Functional Programming for Java Developers

Develop Intelligence

Simon Roberts

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

BACK END DEVELOPMENT

BIG DATA

CLOUD COMPUTING

DEVOPS

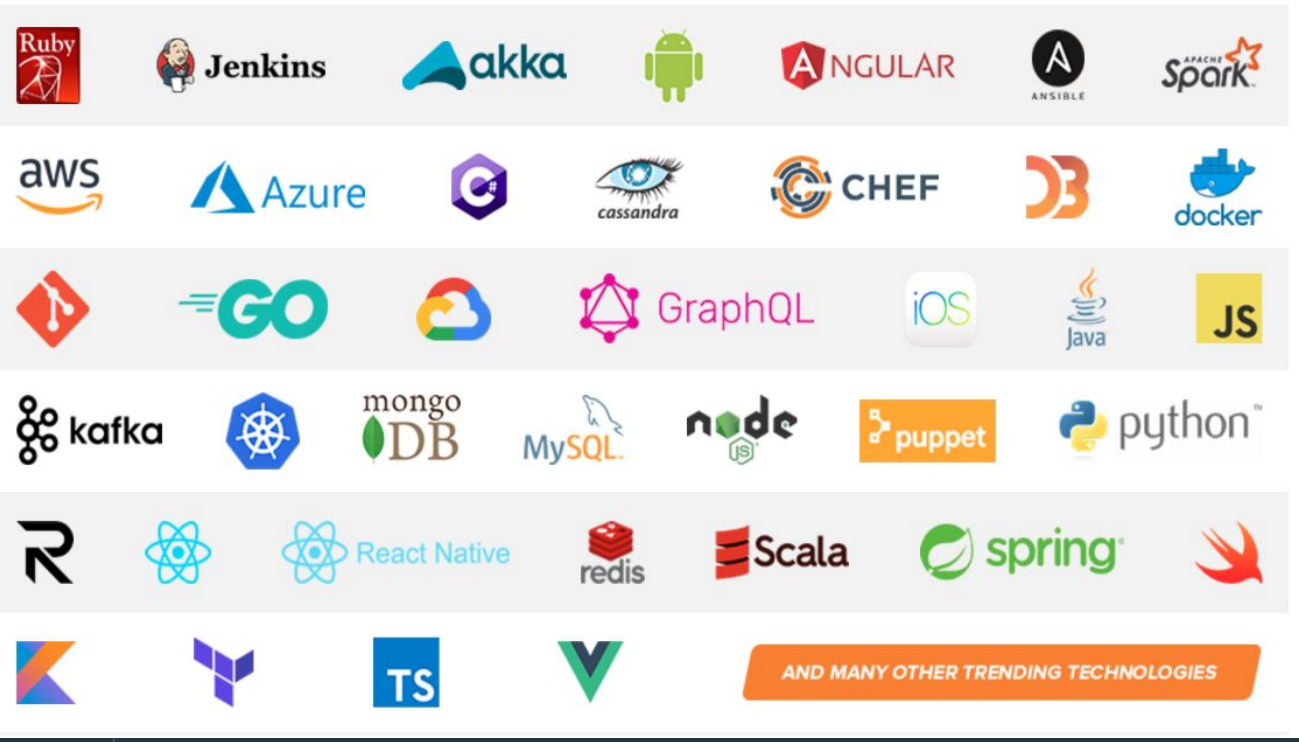
FRONT END DEVELOPMENT

MACHINE LEARNING

MOBILE APP DEVELOPMENT

SOFTWARE ENGINEERING

SYSTEM ADMINISTRATION



World Class Practitioners

Note About Virtual Trainings

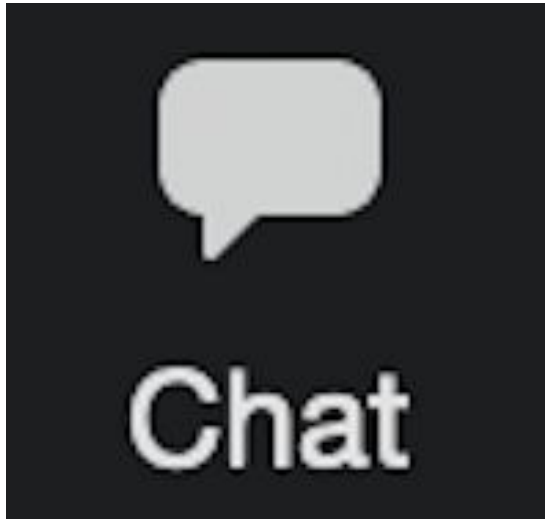
What we want

...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

Develop Intelligence

- "Two years of professional coding with Java"

