's signature

Lambda syntax variations

- Argument types can be omitted if they're unambiguous
  - This is "all or nothing"
- Since Java 10, argument types can be replaced with "var" if they're unambiguous
  - This is also "all or nothing"
- If a single argument carries zero type information the parentheses can be omitted
- If the method body consists of a single return statement, the entire body can be replaced with the expression that is to be returned.

FunctionalInterface annotation

If an interface is intended for use with lambdas, it must define exactly one abstract method.

The @FunctionalInterface annotation asks the compiler to verify this and create an error if this is not the case.

Predefined interfaces

Develop Intelligence

Due to Java's strong static typing, and the restrictions preventing generics being used with primitives, different interfaces must be provided for a variety of different situations.

Key interface categories are:

Function - takes argument, produces result

Supplier - zero argument, produces result

Consumer - takes argument, returns void

Predicate - takes argument, returns boolean

Unary/Binary Operator - Function variants that lock args and returns to identical type

Predefined interface variations

For two arguments, expect a "Bi" prefix

For primitives:

- Int, Long, Double prefix usually means "primitive argument"
  - Note for Supplier, this prefix refers to return type (there are zero arguments.)
- ToInt, ToLong, ToDouble prefix means "primitive return"

Method References

Develop Intelligence

Four forms of lambda can be replaced with an alternate syntax called a method reference:

`(a, b, c) -> a.doStuff(b, c)` [for a of type MyClass, and any number of arguments b, c, but at least the argument a] can be replaced with:

```
MyClass::doStuff
```

`(a, b, c) -> MyClass.doStuff(a, b, c)` for any number of arguments can be replaced with:

```
MyClass::doStuff
```

If more than one method exists that would map correctly, the compilation fails

Method References

Develop Intelligence

`(a, b, c) -> new MyClass(a, b, c)` for any number of arguments can be replaced with:

```
MyClass::new
```

`(a, b, c) -> <obj.expr>.doStuff(a, b, c)` for any number of arguments can be replaced with:

```
<obj.expr>::doStuff
```

Method References

Develop Intelligence

Method references cannot be used unless the arguments of the lambda and the arguments of the target function:

- are in the correct order
- can be used without modification

Method references might sometimes have more than a single target method that would match the translation. In this case, the method reference form cannot be used.

Method references should be used to focus the reader on the functionality to be used, and when the arguments aren't considered important.

The Monad (and Functor) patterns

Develop Intelligence

A data structure that has an operation traditionally called "map", which applies a caller-provided function to each of its contained elements and produce a new data structure of the same overall type, containing the same number of elements (but perhaps of a different type) which are the results of the applied function is often called a "Functor"

A related structure that has an operation traditionally called "flatMap", which applies a caller-provided function to each argument, but where that caller-provided function can itself return another instance of the overall structure, is often called a Monad.

The Monad (and Functor) patterns

Develop Intelligence

Monads can do everything that a Functor can do

The combination of filter, map, and flatMap can create powerful and expressive code that focuses on the operations to be applied to data, rather than being cluttered with the "how" of applying those operations.

In fact, the internal implementation can change (potentially using a lazy approach or a multi-threaded approach) with no difference in how the client code is written.

The approach also works best without mutating any data (and thus is particularly suited to immutable objects).

Stream API

Develop Intelligence

Stream is one of about three "Monad-like" APIs in Java.

It provides filter, map, and flatMap and many more operations

It differs from the "SuperIterable" example in two critical implementation ways:

- Stream is "lazy", items are "pulled" down the pipeline one at a time, so it does not create a complete set of intermediate results at any point
- Stream can "shard" the data across multiple threads, allowing for increased throughput utilizing multiple physical CPUs (but unlike some systems, e.g. Apache Spark, it does not shard across physically distinct, networked, machines.)

Terminal operations

Develop Intelligence

Once individual items have been selected and processed, it might be necessary to produce "a single result".

The result could be:

- a printout of the results
- a data structure containing the items
- a single data structure representing some aggregate result computed from all the intermediate results

The final step is called the "terminal operation"

In the Stream API, no processing is performed until the terminal operation "pulls" data through the pipeline (this is the lazy aspect).

Terminal operations

The Stream API provides several built in, and several supporting terminal operations including:

- forEach
- allMatch, anyMatch, noneMatch
- three overloaded reduce methods
- two overloaded collect methods
- and a variety of implementations of the Collector interface that perform operations such as building List, Set, and Map structures, and possibly post-processing the items selected into a map

Reduction and collection operations

Develop Intelligence

The most general form of terminal operation might be "reduce".

- This operation must be provided with a BinaryOperator that accepts two of the data type in the stream and produces a new one representing the result of combining those two.
- This is applied repetitively to produce a single result
- The most basic form of reduce produces an Optional<T> result which is a monad-like container of zero or one element (avoiding the use of null)

Java's reduce methods

Develop Intelligence

In Java, there are 3 overloaded reduce operations

- Each requires a binary operator of the stream data type
- One returns an Optional if the stream is empty, the other two require an "identity" object (of the stream type), and if the stream is empty they return that value
- One allows a result type (and requires and identity of the result type) that is different from the stream type. This version requires a BiFunction that takes the result type and the stream type and creates a new result. This version also requires a BinaryOperator of the result type to perform a final merging process if the stream is running parallel.
- In all reduce methods, a new intermediate object must be created at every step. This can sometimes adversely impact performance.

Collect operations

Develop Intelligence

To mitigate the potential performance impact of creating many intermediate objects, Java provides a variant reduction operation called "collect"

- A collect mutates a single (per thread) result object, rather than creating a new one every time
- The collect operation therefore needs to be able to create its own result objects
- It also needs an operation that mutates the result object with each stream item
- It needs a way to add the contents of one result object from one thread to another result object in the case of parallel execution

Collect arguments

The three arguments to a collect operation are:

- Supplier<ResultType>
- BiConsumer<ResultType, StreamType>
- BiConsumer<ResultType, ResultType>

Note that the "output" result is the first argument in these latter two cases. Avoid confusing these in the third argument, or data will be ignored.

The Collector interface

Develop Intelligence

The Collector interface allows constructing a class that encapsulates the three pieces of functionality used in a collect operation.

- It also adds a "finisher" which can convert from the result of the main collection to a different type (for example, converting an Average to a double.
- It also changes the combiner that takes intermediate "per-stream" results and uses a BinaryOperator model, avoiding the chance of passing arguments in the wrong order.
- In addition, it provides "characteristics" which can address whether the finisher should be used, whether the collection considers item order to be significant, and whether the collection uses a threadsafe result object.

Prebuilt collectors

Develop Intelligence

The Collectors class contains factory methods for a variety of useful collection operations.

Perhaps the most useful are the various overloads of groupingBy

- These collectors are given a function--called a classifier--that creates a key representing the stream item
- The key is the key in a map structure
- The value, in the simplest form of groupingBy is a list of all the stream items that produced that key.
- Overloaded variants of groupingBy support a "downstream" collector which can perform stream-like operations on each item that matches the key.
- Using these forms, the items can be mapped, flatMapped, filtered, and collected into structures other than a list, or reduced to other values, such as the count

Additional Stream utilities

- Files.lines can create a Stream directly from a file
- Streams implement AutoCloseable to facilitate proper closing when there is a file as backing store
- Pattern (a regular expression) can split a line of text directly to a Stream
- Other Files utilities can give streams from the file system, such as a stream of Path names