sorsix

Javaslang

Achieve functional eloquence in Java

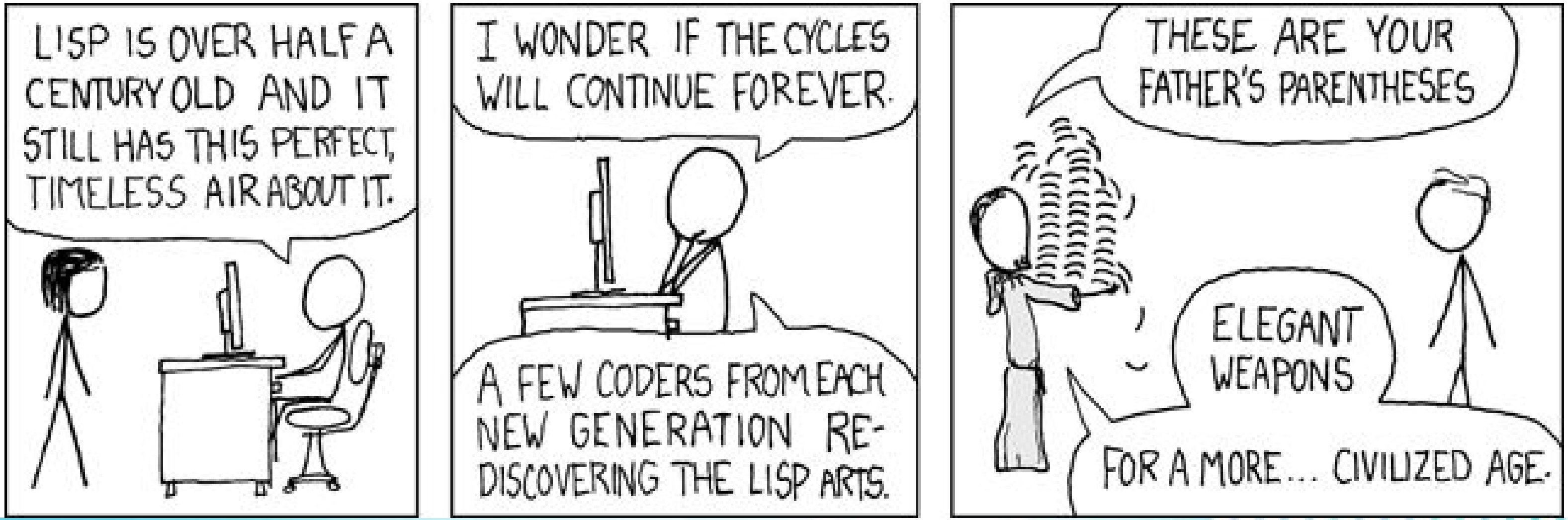
JAVASKOP

Blagoj Atanasovski | dev@Sorsix

What I was taught functional programming is:

```
(defun add-two-lists (a b c &optional (d c))
  (if a
    (add-two-lists
     (cdr a) (cdr b)
     (cdr (rplaca c (+ (car a) (car b)))) d) d))

(add-two-lists '(1 2 3 4 5) '(1 2 3 4 5) '(nil nil nil nil nil))
```


LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.
I WONDER IF THE CYCLES WILL CONTINUE FOREVER.
A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.
THESE ARE YOUR FATHER'S PARENTHESES
ELEGANT WEAPONS
FOR A MORE... CIVILIZED AGE.

Why functional programming?

```
class SomeClass {
 private boolean magicFlag;

 public boolean isMagicFlag() {
   return magicFlag;
 }

 public void updateSomething() { this.magicFlag = true;}

 public int doSomething() {
   return magicFlag ? 1 : 0;
 }
}
```

Why functional programming?

```
class SomeClass {
 private boolean magicFlag;

 public boolean isMagicFlag() {
   return magicFlag;
 }

 public void updateSomething() { this.magicFlag = true;}

 public int doSomething() {
   return magicFlag ? 1 : 0;
 }
}
```

What will doSomething() return?

Why functional programming?

- What kind of input am I getting?

Why functional programming?

- What kind of input am I getting?
  - Is it mutable?

Why functional programming?

- What kind of input am I getting?
  - Is it mutable?
  - Will someone outside of my code modify it?

Why functional programming?

- What kind of input am I getting?
  - Is it mutable?
  - Will someone outside of my code modify it?
  - Can I mutate it?

Why functional programming?

