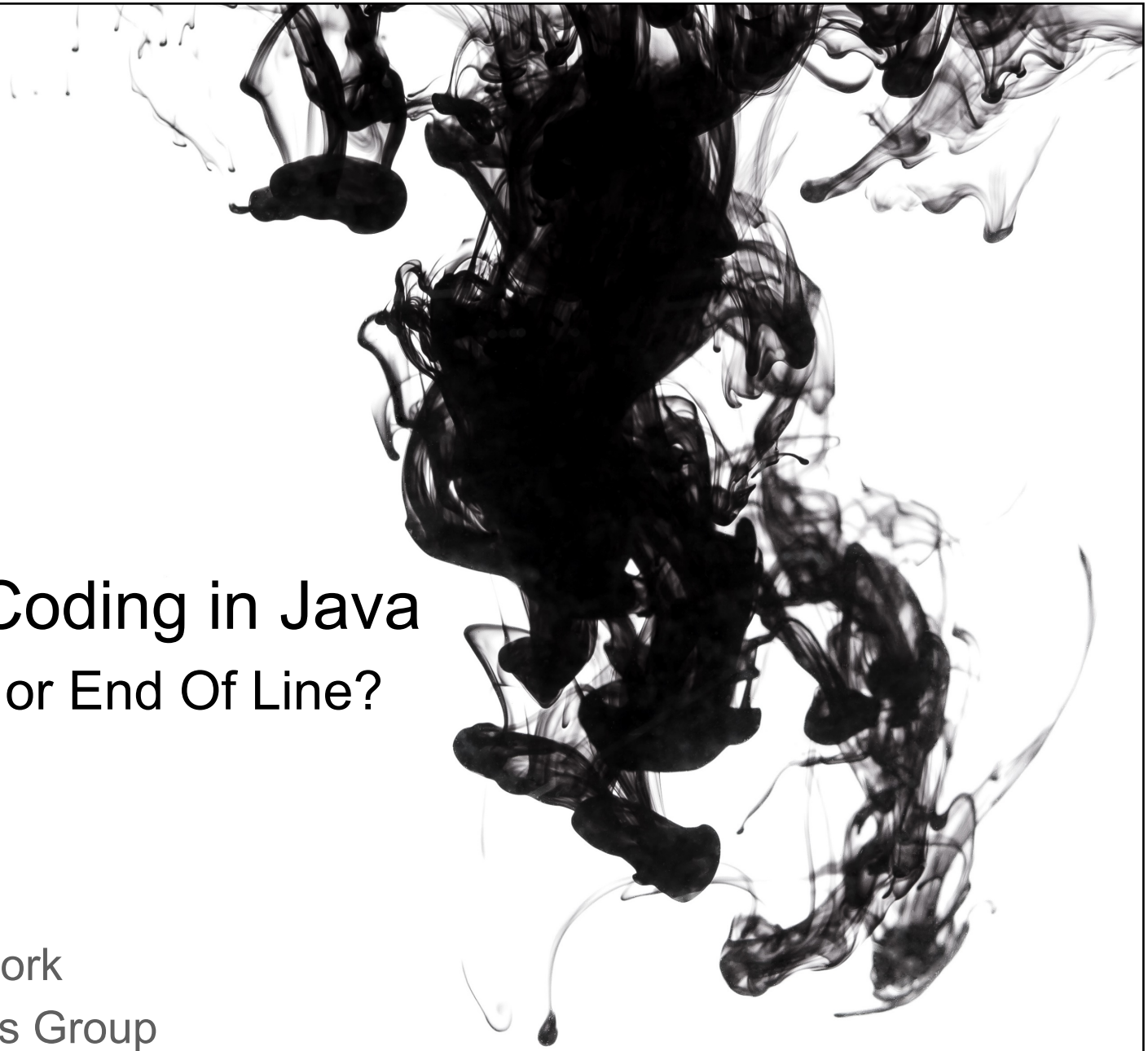


INSTIL

Functional Coding in Java

Gateway Drug or End Of Line?

March 2017

Functional Kats Cork

& Cork Java Users Group

garth.gilmour@instil.co | eamonn.boyle@instil.co

Thank You Kats and JUGs For Inviting Me!

About Me

Experienced trainer

- 15 years in the trenches
- Over 1000 deliveries

The day job

- Head of Learning at Instil
- Coaching, mentoring etc…

The night job(s)

- Husband and father
- Krav Maga Instructor

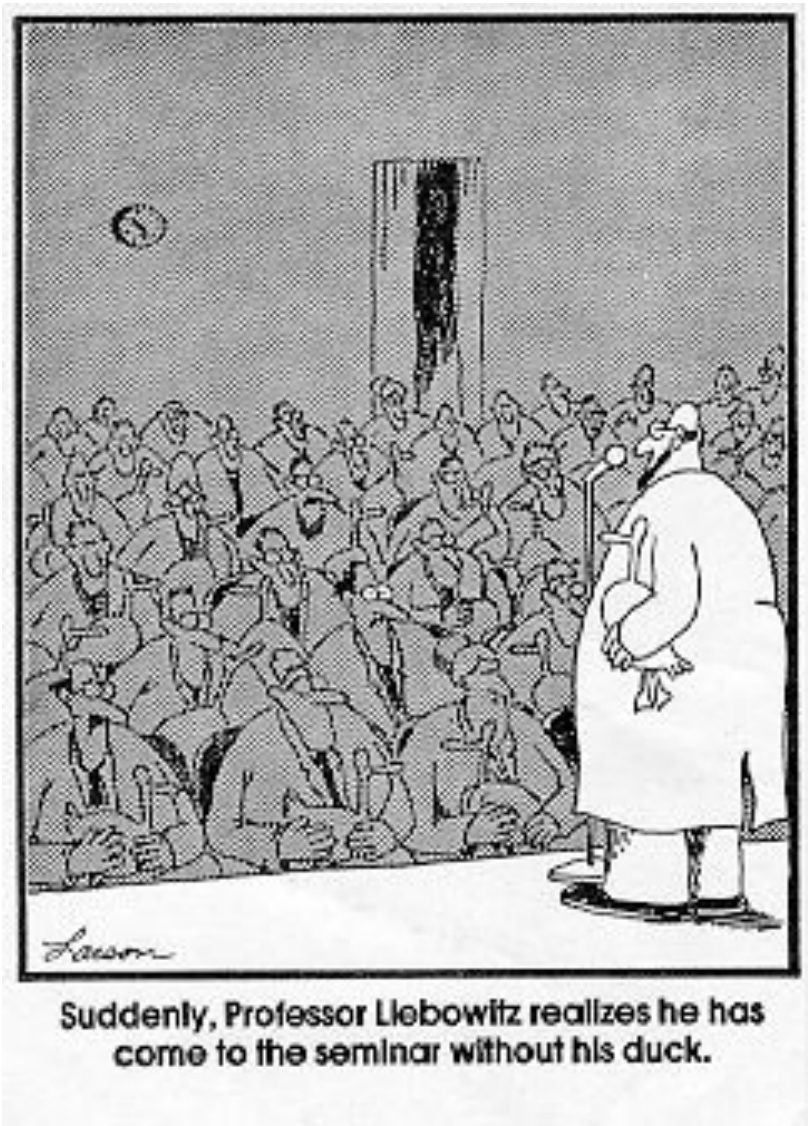
Big Scala fanboy

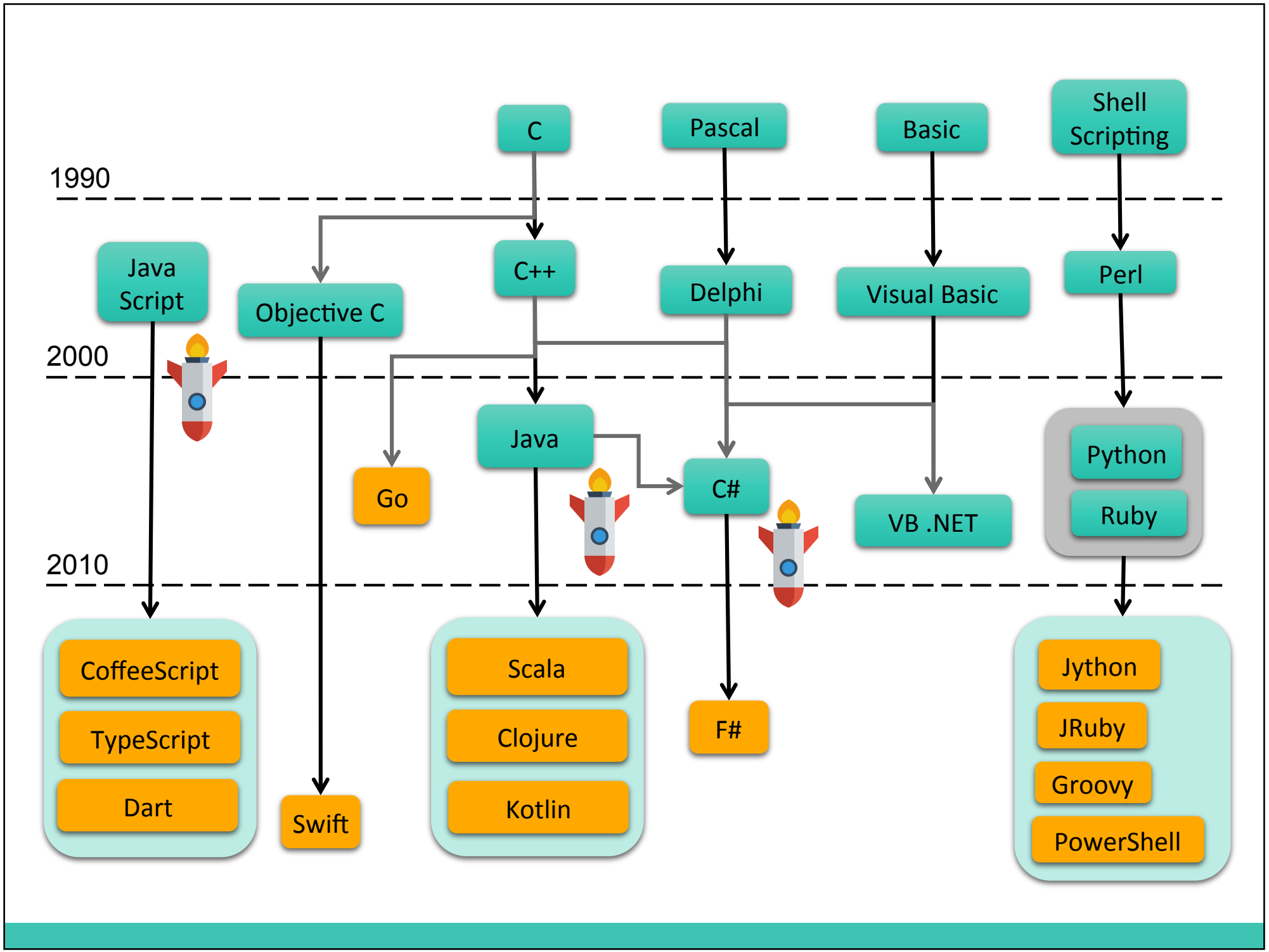
- Can we still be friends?

Arguing About Languages…

"There can be no peace until they renounce their Rabbit God and accept our Duck God."

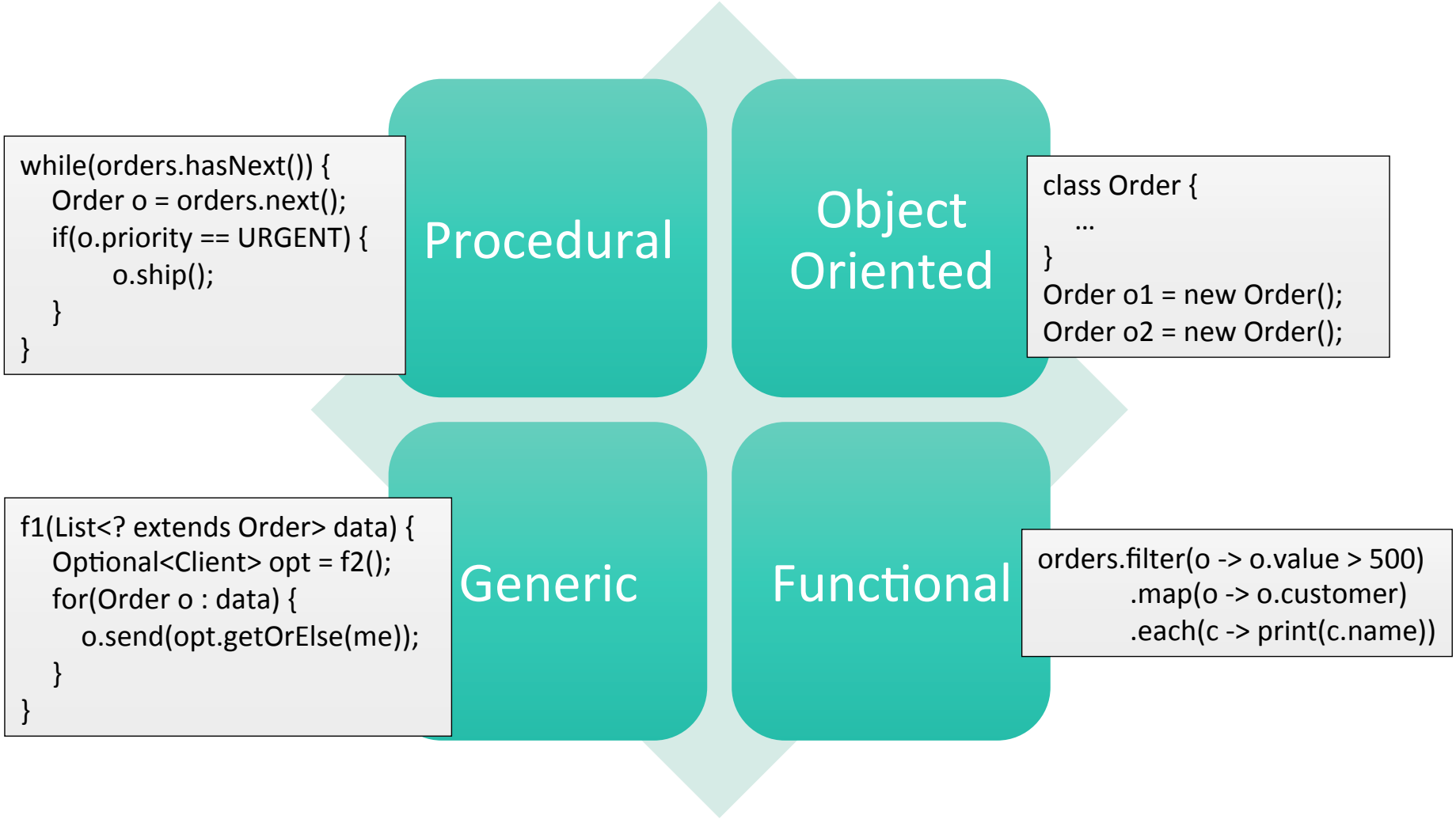
Who Brought A Laptop?