- You are comfortable designing with, and coding in Java:
  - Classes with fields, methods, and constructors
  - Interfaces declaring abstract methods
  - Classes that extend other classes, and implement interface, and which properly initialize their parent features
  - Checked and unchecked exceptions
  - Try-with-resources structure
  - Iterable, Collection, List, Set, and Map
  - Using basic generics to enable type-checking with collections

Objectives

Develop Intelligence

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

- Compare and contrast functional and imperative programming styles
- Explain how side-effect-free functions reduce errors and are easier to test
- Employ immutable data and understand how immutable data structures can be used in a memory-efficient way
- Formulate higher-order functions and use syntax that simplifies these constructs
- Use generics and co- and contra-variance to create more reusable code
- Design with and use monads and functors
- Use Java's monadic libraries for parallel and asynchronous data processing

Agenda

Develop Intelligence

- Introducing the Pure Function Concept
- Immutability in the Real World
- Solving a Simple Design Problem Using Imperative and Functional Styles
- Java's Lambda Syntax
- Behavior Factories
- The Importance of Generics in Functional Programming
- Introduction to the Monad Concept
- Introduction to Producing a Final Result
- More Advanced and Parallel Production of a Result
- Overview of Monad-like Libraries and Frameworks
- Handling Errors in Functional Programming

Introducing the Pure Function Concept

- What are “side effects”?
- What are “observable side effects”?
- Why we care: composability of program elements and maintenance
- Avoiding mocking: Testing a pure function
- Refactoring our thinking and designs for pure functions

Pure Functions and Side Effects

Develop Intelligence

- Any change resulting from function invocation, other than that returned value of that function, is considered a *side effect*

- A *pure function* takes arguments and returns a result:
  - There should be no side effects
  - The result should depend only on the arguments--even if invoked multiple times. (idempotency/determinism)
  - It produces a result for all inputs (totality)

Pure Functions and Side Effects

Develop Intelligence

- In practice, any computation performed on a practical machine will have *some* side effects:
  - At a minimum, writing to some memory or CPU register
- An *observable side effect* is one that could affect the outcome of some later computation
  - So, printing/logging is generally considered not observable
  - But might be if that output is seen by the user
  - So, sometimes, observability might be a grey area
- Practical programming is generally concerned with observable side effects
  - So, mutating *local* variables inside a function is unlikely to be a concern

Side Effects--Why We Care

Develop Intelligence

- If a function causes visible side effects this will make it harder to use, reuse, and test. For example
  - If an effect goes places other than returned value, composing the function with others that perform additional computations on the result is much harder, or perhaps impossible
  - If an effect is stored "downstream" (e.g. writing to a database) then testing will require mocking, by contrast a pure function does not need mocking and is easier to test

Practical Computing with Pure Functions

Develop Intelligence

- In practice, if a program is 100% pure, it cannot create any result other than returning a value, and it cannot get any input other than values on the command line (it cannot take filenames, or database connection information).
- This is clearly impractical
- A practical approach is to separate a pure function that performs business logic, from an impure function that performs some kind of IO
  - The impure function now performs only one operation and is easier to test due to simplicity
  - The pure function is now easier to test, and to reuse / compose

Immutability in the Real World

Develop Intelligence

- If two parts of a software system share mutable data, there is always a chance that one part will make changes that the other part did not expect
  - The result is a bug
  - That bug can be hard to find and hard to fix
  - A common solution is to take "defensive copies", just in case the data are changed
- If all data/structures are immutable after initial creation, then no copy need be taken unless some part of the system wishes to make a change.
  - This can sometimes be more memory efficient, rather than less
  - In general, some uses will benefit, others will suffer
- Java does not readily support immutable data, but careful design can get most of the benefits in parts of the overall system, either for new code, or refactored code