- What kind of input am I getting?
  - Is it mutable?
  - Will someone outside of my code modify it?
  - Can I mutate it?
  - Am I mutating it without knowing?

Why functional programming?

- What kind of input am I getting?
  - Is it mutable?
  - Will someone outside of my code modify it?
  - Can I mutate it?
  - Am I mutating it without knowing?
- How do I synchronize my code?

Why functional programming?

- What kind of input am I getting?
  - Is it mutable?
  - Will someone outside of my code modify it?
  - Can I mutate it?
  - Am I mutating it without knowing?
- How do I synchronize my code?
  - Is the input thread safe?

Why functional programming?

- What kind of input am I getting?
  - Is it mutable?
  - Will someone outside of my code modify it?
  - Can I mutate it?
  - Am I mutating it without knowing?
- How do I synchronize my code?
  - Is the input thread safe?
  - How can I be sure no race conditions occur?

Side-effects

- It’s fairly **easy** to write code in Java with side-effects
  - changing objects or variables in place
  - printing to the console
  - writing to a log file or to a database

Side-effects

- It’s fairly **easy** to write code in Java with side-effects
    - changing objects or variables in place
    - printing to the console
    - writing to a log file or to a database

- Not all side-effects are harmful
- Side-effects are considered **harmful** if they affect the semantics of our program in an undesirable way.

Side-effects

- It's fairly **easy** to write code in Java with side-effects
  - changing objects or variables in place
  - printing to the console
  - writing to a log file or to a database

- Not all side-effects are harmful
- Side-effects are considered **harmful** if they affect the semantics of our program in an undesirable way.

- If a function throws an exception => side-effect that affects our program
  - Exceptions are like **non-local goto-statements**
  - They break normal control-flow

Side-effects

- Real-world applications do perform side-effects.

```
int divide(int dividend, int divisor) {
 return dividend / divisor;
}
```

Side-effects

- Real-world applications do perform side-effects

```
int divide(int dividend, int divisor) {
 return dividend / divisor;
}
```

- Lets’ modify it a bit

```
Try<Integer> divide(int dividend, int divisor) {
 return Try.of(() -> dividend / divisor);
}
```

Side-effects

- Real-world applications do perform side-effects

```
int divide(int dividend, int divisor) {
 return dividend / divisor;
}
```

- Lets’ modify it a bit

```
Try<Integer> divide(int dividend, int divisor) {
 return Try.of(() -> dividend / divisor);
}
```

- This version of divide does not throw any exception anymore.
- We made the possible failure explicit by using the type Try.

Referential Transparency

- A function/expression, is called **referentially transparent** if a call can be replaced by its value without affecting the behavior of the program.

Referential Transparency

- A function/expression, is called **referentially transparent** if a call can be replaced by its value without affecting the behavior of the program.
- Given the same input the output is always the same.

```
        Math.random();              Math.max(1, 2);
```

Referential Transparency

- A function/expression, is called **referentially transparent** if a call can be replaced by its value without affecting the behavior of the program.
- Given the same input the output is always the same.

```
        Math.random();              Math.max(1, 2);
```

- A function is called **pure** if all expressions involved are referentially transparent.

Referential Transparency

- A function/expression, is called **referentially transparent** if a call can be replaced by its value without affecting the behavior of the program.
- Given the same input the output is always the same.

```
        Math.random();            Math.max(1, 2);
```

- A function is called **pure** if all expressions involved are referentially transparent.
- An application composed of pure functions will most probably just work if it compiles.
- We are able to reason about it. Unit tests are easy to write and debugging becomes a relict of the past.

Immutable Values

The key to better Java code is to use immutable values paired with referentially transparent functions.

Immutable Values

The key to better Java code is to use immutable values paired with referentially transparent functions.

Immutable values are:

- Inherently thread-safe

Immutable Values

The key to better Java code is to use immutable values paired with referentially transparent functions.

Immutable values are:

- Inherently thread-safe
  - do not need to be synchronized

Immutable Values

The key to better Java code is to use immutable values paired with referentially transparent functions.

Immutable values are:

- Inherently thread-safe
  - do not need to be synchronized
- Are stable regarding equals and hashCode

Immutable Values

The key to better Java code is to use immutable values paired with referentially transparent functions.

Immutable values are:

- Inherently thread-safe
  - do not need to be synchronized
- Are stable regarding equals and hashCode
  - are reliable hash keys

Immutable Values

The key to better Java code is to use immutable values paired with referentially transparent functions.