1990
2000
2010
C
Pascal
Basic
Shell Scripting
Java Script
Objective C
C++
Delphi
Visual Basic
Perl
Go
Java
C#
VB .NET
Python
Ruby
CoffeeScript
TypeScript
Dart
Swift
Scala
Clojure
Kotlin
F#
Jython
JRuby
Groovy
PowerShell

The Great Language Design Debate

Styles of Programming Supported in Java



```
public class Program {
	public static void main(String [] args) {
		doThing(new Callable<String>() {                                    //bad
				public String call() throws Exception {
					return "porthos";
				}
			},
			new Consumer<Exception>(){
				public void accept(Exception ex) {
					System.err.println(ex);
				}
			});
		doThing(() -> "athos", ex -> System.err.println(ex));               //better
		doThing(() -> "aramis", System.err::println);                       //disco
	}
	public static void doThing(Callable<String> input, Consumer<Exception> handler) {
		try {
			System.out.println(input.call());
		} catch (Exception e) {
			handler.accept(e);
		}
	}
}
```

The Road Not Taken…

The newest version of the Microsoft Visual J++ development environment supports a language construct called *delegates* or *bound method references*…

It is unlikely that the Java programming language will ever include this construct. Sun already carefully considered adopting it in 1996, to the extent of building and discarding working prototypes. Our conclusion was that bound method references are unnecessary and detrimental to the language…

We believe bound method references are *unnecessary* because another design alternative, *inner classes*, provides equal or superior functionality. In particular, inner classes fully support the requirements of user-interface event handling, and have been used to implement a user-interface API at least as comprehensive as the Windows Foundation Classes.

We believe bound method references are *harmful* because they detract from the simplicity of the Java programming language and the pervasively object-oriented character of the APIs….

Extracts From: About Microsoft's "Delegates"
Whitepaper by the Java language team at JavaSoft

How Many Of You Are Using Java 8?

How Many Of You Are Using Scala?

Some things remain the same…

Some FP related tips …

Top Tip!

Michael Feathers
@mfeathers

OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.

Daniel Spiewak @djspiewak · 3m
FP protip: think about how much of your logic *really* involves side effects. If it doesn't involve side effects, make it pure!

2

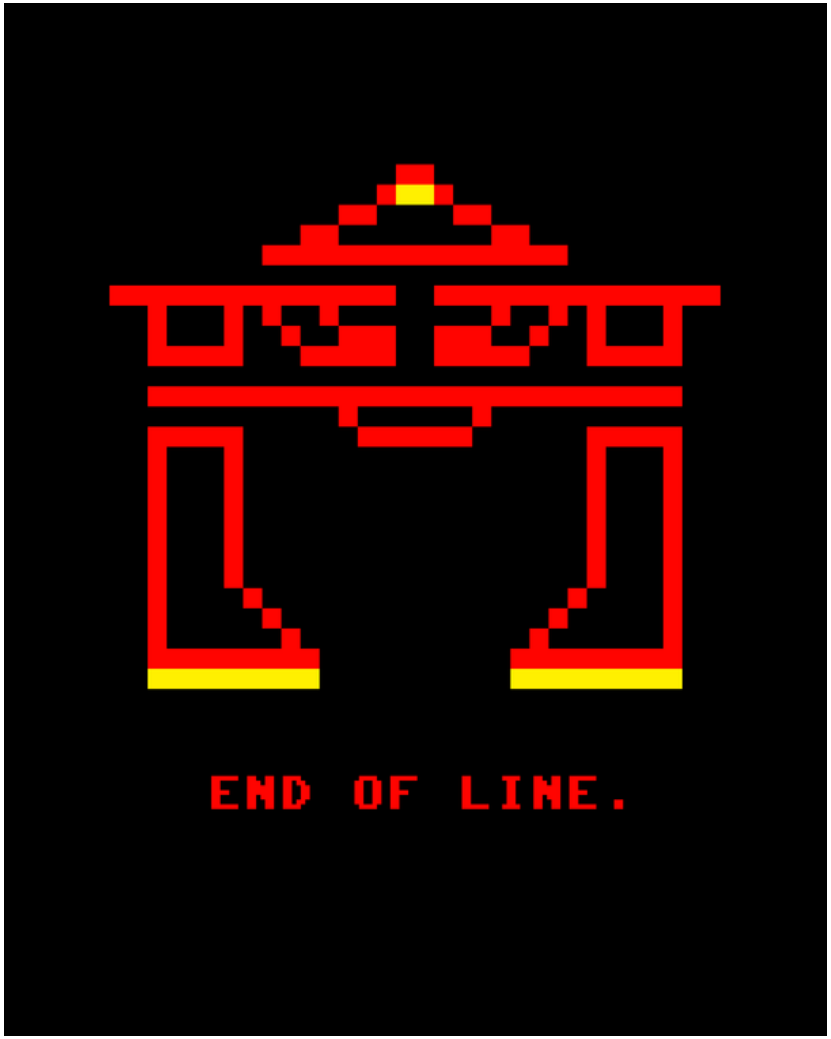
You Retweeted

Alex Gurney @ajtgurney · Jul 11
Programming is like wizardry, but the kind where the better a wizard you are, the less magic you use.

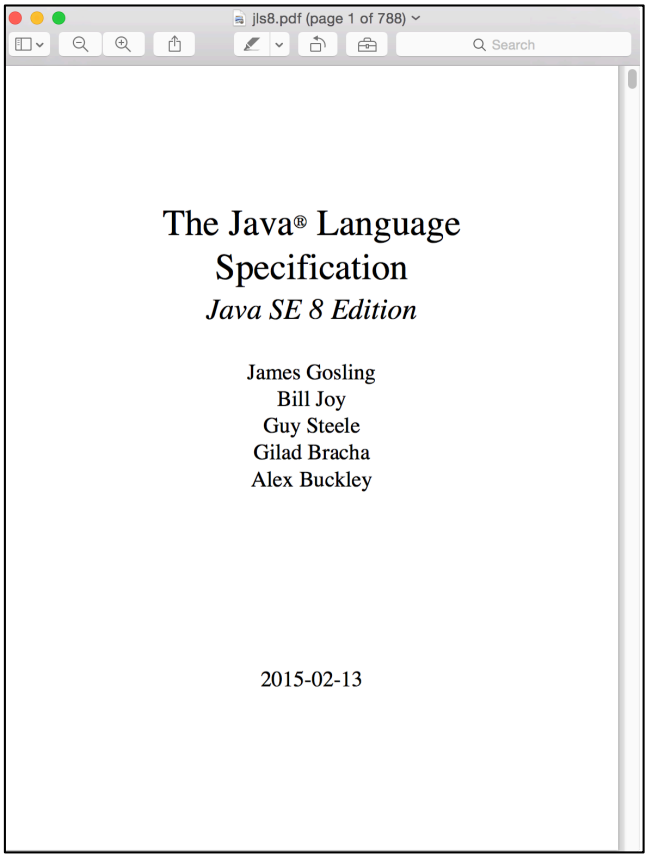
89 123

… and a funny one ☺

The Big Question on the JVM



What This Talk Is **Not** About



Level One (a)

- The Functional Toolkit

Lets Get Judgemental…

Top 10 Java 8 Annoyances (My Own Order)

1. Functional Interfaces
2. Separate Streams
3. Streams For Primitives
4. No Nested Functions
5. Single Use Streams
6. Confusing Reductions
7. Learning About Collect
8. Underpowered Optional
9. Strings Aren't Streams
10. Missing & Misnamed Methods

Annoyance 1: The Functional Interfaces

```
public class Program {
    public static void main(String [] args) {
        Supplier<String> ref1 = () -> "Wibble";
        Consumer<String> ref2 = s -> System.out.println(s);
        Predicate<String> ref3 = s -> s.length() == 4;
        Function<String,Integer> ref4 = s -> s.length();
        UnaryOperator<String> ref5 = s -> s.toUpperCase();
        BinaryOperator<String> ref6 = (s1,s2) -> s1 + s2;
        IntToDoubleFunction ref7 = i -> i * 1.0;

        System.out.println(ref1.get());
        ref2.accept("Wobble");
        System.out.println(ref3.test("abc"));
        System.out.println(ref3.test("abcd"));
        System.out.println(ref4.apply("abcde"));
        System.out.println(ref5.apply("abcdef"));
        System.out.println(ref6.apply("abcd","efg"));
        System.out.println(ref7.applyAsDouble(123));
    }
}
```

```
Wibble
Wobble
false
true
5
ABCDEF
abcdefg
123.0
```

Annoyance 2: Separate Streams

Annoyance 3: Streams For Primitives

DoubleStream	A sequence of primitive double-valued elements supporting sequential and parallel aggregate operations.
DoubleStream.Builder	A mutable builder for a `DoubleStream`.
IntStream	A sequence of primitive int-valued elements supporting sequential and parallel aggregate operations.
IntStream.Builder	A mutable builder for an `IntStream`.
LongStream	A sequence of primitive long-valued elements supporting sequential and parallel aggregate operations.
LongStream.Builder	A mutable builder for a `LongStream`.
Stream<T>	A sequence of elements supporting sequential and parallel aggregate operations.
Stream.Builder<T>	A mutable builder for a `Stream`.

Annoyance 3: Streams For Primitives

<R> **Stream**<R>	**flatMap**(**Function**<? super **T**,? extends **Stream**<? extends R>> mapper) Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
DoubleStream	**flatMapToDouble**(**Function**<? super **T**,? extends **DoubleStream**> mapper) Returns an DoubleStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
IntStream	**flatMapToInt**(**Function**<? super **T**,? extends **IntStream**> mapper) Returns an IntStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
LongStream	**flatMapToLong**(**Function**<? super **T**,? extends **LongStream**> mapper) Returns an LongStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.

Annoyance 4: No Nested Functions

A key FP best practice is:

- Don't go nuts with lambdas
- In particular avoid multi-line lambdas
- Instead put tasks into helper functions

But this creates an issue:

- Our class namespace becomes polluted with helpers

The problem is we don't have nested functions

- Unlike JavaScript, where we use them constantly

```
function fetchAndPrintCourses(domNodeId) {
    function listAllCourses(response) {...}
    function handleError(error) {...}
    var coursesCollection = api.all('courses');
    coursesCollection.getAll().then(listAllCourses, handleError);
}
```

Annoyance 5: Single Use Streams

```
import static java.util.Arrays.stream;
import java.util.stream.Stream;

public class Program {
    public static void main(String [] args) {
        Stream<String> stream = stream(new String[]{"ab","cd","ef"});
        System.out.println("First iteration of stream...");
        stream.forEach(System.out::println);
        System.out.println("Second iteration of stream...");
        try {
            stream.forEach(System.out::println);
        } catch(IllegalStateException ex) {
            System.err.println("Whoops should not have done that...");
        }
    }
}
```

```
First iteration of stream...
ab
cd
ef
Second iteration of stream...
Whoops should not have done that...
```

Streams and their Descriptions

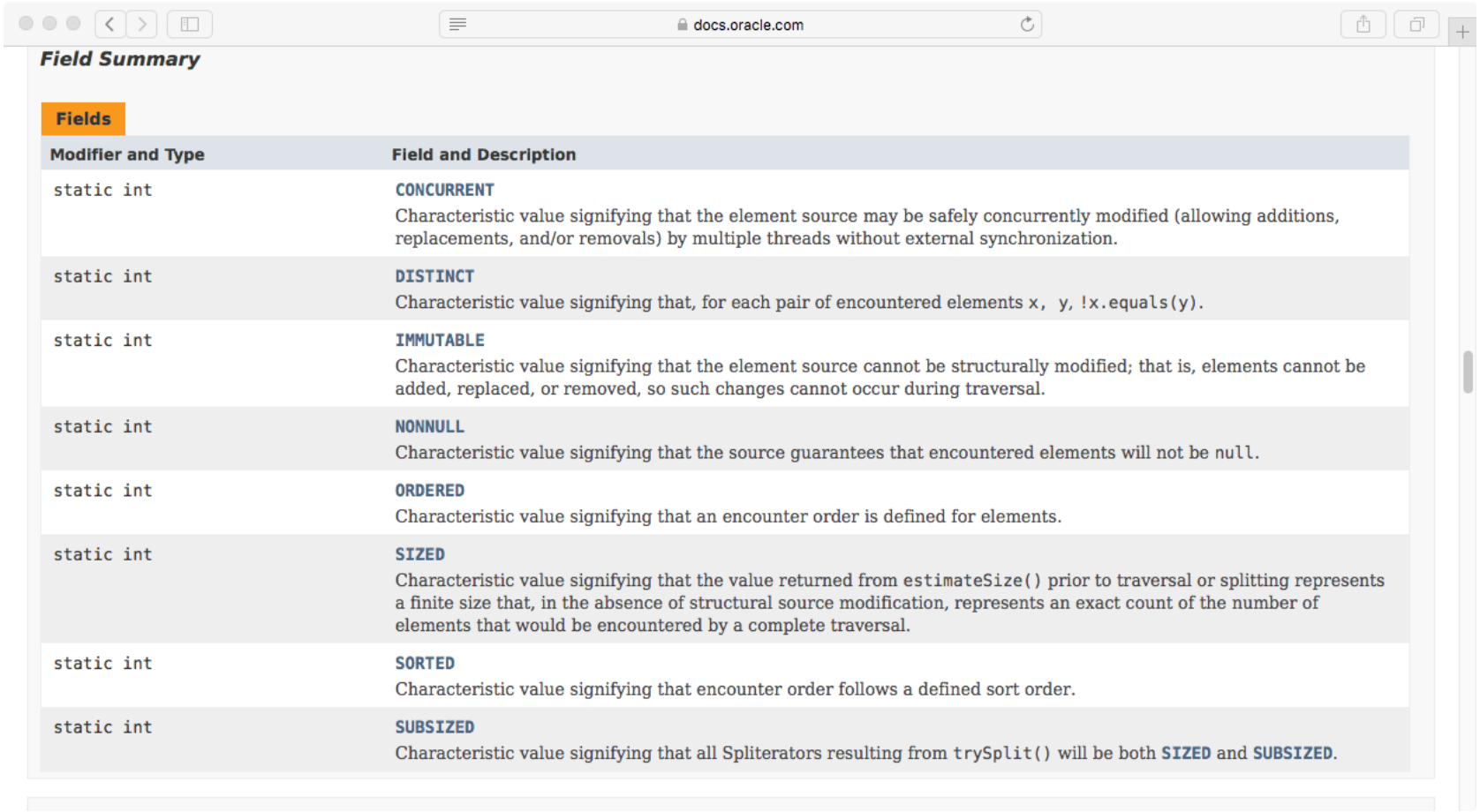
```
public class Program {
    public static void main(String[] args) {
        PrintStream out = System.out;
        String msg1 = "\t%d survived filter\n";
        String msg2 = "Processed result %d\n";

        out.println("Demo begins");
        IntStream input = IntStream.of(10,11,12,13,14,15,16,17,18,19,20);
        out.println("Input has been built");
        IntStream results = input.filter(x -> x % 2 == 0)
                                 .peek(x -> out.printf(msg1,x))
                                 .map(x -> x * 2);
        out.println("Results have been built");
        results.forEach(item -> out.printf(msg2,item));
    }
}
```

What would you expect this program to print?