Unresolved and resolved values

Develop Intelligence

- Simple local variables can often benefit from being immutable too
  - Instead of keeping a balance variable and adding a deposit amount to it, consider two variables: `balanceBefore`, and `balanceAfter`
  - Often mutating a variable actually means it now represents "something else" and a new variable name better describes it

- Functional programmers often use expressions without storing them in variables
  - If the expression is simple (e.g. a well-named method invocation) this need not be any less readable

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
```

More on Immutable Data

Develop Intelligence

- Warning: Initialization is mutation
  - Junior/new functional programmers sometimes believe that immutable data are 100% thread safe
  - This is not reliable: initializing a variable in a physical computing machine is mutation, and this mutation can cause concurrency problems
  - Nevertheless, the scope for concurrency problems is hugely reduced by using immutable data

Overview of recursion

Develop Intelligence

- Controlling a loop often requires a changing variable
  - By contrast, iteration through recursion creates a new set of variables in a new scope for every iteration
  - This is preferred by traditional functional practitioners
- Unfortunately, unbounded recursion will cause stack overflows in the JVM
  - Other languages might have optimizations (e.g. "tail recursion optimization") but Java does not provide this
- If the loop-control variables are entirely local, the side effect is not visible, and so is likely not a practical problem

Overview of immutable data structures

Develop Intelligence

- Immutable data structures typically give the impression of being wasteful:
  - Every change seems to require a new structure
  - However, if data are immutable tail parts of lists, and whole tree branches, can be reused if the structure is appropriately designed.
- The classic immutable data structure for functional programming is the singly linked list
  - Such a structure can have any number of "changes" made at the head, while allowing multiple starting points to share the same tail-end data.
  - However, any change other than at the head requires all the structure to the left of that change to be duplicated
- Uses of immutable data structures must be made with awareness of how to gain the maximum reuse, or devastating inefficiencies can result

Solving a Simple Design Problem Using Imperative and Functional Styles

- In any programming styles, smart design separates elements that change independently
- For example, in printing selected items from a list, three things would likely be separated
  - The operation of printing items in a list
  - Creating a list from items
  - Identifying the items that are to be selected

- In imperative programming, the "selection" element would likely be separated by passing a threshold value that should be exceeded for selection into the output list.
- Functional programming style suggests passing an actual selection behavior rather than simply a threshold value

Introduction to higher-order functions

- Functions (or methods) that take arguments that represent behaviors rather than values, or return results of a behavioral kind are commonly referred to as *higher order functions*
- Such a function could use the argument behavior to augment the primary process the called function performs, for example:
  - Selecting/rejecting items
  - Sorting items
  - Transforming items

Java's Lambda Syntax

Develop Intelligence

- Lambdas
- The single abstract method and functional interfaces
- Method reference syntax

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

`(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
```

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

Method References

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.

Behavior Factories

Develop Intelligence

- A function can return a behavioral value (another function)
- A function that accepts behavioral arguments and returns behavior can potentially return behavior that makes use of (delegates to) the argument behaviors
  - This is literally a function that computes functions from other functions
  - Compare this with a function that computes a number from other numbers

Closures in Java

Develop Intelligence

- A local variable (including a method/function argument) normally has a lifetime that ends when the function that declares it returns to the caller
- If the function returns an object to the caller, that objects lifetime is controlled by reachability, and is definitely longer than the function invocation that returned it
- If such a function creates an object that refers to a method local variable, this seems to create a contradiction
  - However, many languages, including Java, address this by changing the rules related to such local variables
- In Java, such a variable must be final, or effectively final, and the object is given a copy of the value, as a result access to the local variable's value is safe, even though it might occur long after the local variable itself has ceased to exist