Immutable values are:

- Inherently thread-safe
  - do not need to be synchronized
- Are stable regarding equals and hashCode
  - are reliable hash keys
- Do not need to be cloned

Why we need Javaslang

- Java 8 brought a lot of changes, but

Why we need Javaslang

JAVASLANG

- Java 8 brought a lot of changes, but
  - No functional data structures

Why we need Javaslang

JAVASLANG

- Java 8 brought a lot of changes, but
  - No functional data structures
  - No currying, partial application, memoization, lifting

Why we need Javaslang

- Java 8 brought a lot of changes, but
  - No functional data structures
  - No currying, partial application, memoization, lifting
  - No Tuples

Why we need Javaslang

JAVASLANG

- Java 8 brought a lot of changes, but
  - No functional data structures
  - No currying, partial application, memoization, lifting
  - No Tuples
  - Lack of Stream/Optional in existing APIs

Why we need Javaslang

JAVASLANG

- Java 8 brought a lot of changes, but
  - No functional data structures
  - No currying, partial application, memoization, lifting
  - No Tuples
  - Lack of Stream/Optional in existing APIs
  - No checked exceptions in lambdas

Why we need Javaslang

JAVASLANG

- Java 8 brought a lot of changes, but
  - No functional data structures
  - No currying, partial application, memoization, lifting
  - No Tuples
  - Lack of Stream/Optional in existing APIs
  - No checked exceptions in lambdas
  - Failure handling (Try, Either)

Why we need Javaslang

- Java 8 brought a lot of changes, but
  - No functional data structures
  - No currying, partial application, memoization, lifting
  - No Tuples
  - Lack of Stream/Optional in existing APIs
  - No checked exceptions in lambdas
  - Failure handling (Try, Either)
  - list.stream().map(...).collect(toList())

Why we need Javaslang

JAVASLANG

- Java 8 brought a lot of changes, but
  - No functional data structures
  - No currying, partial application, memoization, lifting
  - No Tuples
  - Lack of Stream/Optional in existing APIs
  - No checked exceptions in lambdas
  - Failure handling (Try, Either)
  - list.stream().map(...).collect(toList())
  - ...

- **Javaslang** was created by Daniel Dietrich and first released in 2013. It leverages Java 8's lambdas to create various new features based on functional patterns

Does Java have immutable collections?

```
List<String> underlying = new ArrayList<>();
underlying.add("1","2");
List<String> list = Collections.unmodifiableList(underlying);
```

Does Java have immutable collections?

```java
List<String> underlying = new ArrayList<>();
underlying.add("1","2");
List<String> list = Collections.unmodifiableList(underlying);

underlying.add("3");
assert list.size() != underlying.size(); // What will happen?
```

Functional Data Structures

```
javaslang.collection.List<User> users = List.of(
   new User("1", "1@mail"),
   new User("2", "2@mail"));

// users is immutable
users.push(new User("3", "3@email"));
assert users.size() == 2; // It will pass
```

Functional Data Structures

```
javaslang.collection.List<User> users = List.of(
   new User("1", "1@mail"),
   new User("2", "2@mail"));

// users is immutable
users.push(new User("3", "3@email"));
assert users.size() == 2; // It will pass

users = users.push(new User("3", "3@email"));

users
   .map(User::getEmail)
   .toSet()
   .forEach(emailService::sendWelcomeEmailTo);
```

Partial application

```
// (template, user) => Contents
Function2<String, User, String> emailTxt =
    (template,user) -> template.replace("_user_", user.getName());
```

Partial application

```
// (template, user) => Contents
Function2<String, User, String> emailTxt =
    (template,user) -> template.replace("_user_", user.getName());

String emailTemplate = "Hello _user_";
```

Partial application

```
// (template, user) => Contents
Function2<String, User, String> emailTxt =
    (template,user) -> template.replace("_user_", user.getName());

String emailTemplate = "Hello _user_";

// (user) => "Hello " + user.getUserName()
Function<User, String> contentForUser = emailTxt.apply(emailTemplate);
```

Partial application

```
// (template, user) => Contents
Function2<String, User, String> emailTxt =
    (template,user) -> template.replace("_user_", user.getName());

String emailTemplate = "Hello _user_";

// (user) => "Hello " + user.getUserName()
Function<User, String> contentForUser = emailTxt.apply(emailTemplate);

users.filter(x -> Objects.nonNull(x.getName()))
```