Streams and their Descriptions

```
Demo begins
Input has been built
Results have been built
    10 survived filter
Processed result 20
    12 survived filter
Processed result 24
    14 survived filter
Processed result 28
    16 survived filter
Processed result 32
    18 survived filter
Processed result 36
    20 survived filter
Processed result 40
```

Flags Used in Stream Descriptions

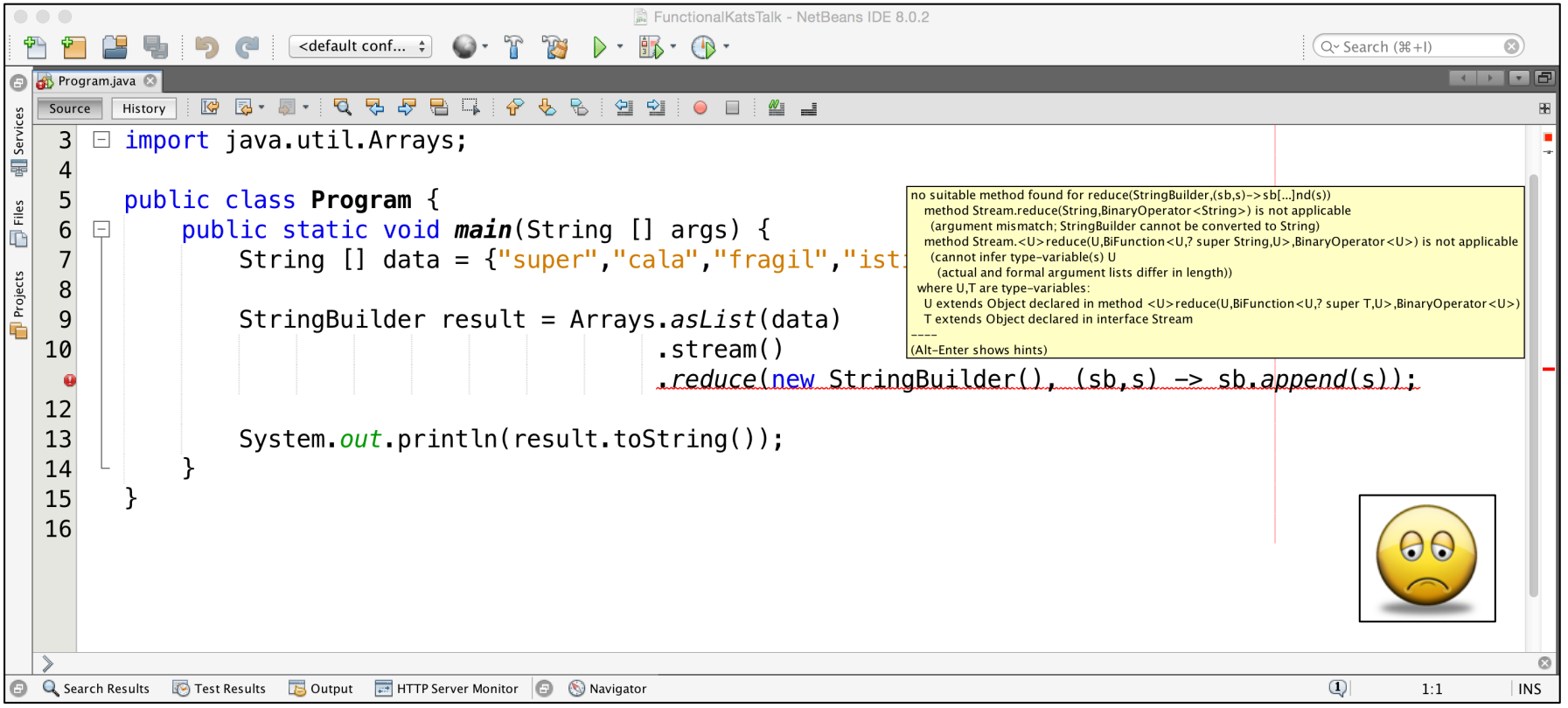


Field Summary

Fields

Modifier and Type	Field and Description
`static int`	**`CONCURRENT`** Characteristic value signifying that the element source may be safely concurrently modified (allowing additions, replacements, and/or removals) by multiple threads without external synchronization.
`static int`	**`DISTINCT`** Characteristic value signifying that, for each pair of encountered elements `x, y, !x.equals(y)`.
`static int`	**`IMMUTABLE`** Characteristic value signifying that the element source cannot be structurally modified; that is, elements cannot be added, replaced, or removed, so such changes cannot occur during traversal.
`static int`	**`NONNULL`** Characteristic value signifying that the source guarantees that encountered elements will not be `null`.
`static int`	**`ORDERED`** Characteristic value signifying that an encounter order is defined for elements.
`static int`	**`SIZED`** Characteristic value signifying that the value returned from `estimateSize()` prior to traversal or splitting represents a finite size that, in the absence of structural source modification, represents an exact count of the number of elements that would be encountered by a complete traversal.
`static int`	**`SORTED`** Characteristic value signifying that encounter order follows a defined sort order.
`static int`	**`SUBSIZED`** Characteristic value signifying that all Spliterators resulting from `trySplit()` will be both **`SIZED`** and **`SUBSIZED`**.

Annoyance 6: Confusing Reductions



Annoyance 7: Learning To Use Collect

```
public class Program {
    private static  Stream<String> sample() {
        return Stream.of("ab","cde","fghi","jk","lmn","opqr","st","uvw","xyz");
    }
    public static void main(String [] args) {
        System.out.println("Count is:\t" + sample().collect(counting()));
        System.out.println("Joining gives:\t" + sample().collect(joining()));
        System.out.println("Average gives:\t" + sample().collect(averagingDouble(s->s.length())));
        System.out.println("Summing gives:\t" + sample().collect(summingInt(s -> s.length())));

        Map<Integer,List<String>> groups = sample().collect(groupingBy(s -> s.length()));
        System.out.println("Grouping gives:");
        for(int key : groups.keySet()) {
            System.out.printf("\t%d indexes ", key);
            for(String value : groups.get(key)) {
                System.out.printf(" %s", value);
            }
            System.out.println();
        }

        List<String> list1 = sample().collect(toList());
        Set<String> set1 = sample().collect(toSet());
        List<String> list2 = sample().collect(toCollection(LinkedList::new));
        Set<String> set2 = sample().collect(toCollection(TreeSet::new));

        System.out.println(list1.getClass().getName());
        System.out.println(list2.getClass().getName());
        System.out.println(set1.getClass().getName());
        System.out.println(set2.getClass().getName());
    }
}
```

```
Count is:       9
Joining gives:  abcdefghijklmnopqrstuvwxyz
Average gives:  2.888888888888889
Summing gives:  26
Grouping gives:
        2 indexes  ab jk st
        3 indexes  cde lmn uvw xyz
        4 indexes  fghi opqr
java.util.ArrayList
java.util.LinkedList
java.util.HashSet
java.util.TreeSet
```

Annoyance 8: Underpowered Optional

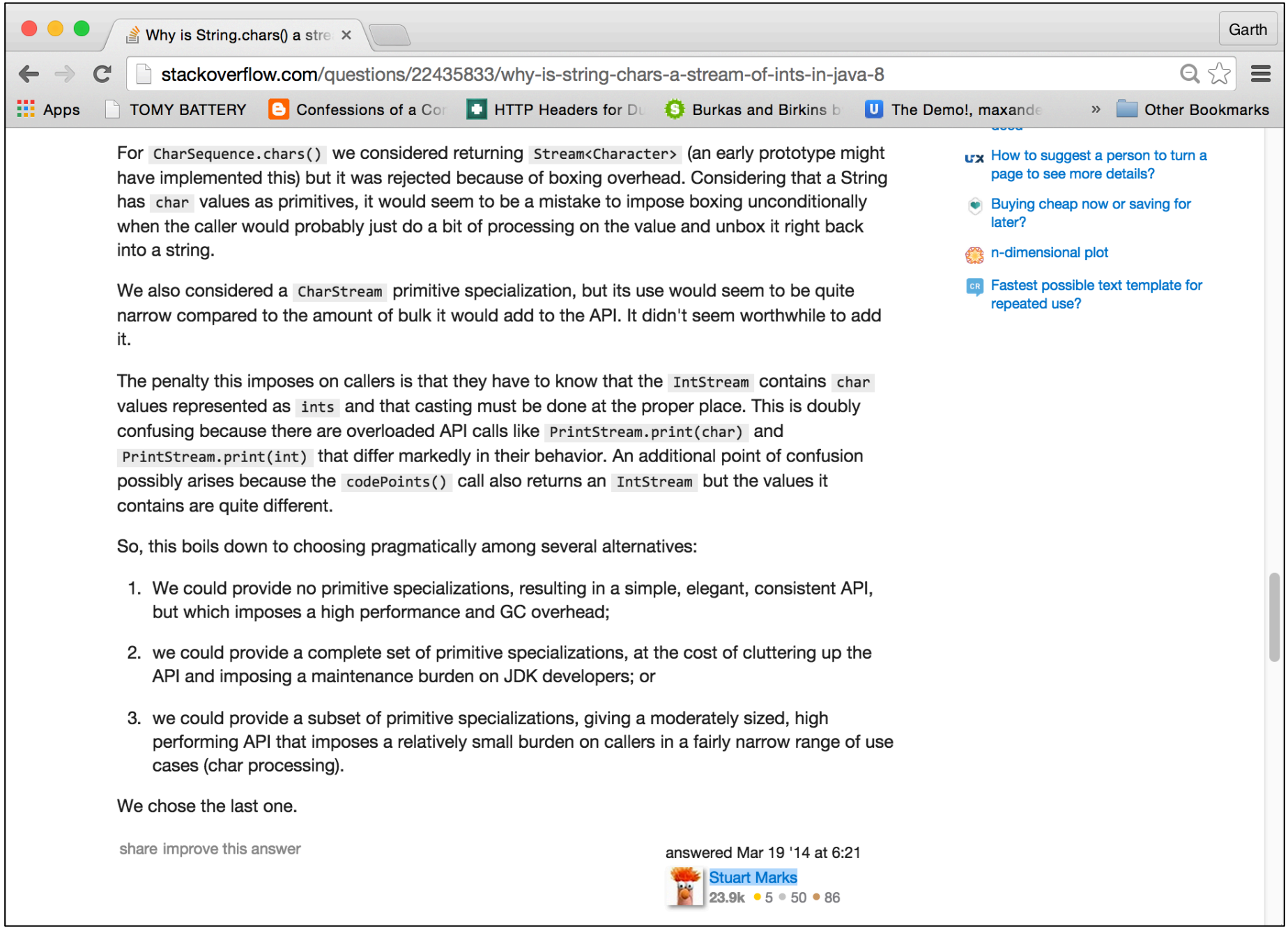
```
Optional<String> result1 = fetchSystemProperty("java.version");
Optional<String> result2 = fetchSystemProperty("wibble");

result1.ifPresent(System.out::println);
result2.ifPresent(System.out::println);
printDivider();
System.out.println(result1.orElse("No such property!"));
System.out.println(result2.orElse("No such property!"));
printDivider();
System.out.println(result1.orElseGet(() -> "No such property!"));
System.out.println(result2.orElseGet(() -> "No such property!"));
printDivider();
Supplier<IllegalArgumentException> errorHandler = () -> new IllegalArgumentException("No such property!");
System.out.println(result1.orElseThrow(errorHandler));
System.out.println(result2.orElseThrow(errorHandler));
```

NB: Partially fixed in Java 9

```
1.8.0
------------
1.8.0
No such property!
------------
1.8.0
No such property!
------------
1.8.0
Exception in thread "main" java.lang.IllegalArgumentException: No such property!
```

Annoyance 9: Strings Aren't Streams

```
public class Program {
    public static void main(String[] args) {
        String data = "s@u€p%e&r^c!a(l)a[f]{r}:a;g?e|l<i>s.t,i±c";

        System.out.println(buildStringOfLetters(data));
        System.out.println(buildStringOfLettersFP(data));
    }
    private static String buildStringOfLetters(String data) {
        StringBuilder result = new StringBuilder();
        for(int i=0;i<data.length();i++) {
            char ch = data.charAt(i);
            if(Character.isLetter(ch)) {
                result.append(ch);
            }
        }
        return result.toString();
    }
    private static String buildStringOfLettersFP(String data) {
        StringBuilder builder = data.codePoints()
                                    .filter(Character::isLetter)
                                    .collect(StringBuilder::new,
                                             StringBuilder::appendCodePoint,
                                             StringBuilder::append);

        return builder.toString();
    }
}
```



For `CharSequence.chars()` we considered returning `Stream<Character>` (an early prototype might have implemented this) but it was rejected because of boxing overhead. Considering that a String has `char` values as primitives, it would seem to be a mistake to impose boxing unconditionally when the caller would probably just do a bit of processing on the value and unbox it right back into a string.