Function decorators

Develop Intelligence

- A function decorator is a factory that accepts one or more behavioral arguments (functions) and returns another function that has the same signature as the argument behavior(s), but which operates by performing some computation of its own, along with delegating to the argument function(s) for some of the computation.
- In some situations these might be referred to as function combinators
- Java provides some function decorators in its core libraries:
  - `java.util.function.Predicate` provides `and`, `or`, and `negate`
  - `java.util.function.Function` provides `andThen` and `compose`
  - `java.util.Comparator` provides `thenComparing`, `reversed` and others

The Importance of Generics in Functional Programming

- Generics provides a means for the programmer to specify consistency requirements for types, rather than specifying explicit types. E.g.:

```
<E,F> List<F> map(List<E> input, Function<E, F> op)
```

- Indicates that the function is described in terms of two type parameters, E, and F
- That the items (of type `E`) in the input list must be compatible with the argument type (also `E`) of the function `op`
- That the type of the items (`F`) in the returned list will be the same as the return type of the function `op`
- Using these features, very generalized operations can be specified, greatly reducing the need for code duplication
- Generics are a long standing feature of functional languages, and are often thought of as the polymorphism of functional programming

Co- and contravariance in higher-order functions

Develop Intelligence

- Strictly, a function that will be called with an actual parameter of type E does not need to declare a formal parameter of type E provided that E is assignment compatible with the formal parameter type
  - So, if the type F of the function's argument is a supertype of E, then the function is acceptable
  - This is an example of contra-variance
- Express this in Java with the syntax

```
Function<? super E, ...>
```

Co- and contravariance in higher-order functions

- Similarly, if the return value of a function that will be assigned to a type G the actual return type need only be assignment compatible, it does not have to be exactly G
  - So, if the type of the function's return is H then H must be assignable to G, or H must be a subclass of G
  - This is an example of co-variance
- Express this in Java with the syntax

```
Function<..., ? extends G>
```

Introduction to the Monad Concept

- A smart container
- Selecting items
- Changing items, the Functor
- The flatMap method, the Monad

Selecting Items with a `filter` Operation

- An operation commonly called `filter` can be applied to certain types of data container
- The operation creates a new container that:
  - Is, generally, of the same type
  - Contains items that pass a test provided by the caller
- The selection operation in Java is supplied as a `Predicate`

The Functor pattern

Develop Intelligence

- An operation traditionally called `map` can be applied to certain types of data container
- The operation takes an argument that is a Function:
  - Which takes a single data item of the existing type
  - And returns a new value, possibly of a different type
- The operation creates a new container that:
  - Contains items that result from applying this function to each input item
  - And therefore contains the same number of items as the original
- Such a container is an example of a *Functor*

The Monad pattern

Develop Intelligence

- An operation traditionally called `flatMap` can be applied to certain types of data container
- The operation takes an argument that is a Function:
  - Which takes a single data item of the existing type
  - And returns a new container of the same type
  - Containing values that may be of a different type
- The operation creates a new container that:
  - Contains all the items that result from applying this function to each input item
  - And therefore can contain more or fewer items than the original container
- Such a container is an example of a *Monad*

The Monad and Functor patterns

Develop Intelligence

- A Monad 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.
- 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).

Introduction to Producing a Final Result

Develop Intelligence

- An operation traditionally called `reduce` can be applied to certain types of data container
- The operation takes an argument that is a `BinaryOperator`:
  - That accepts two of the items in the container as arguments
  - And returns a single item of the same type
- Repeated application of this function allows creation of a single result from all the data in the container
- For a data item type T this *could* be:

```
T reduce(BinaryOperator<T> op)
```

Introduction to Producing a Final Result

- If the container is empty, three solutions are possible
  - Returning null--this is not a desirable style
  - Return an Optional
  - Provide an additional argument that is the initial value

- For a data item type T these look like:

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

T reduce(T initial, BinaryOperator<T> op)
```

More Advanced and Parallel Production of a Result

- The Monoid
- Reductions to a different type
- Fold operations

The Monoid

Develop Intelligence

- Given a data type **T**
  - and an associative binary operation **Op** on **T**
  - and a particular value **Id** of type `T`, such that for all x, **x OP Id = Id**
- The group of three items is called a *monoid*
  - The **Id** value is called the *identity*
- If a reduce operation is operating with a monoid, it is safe to divide the data up into sub-groups, and process those groups in separate computations
  - The sub results can be reassembled into the same result provided only that the ordering is not altered
- If the operation **Op** is also commutative, then order need not be maintained
- Using a monoid, and particularly using a commutative operation with monoid properties, allows parallel data processing and increasing throughput in the reduce operation

Reductions to a Different Type

- A variant of the `reduce` operation can produce a result of a different type
  - Assume the result type is **`R`**
- The combining operation will again be a `BiFunction`:
  - That accepts one argument of type **`R`**
  - And one argument of type **`T`**
  - And returns a single item of type **`R`**
- In principle, this operation could look like this:

```
T reduce(R initial, BiFunction<R,T,R> op)
```

- Operations of this kind are sometimes called *fold*, *foldLeft*, or *foldRight*
  - Though there can be variations of meaning or implementation

Parallel Reductions

Develop Intelligence

- If a reduce / fold type operation produces the same output type, it can be parallelized directly.
- If the operation produces a different type, then a combining function must be provided for parallel operations to work
  - Each separate sub-operation produces a result value of type R
  - The separate R values must be combined to a single, final, R
- The typical form of a parallel-capable reduction to different type is:

```
T reduce(R initial,
         BiFunction<R,T,R> op,
         BinaryOperator<R> combiner)
```

Overview of Monad-like Libraries and Frameworks

- Monad or monad-like libraries common in Java include
  - The `Stream` API
  - Java's promise API, the `CompletableFuture`
  - Apache Spark

Handling Errors in Functional Programming

Develop Intelligence

- Using reliable, purpose-specific, abstract data types can avoid problems
- Total vs partial functions
- The trouble with exceptions
- Using Optional and Either monad-like features
- Wrapping code that throws exceptions

Reliable Abstract Data Types

Develop Intelligence

- In this context, and abstract data type is a well-encapsulated structured data type which requires that all interactions are mediated by methods
- If an object representing a business concept can ever be in an invalid state, all users of these objects become responsible for checking the validity of such objects, and deciding how to behave if they are invalid
  - Imagine a date representing February 29th, 2021 (which is not a leap year)
  - That decision is likely impossible to resolve in a valid way
  - This demands a frightening amount of code embedded in clients of the object, and that code has nothing to do with the job at hand in the place it occurs
- Instead, ensure that such an object enforces its own integrity at all times, and throws an exception if an attempt is made to place it in an illegal state
  - This must be true from the moment of construction

Total vs Partial Functions

Develop Intelligence

- A total function returns a value for all input
  - How can this be possible with code that can fail and need recovery, such as opening a file?
  - Return an object that indicates "success or the problem"
  - If this object has monad-like properties for processing the success and the problem parts independently, writing continuation or recovery code is clean
- Such functions are much easier to compose and reuse than those that throw exceptions

Exceptions and Other Problem Reports

- Functional code uses exceptions only to represent program bugs
  - Which in turn require shutting down the program promptly before it does any more damage
- Classes `Optional` (in the core JDK) and `Either` (not part of the JDK) are examples of this approach

Exceptions and Other Problem Reports

- Wrapping code that throws exceptions with a function decorator allows relatively convenient conversion to code that reports a problem using (for example) an `Optional`
- A declaration of a "function that can throw exception" will be needed for lambdas throwing checked exceptions to be processed
  - This might be a good place for the wrapper operation too

Exceptions and Other Problem Reports

Develop Intelligence

```
@FunctionalInterface
public interface ExFunction<E,F> {
  F apply(E e) throws Throwable;
  static Function<E, Optional<F>> wrap(ExFunction<E,F> op) {
    e -> {
      try {
        return Optional.of(op.apply(e));
      } catch (Throwable t) {
        return Optional.empty();
      }
    }
  }
}
```

- Note that this approach avoids an exception, but fails to report the reason for failure
- Using an `Either` allows the reason to be reported and is likely better

THANK YOU

Develop Intelligence