Partial application

```
// (template, user) => Contents
Function2<String, User, String> emailTxt =
    (template,user) -> template.replace("_user_", user.getName());

String emailTemplate = "Hello _user_";

// (user) => "Hello " + user.getUserName()
Function<User, String> contentForUser = emailTxt.apply(emailTemplate);

users.filter(x -> Objects.nonNull(x.getName()))
   .forEach(user -> emailService.sendEmail(
       user.getEmail(),
       contentForUser.apply(user)));
```

Tuples

- Easily create tuples of length 1 to 8
  - `Tuple.of(1, “two”, Option.empty())`

Tuples

- Easily create tuples of length 1 to 8
  - `Tuple.of(1, “two”, Option.empty())`

```
List<Status> statuses = users.map(user ->
        emailService.sendEmail(
            user.getEmail(),
            contentForUser.apply(user.getName())));
```

Tuples

- Easily create tuples of length 1 to 8
  - `Tuple.of(1, “two”, Option.empty())`

```
List<Status> statuses = users.map(user ->
       emailService.sendEmail(
          user.getEmail(),
          contentForUser.apply(user.getName())));

List<Tuple2<User, Status>> mailStatusForUser = users.zip(statuses);
// Status = OK | NOT_OK
```

Checked Functions

- Lambdas in Java8 can’t throw checked exceptions

```
// Compiler error
Supplier<InputStream> inSupplier = socket::getInputStream;
```

Checked Functions

- Lambdas in Java8 can’t throw checked exceptions

```
// Compiler error
Supplier<InputStream> inSupplier = socket::getInputStream;
```

- Javaslang provides checked functions

```
CheckedFunction0<BufferedReader> readerSupplier =
   CheckedFunction0.of(socket::getInputStream)
        .andThen(InputStreamReader::new)
        .andThen(BufferedReader::new);
```

Checked Functions

- Lambdas in Java8 can’t throw checked exceptions

```
// Compiler error
Supplier<InputStream> inSupplier = socket::getInputStream;
```

- Javaslang provides checked functions

```
CheckedFunction0<BufferedReader> readerSupplier =
   CheckedFunction0.of(socket::getInputStream)
       .andThen(InputStreamReader::new)
       .andThen(BufferedReader::new);

try {
 readerSupplier.apply();
} catch (Throwable throwable) {
 // do something
}
```

Error Handling

- Checked functions can be composed in a clean way
- But there is an even more elegant solution.

Error Handling

- Checked functions can be composed in a clean way
- But there is an even more elegant solution. Instead of:

```
CheckedFunction0<InputStream> inCheckedSuplier =
   CheckedFunction0.of(socket::getInputStream);

try {
 inCheckedSuplier.apply();
} catch (Throwable throwable) {
 // do something
}
```

Error Handling

- Checked functions can be composed in a clean way
- But there is an even more elegant solution. Instead of:

```
CheckedFunction0<InputStream> inCheckedSuplier =
   CheckedFunction0.of(socket::getInputStream);

try {
 inCheckedSuplier.apply();
} catch (Throwable throwable) {
 // do something
}
```

- We could just do

```
Try<BufferedReader> readerTry = Try.of(socket::getInputStream)
   .map(InputStreamReader::new)
   .map(BufferedReader::new);
```

How we usually do it

```
@RequestMapping("/person/{name}")
public ResponseEntity<?> find(String name) {

}
```

How we usually do it

```
@RequestMapping("/person/{name}")
public ResponseEntity<?> find(String name) {
 if (!validate(name)) {
   return ResponseEntity.badRequest().body("request not valid");
 }
}
```

How we usually do it

```
@RequestMapping("/person/{name}")
public ResponseEntity<?> find(String name) {
 if (!validate(name)) {
   return ResponseEntity.badRequest().body("request not valid");
 }

 Person somePerson = this.someService.find(name);
}
```

How we usually do it

```
@RequestMapping("/person/{name}")
public ResponseEntity<?> find(String name) {
 if (!validate(name)) {
   return ResponseEntity.badRequest().body("request not valid");
 }

 Person somePerson = this.someService.find(name);
 return somePerson == null ?
     ResponseEntity.notFound().build()
}
```

How we usually do it

```
@RequestMapping("/person/{name}")
public ResponseEntity<?> find(String name) {
 if (!validate(name)) {
   return ResponseEntity.badRequest().body("request not valid");
 }

 Person somePerson = this.someService.find(name);
 return somePerson == null ?
     ResponseEntity.notFound().build()
     : ResponseEntity.ok(somePerson);
}
```