We also considered a `CharStream` primitive specialization, but its use would seem to be quite narrow compared to the amount of bulk it would add to the API. It didn't seem worthwhile to add it.

The penalty this imposes on callers is that they have to know that the `IntStream` contains `char` values represented as `ints` and that casting must be done at the proper place. This is doubly confusing because there are overloaded API calls like `PrintStream.print(char)` and `PrintStream.print(int)` that differ markedly in their behavior. An additional point of confusion possibly arises because the `codePoints()` call also returns an `IntStream` but the values it contains are quite different.

So, this boils down to choosing pragmatically among several alternatives:

1. We could provide no primitive specializations, resulting in a simple, elegant, consistent API, but which imposes a high performance and GC overhead;
2. we could provide a complete set of primitive specializations, at the cost of cluttering up the API and imposing a maintenance burden on JDK developers; or
3. we could provide a subset of primitive specializations, giving a moderately sized, high performing API that imposes a relatively small burden on callers in a fairly narrow range of use cases (char processing).

We chose the last one.

share improve this answer

answered Mar 19 '14 at 6:21

Stuart Marks
23.9k ● 5 ● 50 ● 86

Annoyance 10: Misnamed / Missing Methods

Name	What It Should Have Been	Comment
allMatch	all	Growl
any Match	any	Growl
noneMatch	none	Growl
findFirst	first	WTF
findAny	any	WTF

Missing Method	Comment
find	Have to use ‘filter(foo).findFirst()’ WTF
single	WTF
sum	Have to use collect instead
groupBy	Have to use collect instead
combinations	WTF

Level One (b)

- The Case Study

The Sample Data to be Queried

```
object Builder {
  def buildData() = {
    val dave = Trainer("Dave Jones", 500.0, List("SQL","Perl","PHP"))
    val jane = Trainer("Jane Smith", 750.0, List("SQL","Java","JEE"))
    val pete = Trainer("Pete Hughes", 1000.0, List("Java", "JEE", "C#", "Scala"))
    val mary = Trainer("Mary Donaghy", 1250.0, List("Java", "JEE", "C#", "C++"))

    List(Course("AB12","Intro to Scala",BEGINNER,4,List(pete)),
         Course("CD34","JEE Web Development",INTERMEDIATE,3,List(pete, mary, jane)),
         Course("EF56","Meta-Programming in Ruby",ADVANCED,2,List()),
         Course("GH78","OO Design with UML",BEGINNER,3,List(jane,pete,mary)),
         Course("IJ90","Database Access with JPA",INTERMEDIATE,3, List(jane)),
         Course("KL12","Design Patterns in C#",ADVANCED,2, List(pete,mary)),
         Course("MN34","Relational Database Design",BEGINNER,4, List(jane, dave)),
         Course("OP56","Writing MySql Stored Procedures",INTERMEDIATE,1, List(jane, dave)),
         Course("QR78","Patterns of Parallel Programming",ADVANCED,2, List(pete,mary)),
         Course("ST90","C++ Programming for Beginners",BEGINNER,5,List(mary)),
         Course("UV12","UNIX Threading with PThreads",INTERMEDIATE,2,List()),
         Course("WX34","Writing Linux Device Drivers",ADVANCED,3,List(mary)))
  }
}
```

Titles of Courses

```
def titlesOfCourses() {
  println("Titles of courses:")
  courses.map(_.title)
         .foreach(printTabbed)
  println("---")
}
```

```
private static void titlesOfCourses(Stream<Course> courses) {
    System.out.println("The titles of all the courses are:");
    courses.map(Course::getTitle)
           .forEach(Program::printTabbed);
}
```

Titles of Courses Without Trainers

```
def titlesOfCoursesWithoutATrainer() {
  println("Titles of courses without a trainer:")
  courses.filter(_.instructors.isEmpty)
         .map(_.title)
         .foreach(printTabbed)
  println("---")
}
```

```
private static void titlesOfCoursesWithoutATrainer(Stream<Course> courses) {
    System.out.println("The titles of all the courses without a trainer are:");
    courses.filter(c -> c.getInstructors().isEmpty())
           .map(Course::getTitle)
           .forEach(Program::printTabbed);
}
```

Names and Rates of Trainers

```
def namesAndRatesOfTrainers() {
  println("Names and rates of trainers:")
  courses.flatMap(_.instructors)
         .distinct
         .map(t => (t.name,t.rate))
         .foreach(printTabbed)
  println("---")
}
```

```
private static void namesAndRatesOfTrainers(Stream<Course> courses) {
    System.out.println("The names and rates of all trainers are:");
    courses.flatMap(c -> c.getInstructors().stream())
           .distinct()
           .map(t -> new Pair<>(t.getName(),t.getRate()))
           .forEach(Program::printTabbed);
}
```

The Number of Advanced Courses

```
def theNumberOfAdvancedCourses() {
  println("The number of advanced courses")
  println(courses.count(_.courseType == ADVANCED))
  println("---")
}
```

```
private static void numberOfAdvancedCourses(Stream<Course> courses) {
    System.out.println("The number of advanced courses:");
    int result = courses.collect(summingInt(c -> c.getType() == ADVANCED ? 1 : 0));
    System.out.println("\t" + result);
}
```

Durations of (Non) Beginner Courses

```
def totalDurationsOfBeginnerAndNonBeginnerCourses() {
  println("Total days for both beginner and non-beginner courses")
  val splitCourses = courses.partition(_.courseType == BEGINNER)
  val beginnerDuration = splitCourses._1.map(_.duration).sum
  val nonBeginnerDuration = splitCourses._2.map(_.duration).sum
  println((beginnerDuration,nonBeginnerDuration))
  println("---")
}
```

Durations of (Non) Beginner Courses

```
private static void totalDurationsOfBeginnerAndNonBeginnerCoursesV1(List<Course> courses) {
    System.out.println("Total course durations are:");
    totalDurationOfBeginnerCourses(courses.stream());
    totalDurationOfNonBeginnerCourses(courses.stream());
}
private static void totalDurationOfBeginnerCourses(Stream<Course> courses) {
    int result = courses.filter(c -> c.getType() == BEGINNER)
                        .mapToInt(Course::getDuration)
                        .sum();
    System.out.println("\tTotal for beginners is: " + result);
}
private static void totalDurationOfNonBeginnerCourses(Stream<Course> courses) {
    int result = courses.filter(c -> c.getType() != BEGINNER)
                        .mapToInt(Course::getDuration)
                        .sum();
    System.out.println("\tTotal for non-beginners is: " + result);
}
```

Durations of (Non) Beginner Courses

```
private static void totalDurationsOfBeginnerAndNonBeginnerCoursesV2(Stream<Course> courses) {
    Pair<Integer,Integer> result = courses.reduce(new Pair<>(0,0),Program::totalDurations,(a,b) -> null);
    System.out.println("Total course durations are:");
    System.out.println("\tTotal for beginners is: " + result.getFirst());
    System.out.println("\tTotal for non-beginners is: " + result.getSecond());
}
private static Pair<Integer,Integer> totalDurations(Pair<Integer,Integer> total, Course course) {
    int totalBeginners = total.getFirst();
    int totalNonBeginners = total.getSecond();
    int duration = course.getDuration();
    if(course.getType() == BEGINNER) {
        return new Pair<>(totalBeginners + duration, totalNonBeginners);
    } else {
        return new Pair<>(totalBeginners, totalNonBeginners + duration);
    }
}
```

Durations of (Non) Beginner Courses

```
private static void totalDurationsOfBeginnerAndNonBeginnerCoursesV3(List<Course> courses) {
    int durationForBeginner = courses.stream()
                                     .filter(c -> c.getType() == BEGINNER)
                                     .collect(summingInt(Course::getDuration));
    int durationForNonBeginner = courses.stream()
                                        .filter(c -> c.getType() != BEGINNER)
                                        .collect(summingInt(Course::getDuration));

    System.out.println("Total course durations are:");
    System.out.println("\tTotal for beginners is: " + durationForBeginner);
    System.out.println("\tTotal for non-beginners is: " + durationForNonBeginner);
}
```

Pairs of Trainers to Deliver Java Courses

```
def everyPairOfTrainersThatCouldDeliverJava() {
  println("Pairs of trainers that could deliver Java")
  courses.flatMap(_.instructors)
         .distinct
         .filter(_.skills.contains("Java"))
         .map(_.name)
         .combinations(2)
         .foreach(list => printf("%s and %s\n",list(0),list(1)))
  println("---")
}
```

Pairs of Trainers to Deliver Java Courses

```
private static void everyPairOfTrainersThatCouldDeliverJava(List<Course> courses) {
    System.out.println("Every pair of trainers that could deliver Java:");
    javaTrainers(courses).flatMap(t1 -> javaTrainers(courses).filter(t2 -> t1 != t2)
                                                             .map(t2 -> new Pair<>(t1,t2)))
                         .map(p -> new Pair<>(p.getFirst().getName(),p.getSecond().getName()))
                         .distinct()
                         .forEach(Program::printTabbed);
}
private static Stream<Trainer> javaTrainers(List<Course> courses) {
    return courses.stream()
                  .flatMap(c -> c.getInstructors().stream())
                  .distinct()
                  .filter(t -> t.getSkills().contains("Java"));
}
```

The Cost of ‘JEE Web Development’

```
def possibleCostsOfJeeWebDevelopment() {
  println("Possible costs of 'JEE Web Development'")
  val course = courses.find(_.title == "JEE Web Development")
  val duration = course.map(_.duration)
                       .getOrElse(0)
  course.foreach(_.instructors
                  .map(p => (p.name, p.rate * duration))
                  .foreach(printTabbed))
  println("---")
}
```

The Cost of ‘JEE Web Development’

```
private static void possibleCostsOfJeeWebDevelopment(List<Course> courses) {
    System.out.println("Possible costs of 'JEE Web Development'");
    Optional<Course> selected = courses.stream()
                                       .filter(c -> c.getTitle().equals("JEE Web Development"))
                                       .findFirst();

    String msg = "\t%s at a cost of %.2f\n";
    int duration = selected.map(c -> c.getDuration())
                           .orElse(0);
    selected.ifPresent(c -> c.getInstructors()
                             .stream()
                             .forEach(t -> System.out.printf(msg, t.getName(), t.getRate() * duration)));
}
```

Grouping ID’s and Titles by Type

```
def coursesIdsAndTitlesGroupedByType() {
  println("Course ID's and titles grouped by type")
  def process(row: (Value, List[Course])) {
    printTabbed(row._1)
    row._2.foreach(c => printf("\t\t%s %s\n", c.id, c.title))
  }
  courses.groupBy(_.courseType)
         .foreach(process)
}
```

Grouping ID's and Titles by Type

```
private static void coursesIdsAndTitlesGroupedByType(Stream<Course> courses) {
    System.out.println("Course id's and titles grouped by type are:");
    Map<CourseType, List<String>> coursesByType
            = courses.collect(groupingBy(Course::getType, mapping(Course::getTitle,toList())));
    coursesByType.forEach(Program::printGroup);
}
private static void printGroup(CourseType key, List<String> values) {
    String msg = "\tCourses of type %s are:\n";
    System.out.printf(msg, key);
    values.stream().forEach(s -> System.out.println("\t\t" + s));
}
```

Level Two

- Getting Funky With RESTful

The Reactive Web Framework in Spring 5

‘Reactive Design’ has been trending for a while now

- After being announced to great fanfare its quietly been integrated into both client and server side frameworks
- E.g. Angular 2 returns values as RxJS Observables

The goal is to be ‘asynchronous all the way down’

- Removing all blocking operations in order to increase performance, scalability and utilise many cores
- Were ‘many’ could mean tens of thousands

