

BEYOND TRY-CATCH

Exception Handling in Java

`sam.wang@agfa.com`

Shanghai

October 22, 2017

Outline

Plain Old Try

- Checked Exception

- Unchecked Exception

- Try-Catch Sample

Functional Programming

- Algebraic Data Type (ADT)

- Design Pattern in Functional Programming

- Better Way in Java 8

- Problems with Optional

Monad Try in Action

- Try

- Failure

- Success

- Try Application

- Improve Try

- TryBuilder

- TryBuilder Application

Checked Exception

Definition

Checked at compile time. If the code in a method throws a checked exception, then the method must either **1.handle the exception** or it must specify the exception **2.using throws** keyword.

Example

- ▶ `java.io.FileNotFoundException`
- ▶ `java.lang.InterruptedException`
- ▶ `java.sql.SQLException`

Not a best practice

Not implemented by modern languages later than Java

Unchecked Exception

Definition

Not checked at compiled time. Exceptions under **Error** and **RuntimeException** classes are unchecked exceptions, everything else under throwable is checked.

Example

- ▶ `java.lang.NullPointerException`
- ▶ `java.lang.ArrayIndexOutOfBoundsException`
- ▶ `java.lang.OutOfMemoryError`

Try-Catch Sample

PatientList find(String strAge, String strCreated)

```
1 // Process input age
2 int age = -1;
3 try {
4     age = Integer.parseInt(strAge);
5 } catch (NumberFormatException e) {
6     throw new IllegalArgumentException("Invalid_age...");
7 }
8 // Process input date
9 Date created = null;
10 try {
11     created = dateFormat.parse(strCreated);
12 } catch (ParseException e) {
13     throw new IllegalArgumentException("Invalid_create_date...");
14 }
15 // Major logic starts here
16 age = age & 0x7F;
17 created = created.later(today) ? today : created;
18 patientService.search(age, created);
```

Algebraic Data Type (ADT)

λ Calculus

1. $Void \rightarrow 0, Unit, () \rightarrow 1, f(\alpha) \rightarrow \beta \rightarrow \beta_{size}^{\alpha_{size}}$
2. $A|B \rightarrow A_{size} + B_{size}$
 $WorkDay | Weekend \rightarrow Week (5 + 2 = 7)$
3. $(A, B) \rightarrow A_{size} \cdot B_{size}$
 $Tuple [WorkDay, Weather] \rightarrow WorkDay_{days} \cdot Weather_{kinds}$

Example

LinkedList: $Nil | (a, (List(a')))) \rightarrow \theta(a) = 1 + a \cdot \theta(a')$

$\frac{1}{1-x} \rightarrow 1 + x + x^2 + x^3 \dots \rightarrow$ a list is either empty or containing a single element, or two elements, or three ...

BinaryTree: $Nil | Node(a, Tree(a), Tree(a)) \rightarrow \theta(a) = 1 + a \cdot \theta(a')^2$

$\frac{1 - \sqrt{1-4x}}{2x} \rightarrow 1 + a + 2 \cdot a^2 + 5 \cdot a^3 \dots \rightarrow$ a binary tree is either empty or containing a value of type a, or two values of type a in two ways, or three values of type a in five different ways ...

Design Pattern in Functional Programming

Functor

Apply a function to a wrapped value. $m\ a \rightarrow (a \rightarrow b) \rightarrow m\ b$

Optional [T] :: *map*($f : T \rightarrow U$) : *Optional* [U]

Stream [T] :: *map*($f : T \rightarrow U$) : *Stream* [U]

Applicatives

Apply a wrapped function to a wrapped value

$m\ a\ \langle * \rangle\ n\ b \rightarrow Just\ (a.apply\ b)$

Monad

Apply a function that returns a wrapped value, to a wrapped value. $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

Optional [T] :: *flatMap*($f : T \rightarrow Optional\ [U]$) : *Optional* [U]

Stream [T] :: *flatMap*($f : T \rightarrow Stream\ [U]$) : *Stream* [U]

Better Way in Java 8

java.util.Optional

tryParseInt(s: String): Optional<Integer> \in *Integer*

tryParse(s: String): Optional<Date> \in *DateFormat*

find(String age, String created): Optional<PatientList>

Solution

Expression-oriented: one compact expression, no temporary variables and **composability**¹

```
1 Integer.parseInt(age).flatMap(a ->
2   dateFormat.tryParse(created).map(c -> {
3       // Only when both age and created have no problem
4       a = a & 0x7F;
5       c = c.later(today) ? today : c;
6       return patientService.search(a, c);
7   })
8 );
```

¹ $\beta = f(\alpha), \gamma = g(\beta) \rightarrow \gamma = g(f(\alpha))$

Problems with Optional

1. Not adopted by many existing APIs yet.
 - *Map* :: *get*(*key* : *String*) : *Object* → *null*
 - *String* :: *indexOf*(*ch* : *int*) : *int* → -1
2. Happy path only. Nowhere to get exception details
3. Ugly when too many nested λ
Example *opt1.flatMap(v1 → opt2.flatMap(v2 → opt3.flatMap(v3 → opt4.map(v4 → v1 + v2 + v3 + v4))))*);
4. Not able to get all validation results.