How we usually do it

```
@RequestMapping("/person/{name}")
public ResponseEntity<?> find(String name) {
 if (!validate(name)) {
   return ResponseEntity.badRequest().body("request not valid");
 }

 Person somePerson = this.someService.find(name);
 return somePerson == null ?
     ResponseEntity.notFound().build()
     : ResponseEntity.ok(somePerson);
}
   // What if someService.find throws some exception?
   // What if validate throws some exception?
```

Let’s play with it

```
@RequestMapping("/find/{name}")
public ResponseEntity<?> find(String name) {
 return validate(name) // returns Either<Throwable, String>
}
```

Let’s play with it

```java
@RequestMapping("/find/{name}")
public ResponseEntity<?> find(String name) {
 return validate(name) // returns Either<Throwable, String>
     .flatMap(this.someService::find) // only if validation passed
}

// SomeService.find returns Either a person (correct result)
// or a Throwable on an Error
public Either<Throwable, Person> find(String name)
```

Let’s play with it

```
@RequestMapping("/find/{name}")
public ResponseEntity<?> find(String name) {
 return validate(name) // returns Either<Throwable, String>
     .flatMap(this.someService::find) // only if validation passed
     .fold(this::getResponseOnError, ResponseEntity::ok);
       //always returns a response
}
```

Some of us have to work with InputStreams

```
String getContent(String location) throws IOException {
 try {
   final URL url = new URL(location);
   if (!"http".equals(url.getProtocol())) {
     throw new UnsupportedOperationException("Protocol is not http");
   }

   final URLConnection con = url.openConnection();
   final InputStream in = con.getInputStream();
   return readAndClose(in);
 } catch(Exception x) {
   throw new IOException("Error loading location " + location, x);
 }
}
```

Let’s fix it

```
Try<String> getContentT(String location) {
 return Try
     .of(() -> new URL(location))

}
```

Let’s fix it

```
Try<String> getContentT(String location) {
 return Try
     .of(() -> new URL(location))
     .filter(url -> "http".equals(url.getProtocol()))

}
```

Let’s fix it

```
Try<String> getContentT(String location) {
 return Try
     .of(() -> new URL(location))
     .filter(url -> "http".equals(url.getProtocol()))
     .flatMap(url -> Try.of(url::openConnection))

}
```

Let’s fix it

```
Try<String> getContentT(String location) {
 return Try
     .of(() -> new URL(location))
     .filter(url -> "http".equals(url.getProtocol()))
     .flatMap(url -> Try.of(url::openConnection))
     .flatMap(con -> Try.of(con::getInputStream))

}
```

Let’s fix it

```
Try<String> getContentT(String location) {
 return Try
     .of(() -> new URL(location))
     .filter(url -> "http".equals(url.getProtocol()))
     .flatMap(url -> Try.of(url::openConnection))
     .flatMap(con -> Try.of(con::getInputStream))
     .map(this::readAndClose);
}
```

Conclusion

- Right tool for the right job

Conclusion

- Right tool for the right job
- We’re still struggling with proper use of OOP

Conclusion

- Right tool for the right job
- We’re still struggling with proper use of OOP
- Pure functional programming is hard

Conclusion

- Right tool for the right job
- We’re still struggling with proper use of OOP
- Pure functional programming is hard
- Javaslang offers good functional patterns
- And we can combine them with our Java code
    - When we need them, if we need them

Conclusion

- Right tool for the right job
- We're still struggling with proper use of OOP
- Pure functional programming is hard
- Javaslang offers good functional patterns
- And we can combine them with our Java code
  - When we need them, if we need them
- It offers a lot more than I talked about here

Conclusion

- Right tool for the right job
- We're still struggling with proper use of OOP
- Pure functional programming is hard
- Javaslang offers us good functional patterns
- And we can combine them with our Java code
    - When we need them, if we need them
- It offers a lot more than I talked about here
- So please check it out
    - http://www.javaslang.io/
    - https://github.com/javaslang/javaslang

Thank you!
Questions?

About Sorsix

SORSIX International is an established Australian IT company with offices in Australia, USA, Macedonia and Serbia. We build mission-critical systems for finance, telecommunications and healthcare. Our systems keep planes flying, banks working, and phones connected. Ten million people live and prosper on our healthcare platform, spanning three countries. Sorsix believes in building systems that never go down because lives and businesses depend on them.