Spring 5 offers a reactive version of Spring MVC

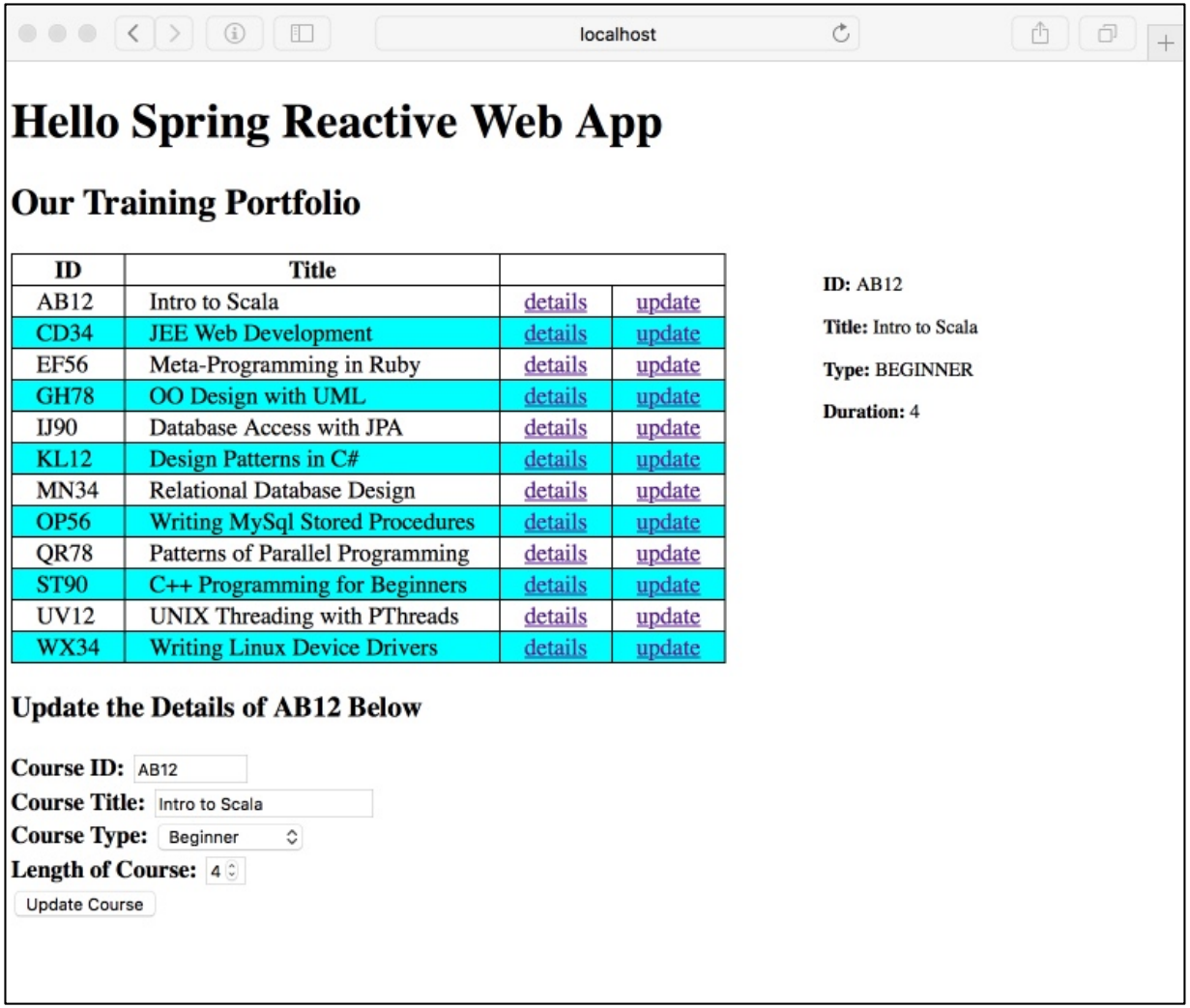
- Based on the ‘Reactor’ implementation of Reactive Streams
- A ‘Flux<T>’ is a stream of ‘Item’ which may be asynchronous
- A ‘Mono<T>’ is a the same but holding a single item
  - Useful in the simpler cases e.g. when retrieving a list

```
@RestController
public class CourseController {
    @GetMapping(value="/courses", produces="application/json")
    public Flux<Course> fetchAllTheCoursesAsJSON() {
        return repository.allCourses();
    }
    @GetMapping(value="/courses", produces="application/xml")
    public Mono<CourseList> fetchAllTheCoursesAsXML() {
        CourseList courses = new CourseList(new ArrayList<>());
        return repository.allCourses()
                         .reduce(courses, (a,b) -> a.add(b));
    }
    @GetMapping("/courses/{id}")
    public Mono<Course> findById(@PathVariable String id) {
        return this.repository.singleCourse(id);
    }
    @PutMapping("/courses")
    Mono<Void> addOrUpdate(@RequestBody Mono<Course> courseStream) {
        return this.repository.addOrUpdateCourse(courseStream).then();
    }
    @Autowired
    public void setRepository(CourseRepository repository) {
        this.repository = repository;
    }
    private CourseRepository repository;
}
```

```java
@Component
public class CourseRepositoryImpl implements CourseRepository {
    @Autowired
    public CourseRepositoryImpl(@Qualifier("portfolio") List<Course> portfolio) {
        this.portfolio = portfolio;
    }
    public Mono<Course> singleCourse(String id) {
        Optional<Course> selected = portfolio.stream()
                                             .filter(c -> c.getId().equals(id))
                                             .findAny();
        return Mono.justOrEmpty(selected);
    }
    public Flux<Course> allCourses() {
        return Flux.fromIterable(portfolio);
    }
    public Mono<Void> addOrUpdateCourse(Mono<Course> mono) {
        return mono.then(this::adder);
    }
    private Mono<Void> adder(Course course) {
        if(portfolio.contains(course)) {
            portfolio.remove(course);
        }
        portfolio.add(course);
        return Mono.empty();
    }
    private List<Course> portfolio;
}
```

```java
public interface CourseRepository {
    Mono<Course> singleCourse(String id);
    Flux<Course> allCourses();
    Mono<Void> addOrUpdateCourse(Mono<Course> mono);
}
```

An Angular Front End to Spring Reactive MVC



But We've Seen All This Before…

Introducing Akka HTTP

Akka offers a message-oriented, actor based coding model

- It has been around (in its current form) since 2010

Akka HTTP lets you create and consume REST API's

- Replacing the earlier Spray framework

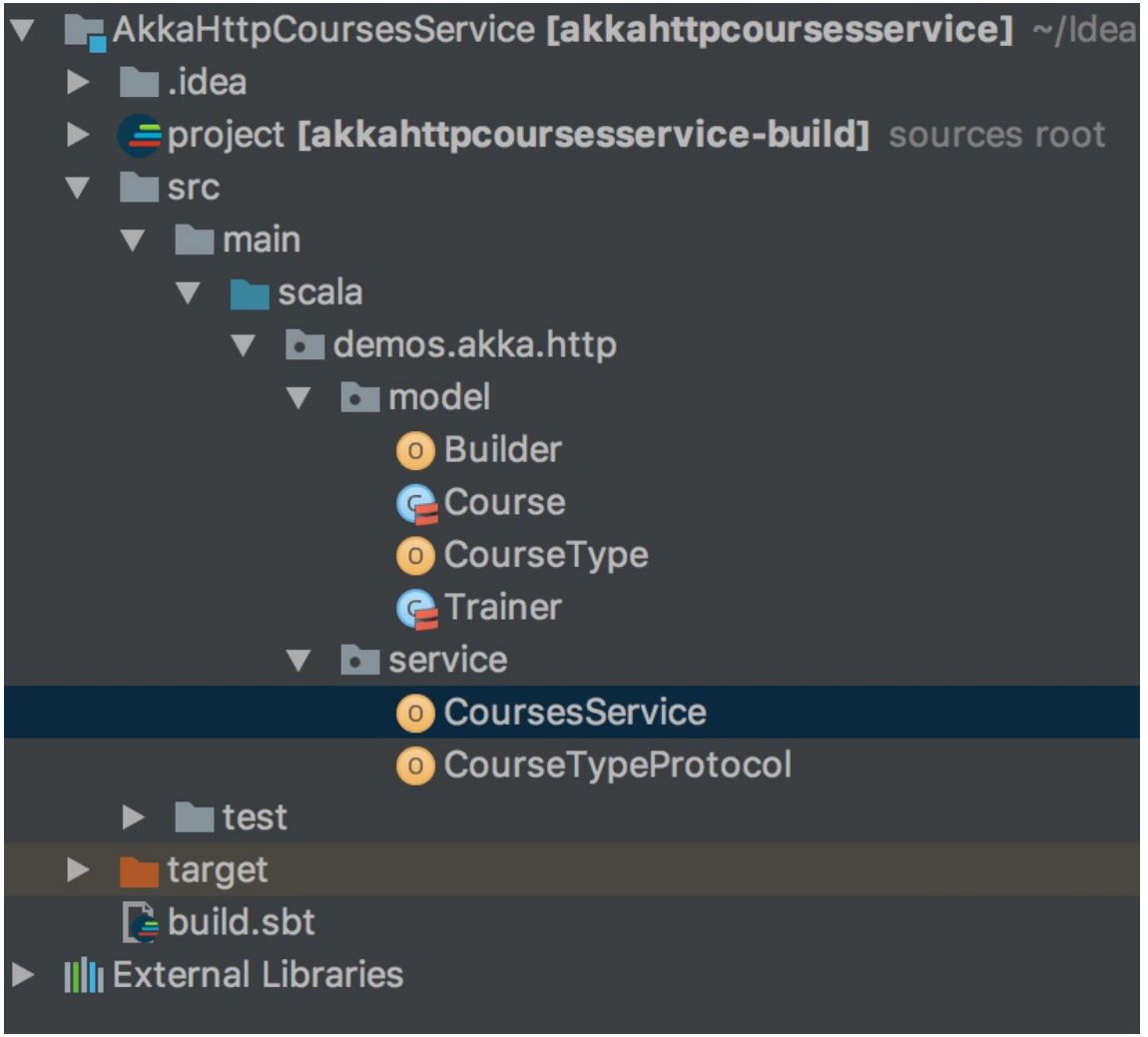
Everything you need for Reactive Design is already there

- It is typically combined with the Slick library for DB access

It can be used from both Java and Scala

- But the latter is much more succinct…

AKKA HTTP Courses Setup



AKKA HTTP Courses Setup

```
name := "AkkaHttpCoursesService"

version := "1.0"

scalaVersion := "2.12.1"

libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-http-core" % "10.0.5",
  "com.typesafe.akka" %% "akka-http" % "10.0.5",
  "com.typesafe.akka" %% "akka-http-testkit" % "10.0.5",
  "com.typesafe.akka" %% "akka-http-spray-json" % "10.0.5",
  "com.typesafe.akka" %% "akka-http-jackson" % "10.0.5",
  "com.typesafe.akka" %% "akka-http-xml" % "10.0.5"
)
```

AKKA HTTP Utility Methods

```
package demos.akka.http.service

import ...

object CoursesService extends Directives {
  def main(args: Array[String]) {...}

  def shutdownServer(bindingFuture: Future[ServerBinding], system: ActorSystem) = {
    implicit val executionContext = system.dispatcher
    bindingFuture
      .flatMap(_.unbind())                // unbind from the port
      .onComplete(_ => system.terminate()) // shutdown when done
  }

  private def pauseTillDone = {
    println("AKKA HTTP running on 'http://localhost:8080'")
    println("Hit RETURN to quit")
    StdIn.readLine()
  }
}
```

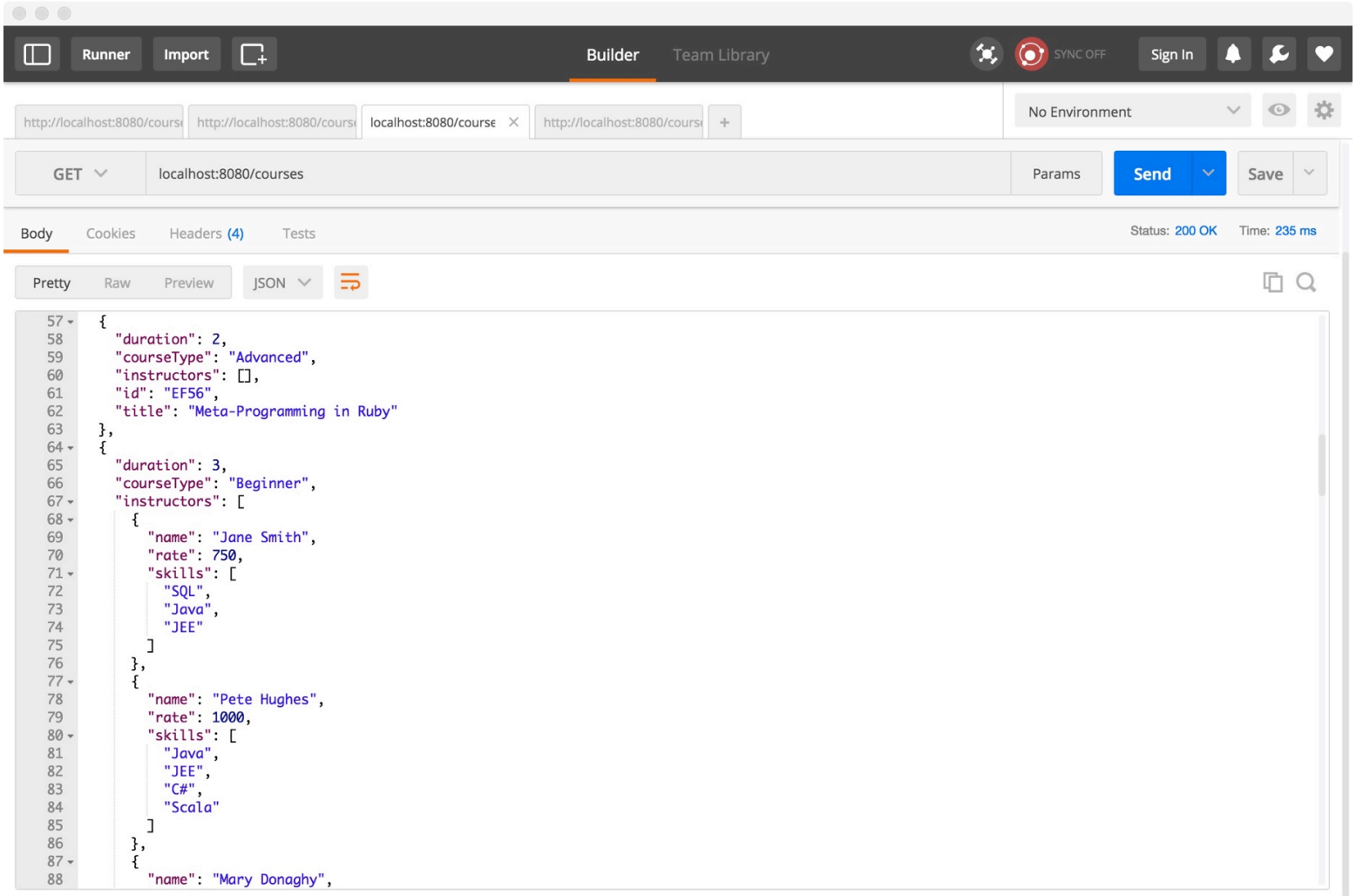
AKKA HTTP Server Loop & Routing DSL

```
object CoursesService extends Directives {
  def main(args: Array[String]) {
    //AKKA Infrastructure - injected as implicits
    implicit val system = ActorSystem("course-booking-system")
    implicit val materializer = ActorMaterializer()

    var courses = buildData()

    val routes = {
      path("courses") {
        get {
          complete(courses)
        }
      } ~
      pathPrefix("courses" / Segment) { id =>
        get {
          complete(courses.find(_.id == id))
        }~
        delete {
          courses = courses.filterNot(_.id == id)
          complete(HttpResponse(StatusCodes.NoContent))
        }~
        put {
          entity(as[Course]) { c =>
            courses = c :: courses
            complete(HttpResponse(StatusCodes.NoContent))
          }
        }
      }
    }
```

