BEYOND TRY-CATCH

Exception Handling in Java

sam.wang@agfa.com

Shanghai

October 18, 2017

Outline

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)

```
// Process input age
int age = -1;
try {
  age = Integer.parseInt(strAge);
} catch (NumberFormatException e) {
  throw new IllegalArgument("Invalid age...}

// Process input date
Date created = null;
try {
  created = dateFormat.parse(strCreated);
} catch (ParseException e) {
  throw new IllegalArgument("Invalid create date...);
}

// Major logic starts here
age = age & 0x7F;
created = created.later(today) ? today : created;
patientService.search(age, created);
```

Algebraic Data Type (ADT)

λ Calculus

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

Example

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

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

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

$\frac{1-\sqrt{1-4x}}{2x} \to 1 + a + 2 \cdot a^2 + 5 \cdot a^3 ... \to$ 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]

```
Integer.tryParseInt(age).flatMap(a ->
  dateFormat.tryParse(created).map(c -> {
    // Only when both age and created have no problem
    a = a & 0x7F;
    c = c.later(today) ? today : c;
    return patientService.search(a, c);
  })
);
```

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

Problems with Optional

1. Not adopted by many existing APIs *yet*.
   - $Map :: get(key : String) : Object \rightarrow null$
   - $String :: indexOf(ch : int) : int \rightarrow -1$
2. Happy path only. Nowhere to get exception details
3. Ugly when too many nested λ

   Example $opt1.flatMap(v1 \rightarrow opt2.flatMap(v2 \rightarrow opt3.flatMap(v3 \rightarrow opt4.map(v4 \rightarrow v1 + v2 + v3 + v4))));$
4. Not able to get all validation results.

Try

interface *Try* [*R*]

```
<T> Try<T> map(Function<R,T> f);
<T> Try<T> flatMap(Function<R, Try<T>> f);

// Resolve with no worry about error
void forEach(Consumer<R> callback);
// Resolve with a success callback and an error handling
void andThen(Consumer<R> callback, Consumer<Throwable> errorHandling);
// Resolve separately by two steps
Try<R> ifSuccess(Consumer<R> callback);
void orElse(Consumer<Throwable> errorHandling);
// Miscellaneous methods
Try<R> filter(Predicate<R> f);
Optional<R> toOption();

static <T> Try<T> tryWith(Block<T> s) {
  try { return new Success(s.execute());
  } catch (Throwable e) {
    if (isFatal(e)) throw new RuntimeException(e);
    else return new Failure(e);
  }
}
```

Failure

final class Failure implements Try

```
private Throwable exception;
Failure(Throwable exception) { this.exception = exception; }

@Override public Try map(Function f) { return this; }
@Override public Try flatMap(Function f) { return this; }

@Override public void forEach(Consumer callback) { return; }
@Override public void andThen(Consumer callback, Consumer errorHandling) {
  orElse(errorHandling);
}
@Override public Try ifSuccess(Consumer callback) { return this; }
@Override public void orElse(Consumer errorHandling) {
  errorHandling.accept(exception);
}

@Override public Try filter(Predicate f) { return this; }
@Override public Optional toOption() { return Optional.empty(); }
```

Success

final class Success⟨R⟩ implements Try⟨R⟩

```
private R result;
Success(R result) { this.result = result; }
@Override public <T> Try<T> map(Function<R, T> f) {
  return Try.tryWith(() -> f.apply(result));
}
@Override public <T> Try<T> flatMap(Function<R, Try<T>> f) {
  Try<Try<T>> mapped = Try.tryWith(() -> f.apply(result));
  if (mapped.isSuccessful()) {
    return mapped.get();
  } else {
    return new Failure(mapped.exception());
  }
}
@Override public void forEach(Consumer<R> callback) {
  Try.tryWith(() -> { callback.accept(result); return null; });
}
@Override public Try<R> ifSuccess(Consumer<R> callback) {
  return Try.tryWith(() -> { callback.accept(result); return result; });
}
@Override public void andThen(Consumer<R> callback,
    Consumer<Throwable> errorHandling) {
  ifSuccess(callback);
}
```

Try Application

```
Try<Integer> readAge = tryWith(Integer.parseInt(age));
Try<Date> readDate = tryWith(dateFormat.parse(created));

BiFunction<Integer,Date,PatientList> search = (a,c) -> {
  a = a & 0x7F;
  c = c.later(today) ? today : c;
  return patientService.search(a,c);
}

Try<PatientList> result = readAge.flatMap(a ->
  readDate.map(c -> search.apply(a, c)));

result.forEach(patientList -> ... );

result.ifSuccess(patientList -> ... ).orElse(exception -> ... );

result.andThen(
  patientList -> ... ,
  exception -> ...
);
```

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:

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

TryBuilder

final class TryBuilderN$\langle R1, R2, ...RN \rangle$

```
private Block<R1> b1;
private Block<R2> b2;
...
private Block<RN> bn;

static TryBuilderN forN(Block<R1> b1, Block<R2> b2, ... Block<RN> bn) {
  this.b1 = b1;
  this.b2 = b2;
  ...
  this.bn = bn;
}

public <T> Try<T> yield(MultiFunction<R1, R2, ...RN, T> f) {
  return
    Try.tryWith(b1).flatMap(v1 ->
      Try.tryWith(b2).flatMap(v2 ->
        ...
          Try.tryWith(bn).map(vn ->
            f.apply(v1, v2, ...vn))));
}
```

TryBuilder Application

```
for2(() -> Integer.parseInt(age),
     () -> dateFormat.parse(created))

  .yield((a,c) -> {
      a = a & 0x7F;
      c = c.later(today) ? today : c;
      return patientService.search(a,c);})

  .andThen(
      patientList -> ... ,
      exception -> ...
  );
```

Next Step

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