interface Try[R]

```
1 <T> Try<T> map(Function<R,T> f);
2 <T> Try<T> flatMap(Function<R, Try<T>> f);
3
4 // Resolve with no worry about error
5 void foreach(Consumer<R> callback);
6 // Resolve with a success callback and an error handling
7 void andThen(Consumer<R> callback ,
8             Consumer<Throwable> errorHandling);
9
10 // Resolve separately by two steps
11 Try<R> ifSuccess(Consumer<R> callback);
12 void orElse(Consumer<Throwable> errorHandling);
13 // Miscellaneous methods
14 Try<R> filter(Predicate<R> f);
15 Optional<R> toOption();
16 static <T> Try<T> tryWith(Block<T> s) {
17     try { return new Success(s.execute());
18     } catch (Throwable e) {
19         if (isFatal(e)) throw new RuntimeException(e);
20         else return new Failure(e);
21     }
22 }
```

final class Failure implements Try

```
1 private Throwable exception;  
2 Failure(Throwable exception) { this.exception = exception; }  
3  
4 public Try map(Function f) { return this; }  
5 public Try flatMap(Function f) { return this; }  
6  
7 public void forEach(Consumer callback) { return; }  
8 public void andThen(Consumer callback,  
9     Consumer errorHandling) {  
10     orElse(errorHandling);  
11 }  
12 public Try ifSuccess(Consumer callback) { return this; }  
13 public void orElse(Consumer errorHandling) {  
14     errorHandling.accept(exception);  
15 }  
16  
17 public Try filter(Predicate f) { return this; }  
18 public Optional toOption() { return Optional.empty(); }
```

final class Success[R] implements Try[R]

```
1 private R result;  
2 Success(R result) { this.result = result; }  
3 public <T> Try<T> map(Function<R, T> f) {  
4     return Try.tryWith(() -> f.apply(result));  
5 }  
6 public <T> Try<T> flatMap(Function<R, Try<T>> f) {  
7     Try<Try<T>> mapped = Try.tryWith(() -> f.apply(result));  
8     if (mapped.isSuccessful()) {  
9         return mapped.get();  
10    } else {  
11        return new Failure(mapped.exception());  
12    }  
13 }  
14 public void forEach(Consumer<R> callback) {  
15     Try.tryWith(() -> { callback.accept(result); ... });  
16 }  
17 public Try<R> ifSuccess(Consumer<R> callback) {  
18     return Try.tryWith(() -> { callback.accept(result); ... });  
19 }  
20 public void andThen(Consumer<R> callback,  
21     Consumer<Throwable> errorHandling) {  
22     ifSuccess(callback);  
23 }
```

Try Application

```
1 Try<Integer> readAge = tryWith(Integer.parseInt(age));
2 Try<Date> readDate = tryWith(dateFormat.parse(created));
3
4 BiFunction<Integer, Date, PatientList> search = (a,c) -> {
5     a = a & 0x7F;
6     c = c.later(today) ? today : c;
7     return patientService.search(a,c);
8 }
9
10 Try<PatientList> result = readAge.flatMap(a ->
11     readDate.map(c -> search.apply(a, c)));
12
13 result.forEach(patientList -> ... );
14
15 result.isSuccess(patientList -> ... ).orElse(exception -> ... );
16
17 result.andThen(
18     patientList -> ... ,
19     exception -> ...
20 );
```

Improve Try

1. Still ugly when too many *Try*s
2. No lazy evaluation (No need to continue readDate when readAge failed)
3. It would be nice to have:

```
1 for {a <- Integer.parseInt(age)
2     c <- dateFormat.parse(created)}
3   yield(patientService.search(
4     a & 0x7F,
5     c.later(today) ? today : c)
6   )
```

TryBuilder

final class TryBuilderN<R1, R2, ...RN>

```
1 private Block<R1> b1;
2 private Block<R2> b2;
3 ...
4 private Block<RN> bn;
5
6 static TryBuilderN tryN(Block<R1> b1, Block<R2> b2, ... Block<RN>
7     this.b1 = b1;
8     this.b2 = b2;
9     ...
10    this.bn = bn;
11 }
12
13 public <T> Try<T> yield(MultiFunction<R1, R2, ...RN, T> f) {
14     return
15         Try.tryWith(b1).flatMap(v1 ->
16             Try.tryWith(b2).flatMap(v2 ->
17                 ...
18                 Try.tryWith(bn).map(vn ->
19                     f.apply(v1, v2, ... vn)))));
20 }
```

TryBuilder Application

```
1 try2 (() -> Integer.parseInt(age),  
2      () -> dateFormat.parse(created))  
3  
4 .yield((a,c) -> {  
5     a = a & 0x7F;  
6     c = c.later(today) ? today : c;  
7     return patientService.search(a,c);})  
8  
9 .andThen(  
10     patientList -> ... ,  
11     exception -> ...  
12 );
```

Next Step

Monad Validation to support exception accumulation and return them once for all