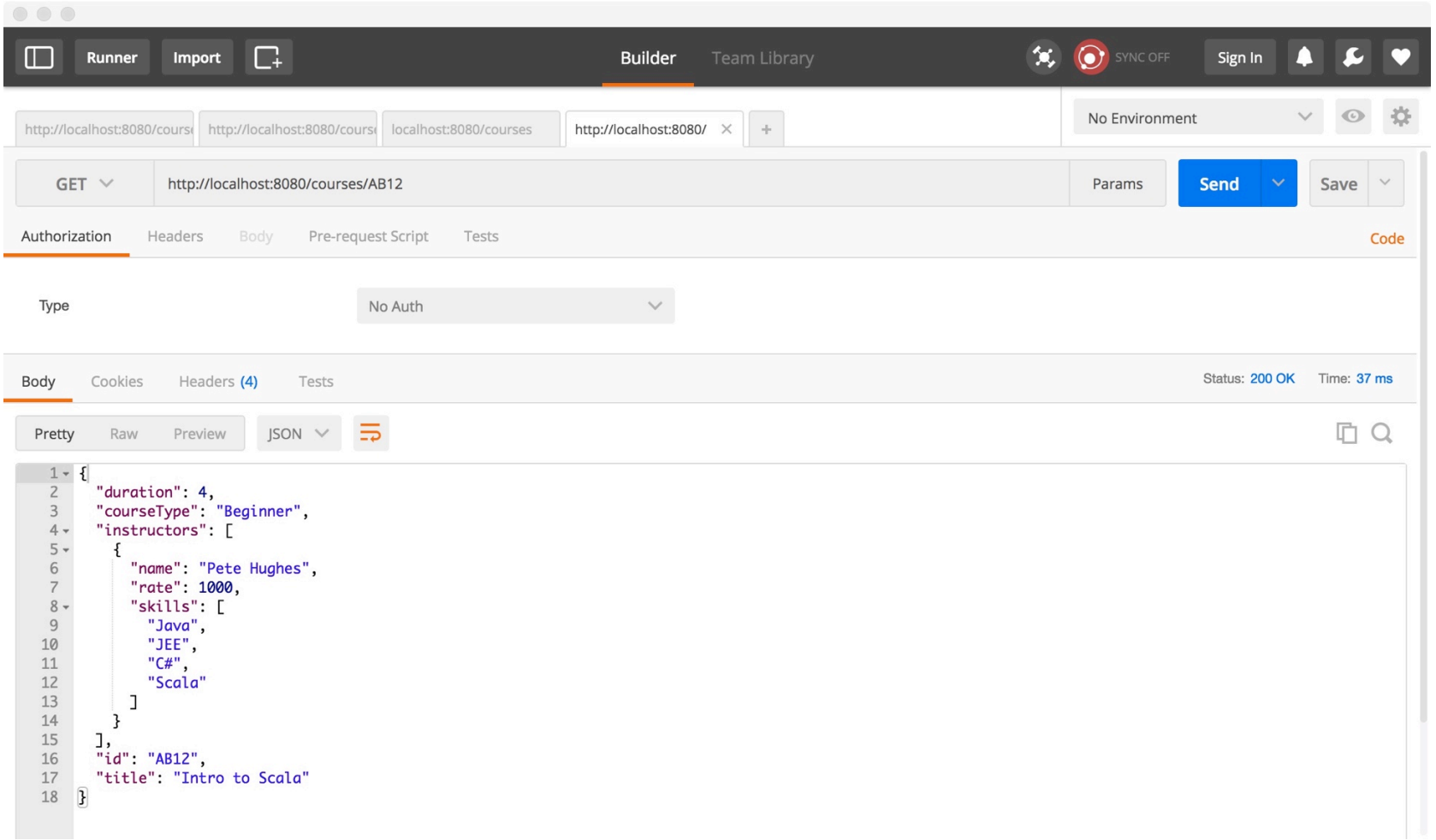
AKKA HTTP JSON Conversion

```
object CourseTypeProtocol extends SprayJsonSupport with DefaultJsonProtocol {
        implicit object CourseTypeFormat extends RootJsonFormat[CourseType.Value] {
                def write(obj: CourseType.Value) = JsString(obj.toString)
                def read(json: JsValue) = {
                        json match {
                                case JsString(txt) => CourseType.withName(txt)
                                case _ => deserializationError("Expected a value from enum CourseType")
                        }
                }
        }
        implicit val trainerFormat = jsonFormat3(Trainer)
        implicit val courseFormat = jsonFormat5(Course)
}
```

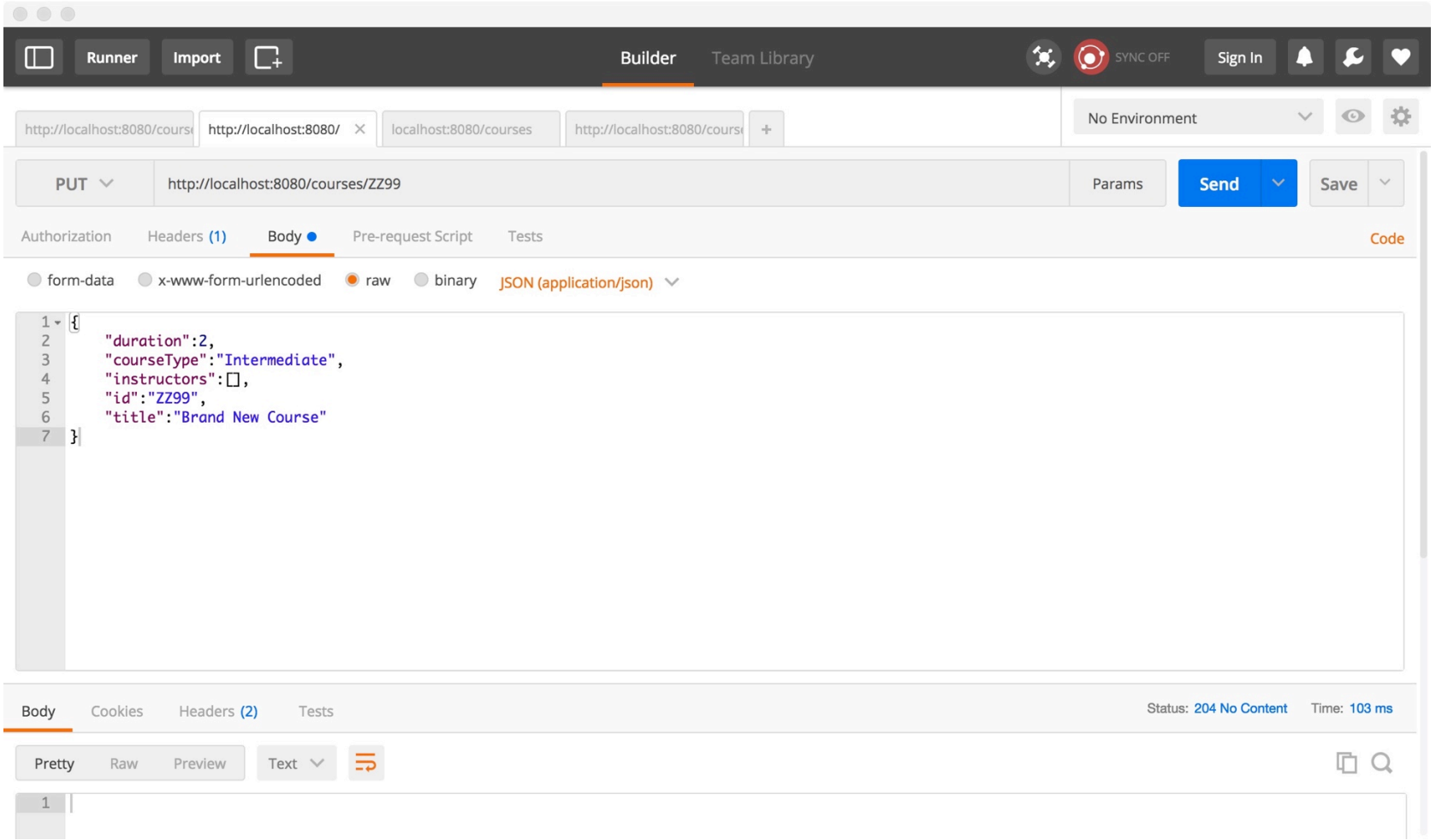
Testing Via Postman (Finding All Courses)



Testing Via Postman (Finding a Single Course)

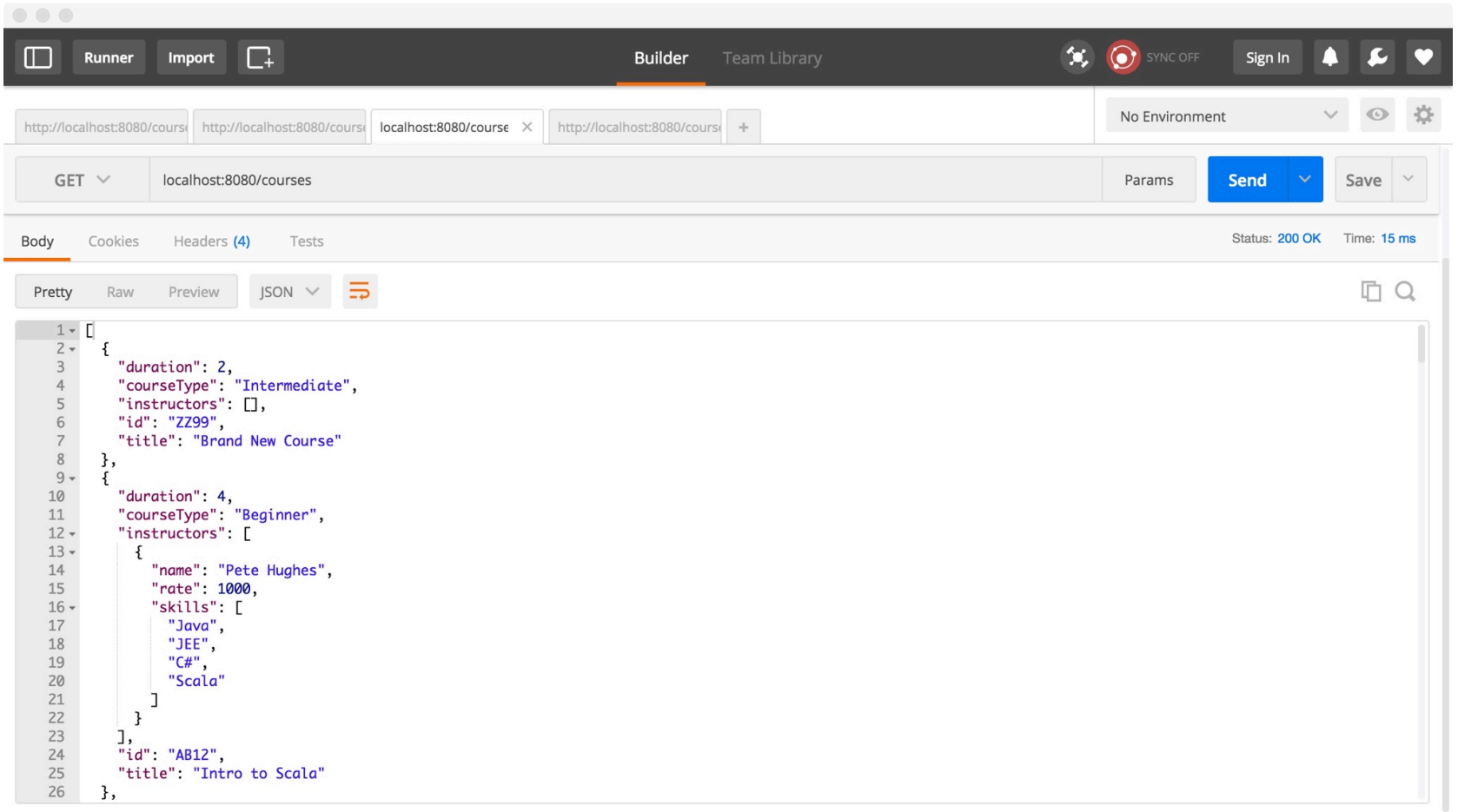


Testing Via Postman (Adding Courses)



INSTIL

Testing Via Postman (Adding Courses)



INSTIL

Level Three

- Tests, tests, tests…

The Limits of TDD & BDD

TDD and BDD are brilliant methodologies

- But they leave at least one question unresolved

Explicit tests can't address the 'unknown unknowns'

- More correctly called edge or corner cases

Reports that say that something hasn't happened are always interesting to me, because as we know, there are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns – the ones we don't know we don't know.

Donald Rumsfeld

Introducing Properties

Properties are simply parameterized tests

- Where actual values are replaced with placeholders

A property is the generalization of a test

- We make our tests as close to specifications as possible
- We express logical truths that should apply to all inputs

To take a simple example:

- “If we add ‘ab’, ‘cd’ and ‘ef’ to a collection then its size is 3”
- “If we add *n* items to a collection then its size is *n*”
- The former is a test whereas the latter is a property

Property Based Testing

In property based testing we instantiate many tests by replacing placeholders with random values

- We are trying to falsify the hypothesis that your property is a logical truth that applies to all inputs

Property based testing tools allow you to:

- Specify the number of test instances required
- Provide your own generators to create input values
- Combine properties with more conventional kinds of test

The merits of properties may include:

- A smaller and more maintainable test codebase
- A more complete spec with better code coverage

Examples of Tools

QuickCheck was the original property based test tool

- However it is written in Haskell and so not widely known

ScalaCheck is the Scala rewrite

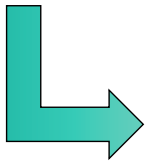
- At the moment it is the best known tool of its kind
- It provides a wide range of additional features

Similar tools now exist on other platforms

Language	Tool
F#	FsCheck
Python	Hypothesis
Ruby	Rantly
JavaScript	JSVerify, purescript-quickcheck

Hello World in ScalaCheck

```
object HelloScalaCheck {
  def main(args: Array[String]) {
    val prop = forAll { (no1: Int, no2: Int) =>
      printf("Checking with %d and %d\n",no1,no2)
      (no1 + no2) == (no2 + no1)
    }
    val params = Parameters.default.withMinSuccessfulTests(10)
    prop.check(params)
  }
}
```



```
Checking with 108643437 and 1550798728
Checking with 1773604856 and 1774391105
Checking with 2147483647 and 1413099849
Checking with -1996116831 and 2147483647
Checking with -1800499841 and 454421160
Checking with -1 and 1605504523
Checking with 172417505 and -279849306
Checking with -2147483648 and 1156729111
Checking with 1577680038 and 2147483647
Checking with -2147483648 and 0
+ OK, passed 10 tests.
```

Basic Testing with ScalaCheck

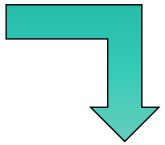
ScalaCheck integrates with other testing tools

- Within which it can use their support for matching
- E.g. the ‘Matchers’ trait and DSL in ScalaTest

To use ScalaCheck as a standalone tool

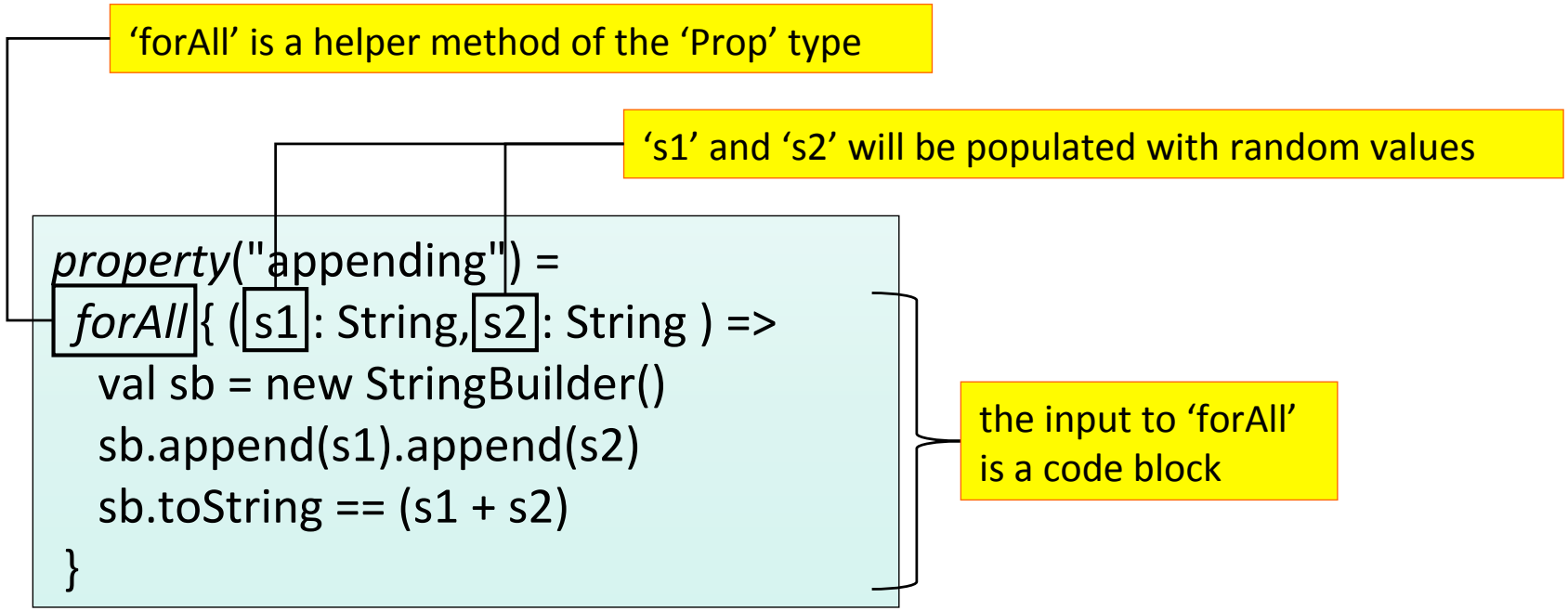
- Create a class which extends from ‘Properties’
  - The name of the spec is passed to the base constructor
- Define your properties via ‘property(foo) = Props.bar’
  - ‘foo’ represents the name of the property and ‘bar’ is the name of one of the factory methods of the ‘Props’ type
- ‘Props.forAll’ is the most commonly used factory method
  - The inputs are an optional generator and a code block
  - The result of the block determines if the property passes

```
object StringBuilderProps extends Properties("StringBuilder") {
  property("appending") =
    forAll { (s1: String, s2: String) =>
      val sb = new StringBuilder()
      sb.append(s1).append(s2)
      sb.toString == (s1 + s2)
    }
  property("inserting") =
    forAll { (s1: String, s2: String) =>
      val sb = new StringBuilder()
      sb.insert(0,s1).insert(0,s2)
      all(
        "content" |: sb.toString =? (s2 + s1),
        "length"  |: sb.toString.length =? (s2.length + s1.length)
      )
    }
  property("reversal") =
    forAll(alphaStr) { s: String =>
      (s.length > 1) ==> {
        val sb = new StringBuilder()
        sb.append(s)
        sb.toString != sb.reverse.toString
      }
    }
}
```

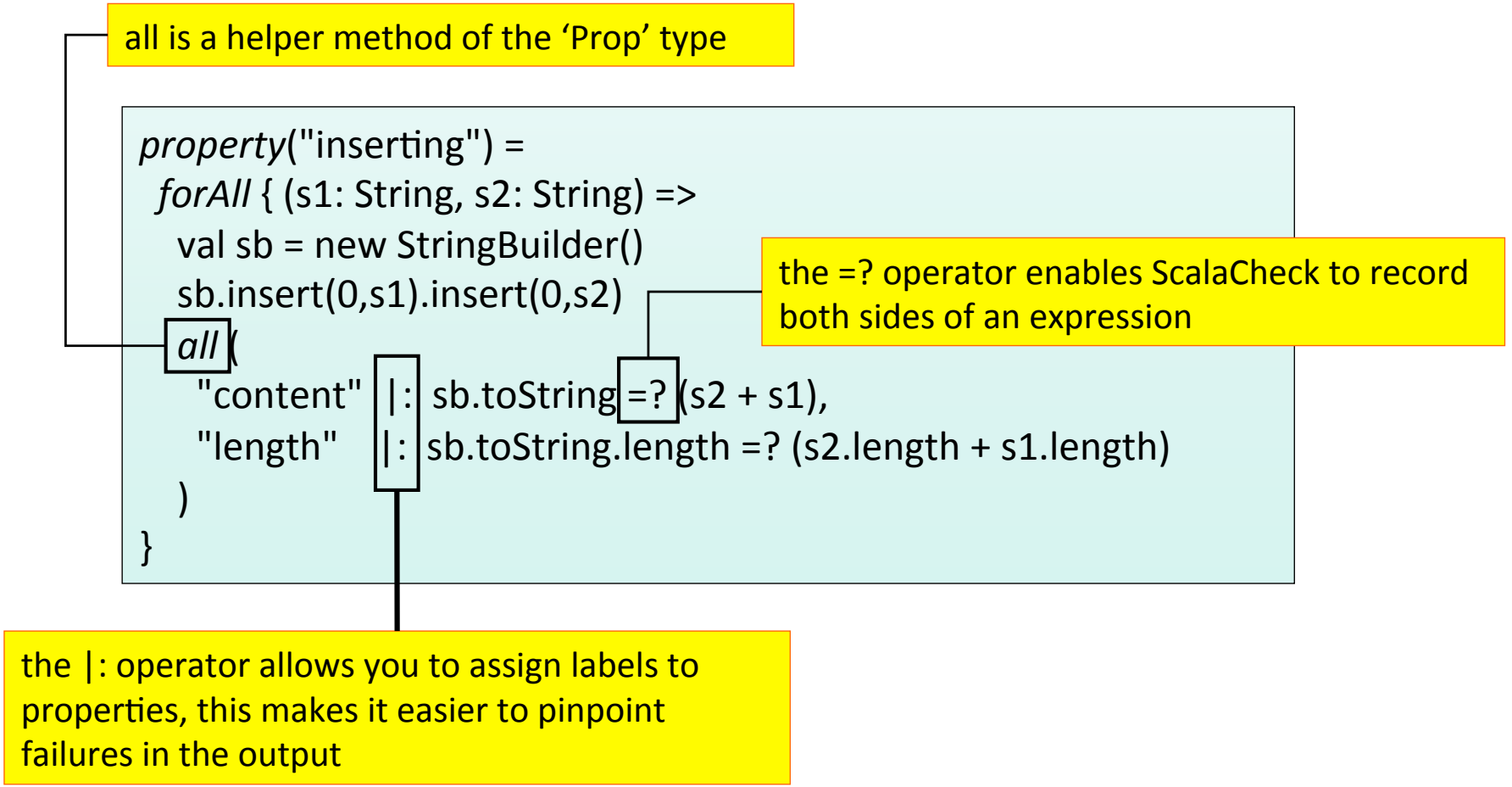


```
+ StringBuilder.appending: OK, passed 100 tests.
+ StringBuilder.inserting: OK, passed 100 tests.
! StringBuilder.reversal: Falsified after 3 passed tests.
> ARG_0: "xx"
> ARG_0_ORIGINAL: "xsx"
```

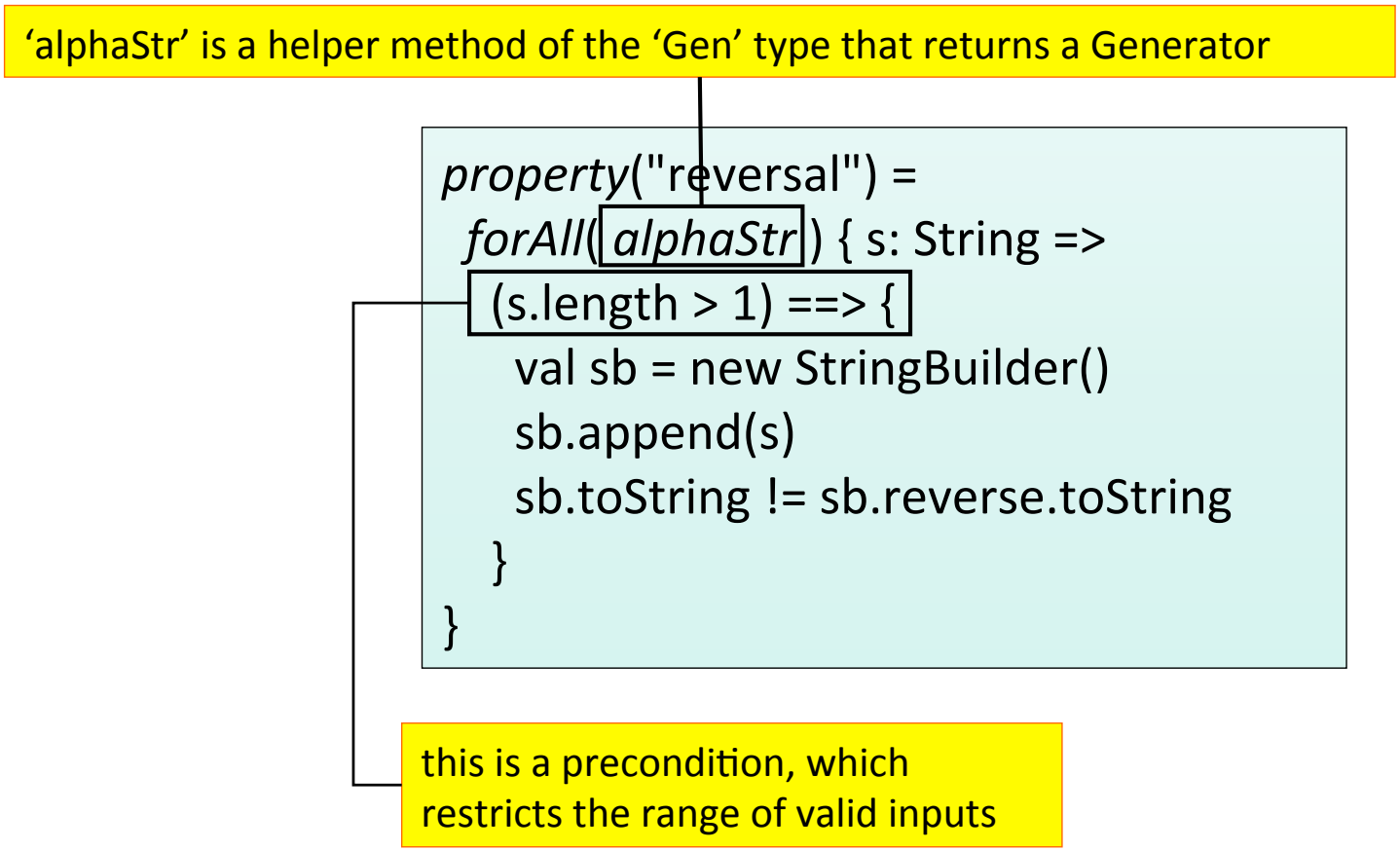
Property Declarations in Depth



Property Declarations in Depth



Property Declarations in Depth



ScalaCheck with ScalaTest

```
class StringBuilderTest extends FunSuite with Checkers {
  test("appending is supported") {
    check(
      forAll { (s1: String, s2: String) =>
        val sb = new StringBuilder()
        sb.append(s1).append(s2)
        sb.toString == (s1 + s2)
      })}
  test("inserting is supported") {
    check(
      forAll { (s1: String, s2: String) =>
        val sb = new StringBuilder()
        sb.insert(0,s1).insert(0,s2)
        all( "content" |: sb.toString =? (s2 + s1),
             "length"  |: sb.toString.length =? (s2.length + s1.length))
      })}
}
```

A trait to support ScalaCheck

A standard ScalaTest base class

Simplifying Test Specifications

ScalaCheck does not stop when a property fails

- Instead it examines the failing input to see if there is a subset which will still invalidate the property
- It will reduce the contents of a list, the size of a string, the variation in a range of numbers etc…

Both the original and simplified inputs are printed

- The original input is labelled ‘ARG_N_ORIGINAL’
- The simplified input is labelled ‘ARG_N’

You can add this feature to your own generators

- By providing an implementation of the ‘Shrink’ type with a ‘shrink’ method that returns a stream of simplified inputs

Controlling How ScalaCheck Runs

ScalaCheck defaults to 100 runs per property

- In many cases this will not be sufficient
- You can increase it to any number you like

This is one of many settings your can alter

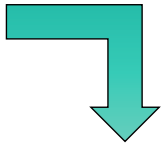
- Other examples are the ratio between discarded and successful tests, the number of worker threads and which random number generator to use

These settings can be changed by

- Creating and injecting a custom ‘Parameters’ object
- Specifying options on the applications command line
- Specifying options within an ‘sbt’ or ‘Maven’ build file

Controlling How ScalaCheck Runs

```
object StringBuilderPropsRedux {
  def main(args:Array[String]) {
    val params = new Parameters.Default {
      override val minSuccessfulTests = 10000
    }
    val prop = forAll(alphaStr) { s: String =>
      (s.length > 1) ==> {
        val sb = new StringBuilder()
        sb.append(s)
        sb.toString != sb.reverse.toString
      }
    }
    prop.check(params)
  }
}
```



```
! Falsified after 112 passed tests.
> ARG_0: "pp"
> ARG_0_ORIGINAL: "pxp"
```

Hints for Writing Properties

Properties can be hard to write at first

- Because we are used to thinking in terms of sample inputs

Some possible strategies are

- Express relationships between different outputs
  - Timestamps should increase, search results should be ranked by popularity, a GET should not succeed after a DELETE etc…
- Perform round trip checking
  - E.g. a message which has been encrypted and then decrypted should have the same hash as the original
- Use a reference implementation
  - When a result can be found by brute force or a 3rd party implementation you can use this to validate your work

Methods of the ‘Prop’ Type

Most properties are declared via ‘Prop.forAll’

- But other factory methods are available

There are also constant properties

- Such as ‘proved’, ‘passed’ and ‘falsified’

Factory Method	Description
forAll	Acts as the universal qualifier i.e. a condition is true for all inputs within a set (integers, positive integers, string etc…)
exists	Acts as the existential qualifier i.e. a condition is true for at least one input. ScalaCheck stops at the first successful result
throws	Returns true if an expression throws an exception of type ‘T’
all	Combines multiple properties such that all must validate
atLeastOne	Combines multiple properties such that one or more validates
within	Wraps a property to ensure it completes within a time limit

The Standard Generators

Normally the set of inputs will need to be restricted

- Allowing any integer, float or string will not be acceptable

You can specify a generator object when creating properties

- The ‘Gen’ class provides a wide range of factory methods
- Generators can be nested, combined and used in sequences

Factory Method	Description
choose	Returns an integer in the specified inclusive range
posNum, negNum	Return random positive and negative integers
alphaChar, alphaUpperChar	Return random characters
alphaStr, identifier	Return different kinds of string
const	An implicit method that wraps a value as a generator
listOf, nonEmptyListOf, listOfN	Take other generators and use then to produce lists of values

```
object UsingGenerators {
  def main(args:Array[String]): Unit = {
    demoFrequency
    demoListOf
    demoSequence
    demoConst
  }
  def demoFrequency(): Unit = {
    val gen = frequency(
      (9,choose(1,99)),
      (1,choose(100,500))
    )
    runDemo("The Frequency Generator",gen,100)
  }
  def demoListOf(): Unit = {
    val gen1 = listOf(choose(1,100))
    val gen2 = listOfN(5,choose(10,99))
    runDemo("The List Of Generator", gen1,10)
    runDemo("The List Of N Generator", gen2,10)
  }
```

```
  def demoSequence(): Unit = {
    val gen = sequence(List(choose(10,99),choose(100,999),const("Fred")))
    runDemo("The Sequence Generator", gen,10)
  }
  def demoConst(): Unit = {
    val gen = const(7)
    runDemo("The Const Generator", gen,10)
  }
  def runDemo(title: String, generator: Gen[Any], numTests: Int): Unit = {
    println(title)
    val prop = forAll(generator) { input: Any =>
      printf("\t%s\n",input)
      true
    }
    Test.check(Parameters.default.withMinSuccessfulTests(numTests),prop)
  }
}
```

The Frequency Generator

38
52
2
22
24
30
217
43
49

Other values omitted

The List Of N Generator

List(91, 82, 94, 40, 47)
List(74, 84, 38, 54, 43)
List(63, 86, 22, 23, 96)
List(57, 98, 55, 22, 41)
List(17, 25, 23, 28, 20)
List(29, 40, 86, 10, 56)
List(97, 37, 50, 82, 23)
List(92, 85, 76, 89, 19)
List(22, 23, 39, 41, 94)
List(61, 95, 12, 33, 61)

The Sequence Generator

[31, 671, Fred]
[63, 380, Fred]
[79, 369, Fred]
[61, 312, Fred]
[24, 721, Fred]
[87, 749, Fred]
[68, 799, Fred]
[37, 691, Fred]
[14, 147, Fred]
[22, 230, Fred]

The Const Generator

7
7
7
7
7
7
7
7
7
7

The List Of Generator

Some lists truncated

List()
List()
List(12, 35, 7, 67, 14, 38, 3, 31, 94, 63, 71, 99, 24, 43, 93, 88, 55, 89, 79)
List(53, 100, 13, 12, 29, 51, 42, 77, 62, 99, 92, 98, 24, 57, 47, 21, 49, 53, 60)
List(87, 72, 28)
List(46)
List(22, 73, 33, 37, 57, 59, 26, 69, 39, 53, 1, 16, 18, 15, 37, 77, 7, 75, 36, 31, 54, 7, 86, 35)
List(10, 65, 29, 14, 75, 93, 58, 48, 43, 18, 55, 54, 1, 50, 75, 5, 44, 99, 35, 79, 25, 25, 82, 14, 16, 15, 82, 84, 84)
List(100, 35, 86, 40)
List(54, 38, 66, 87, 61, 45, 79, 15, 31, 66, 38, 58, 59, 45, 85, 30, 54, 56, 87, 3, 10, 5, 98, 57, 59, 66, 52, 51)

A More Complex Example

We need a program that recognises poker hands

- So given “3H JH 4D 3S JC” it should output “two pair”

To test this using properties we need to:

- Create a custom generator for poker hands
- Raise the ‘minimum successful test’ parameter
- Write properties to validate the identification of hands

The custom generator can be created by:

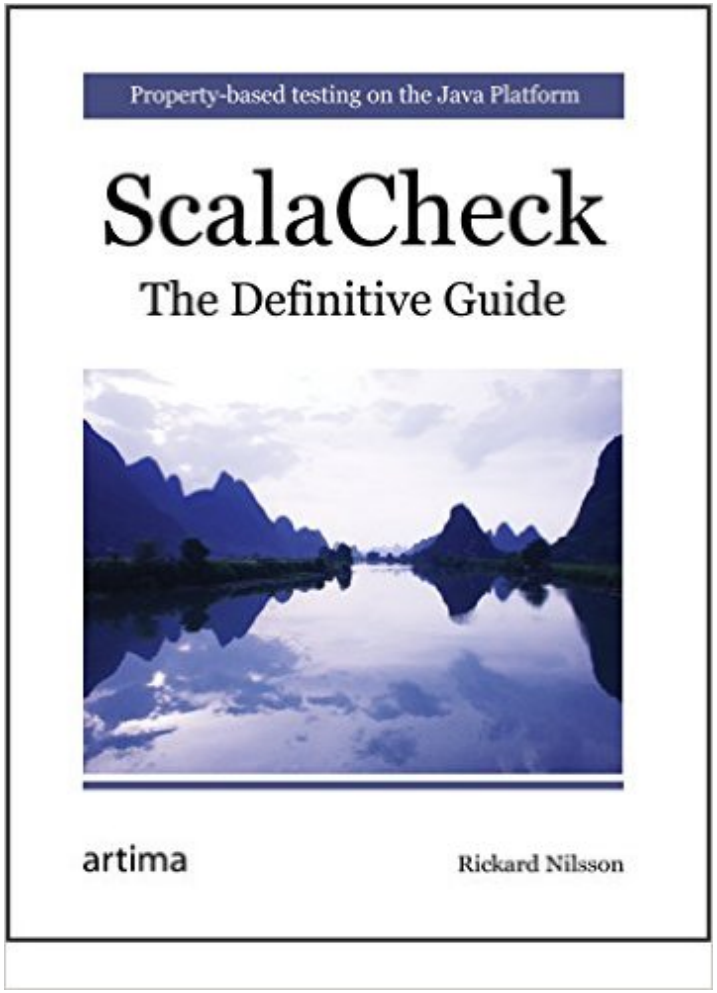
- Using ‘Gen.oneOf’ to pick a rank and a suit
- Then using a for comprehension to yield a card
- Then using ‘Gen.listOfN’ to produce a 5 card hand
  - ‘suchThat’ or ‘filter’ will prevent hands with duplicate cards

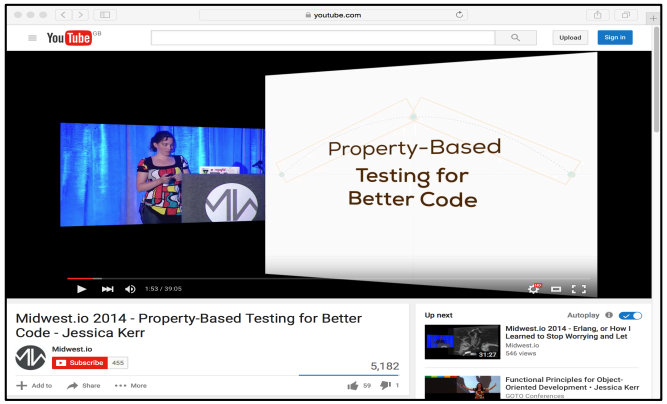
```
class PokerHandGenSpec extends FunSuite {

 def buildHandGen() = {
  val rankGen = Gen.oneOf(List("A", "Q", "K", "J", "10", "9", "8", "7", "6", "5", "4", "3", "2"))
  val suitGen = Gen.oneOf(List("C", "H", "S", "D"))
  val cardGen = for {
    rank <- rankGen
    suit <- suitGen
  } yield rank + suit

  Gen.listOfN(5, cardGen)            //create a list of 5 cards
     .suchThat(_.distinct.size == 5)  //remove hands with duplicated cards
     .map(PokerHand.apply)            //convert cards to poker hands
 }

 def assertProp(prop: Prop) {
  val params = Parameters.default.withMinSuccessfulTests(50000)
  val res = Test.check(params, prop)
  assert(res.passed)
 }

 val handGen = buildHandGen()
```

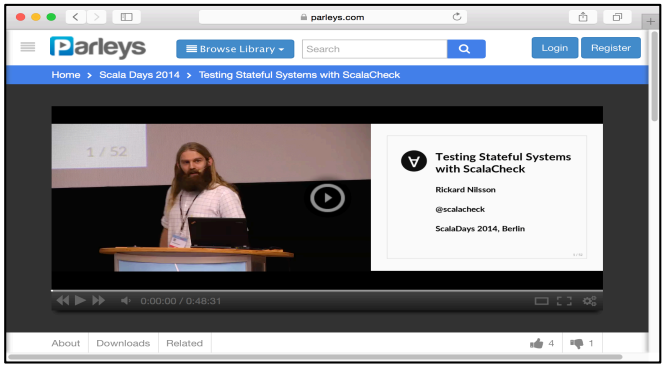
```
test("Every poker hand is present") {
  val props = List(
    exists(handGen)(_.isPair),
    exists(handGen)(_.isTwoPair),
    exists(handGen)(_.isThreeOfAKind),
    exists(handGen)(_.isStraight),
    exists(handGen)(_.isFlush),
    exists(handGen)(_.isFullHouse),
    exists(handGen)(_.isRoyalFlush),            //may require more hands
    exists(handGen)(_.isFourOfAKind))
    props.foreach(assertProp(_))
}
test("All 2 Pairs are Pairs") {
  val prop = forAll(handGen)(h => if (h.isTwoPair) h.isPair else true)
  assertProp(prop)
}
test("All Full Houses are Pairs and Three Of A Kind") {
  val check = (h: PokerHand) => if (h.isFullHouse) (h.isPair && h.isThreeOfAKind) else true
  val prop = forAll(handGen)(check)
  assertProp(prop)
}
```

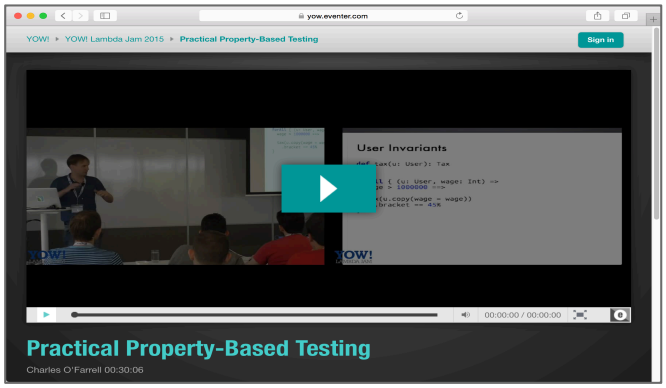
```
  test("All Three Of A Kind are Pairs") {
    val prop = forAll(handGen)(h => if (h.isThreeOfAKind) h.isPair else true)
    assertProp(prop)
  }

  test("All Four Of A Kind are Pairs and Three Of A KInd") {
    val check = (h: PokerHand) => if (h.isFourOfAKind) {
                                                    h.isPair && h.isThreeOfAKind
                                                } else { true }
    val prop = forAll(handGen)(check)
    assertProp(prop)
  }

  test("All Royal Flushes are Flushes") {
    val prop = forAll(handGen)(h => if (h.isRoyalFlush) h.isFlush else true)
    assertProp(prop)
  }

}
```

Resources









INSTIL

Summing Up

- What Floats Your Boat?

Where Do We All Stand?

Thank You - And Do Enjoy Your Party…