Functional thinking: Functional error handling with Either and Option

Type-safe functional exceptions

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

12 June 2012

Java™ developers are accustomed to handling errors by throwing and catching exceptions, which doesn't match the functional paradigm. This *Functional thinking* installment investigates ways to indicate Java errors functionally while still preserving type safety, shows how to wrap checked exceptions with functional returns, and introduces a handy abstraction named `Either`.

View more content in this series

About this series

This series aims to reorient your perspective toward a functional mindset, helping you look at common problems in new ways and find ways to improve your day-to-day coding. It explores functional programming concepts, frameworks that allow functional programming within the Java language, functional programming languages that run on the JVM, and some future-leaning directions of language design. The series is geared toward developers who know Java and how its abstractions work but have little or no experience using a functional language.

When you investigate deep subjects such as functional programming, fascinating offshoots occasionally emerge. In the last installment, I continued my miniseries on rethinking traditional Gang of Four design patterns in a functional way. I'll resume that topic in the next installment with a discussion on Scala-style pattern matching, but first I need to establish some background through a concept called `Either`. One of `Either`'s uses is a functional style of error handling, which I cover in this installment. After you understand the magic that `Either` can do with errors, I'll turn to pattern matching and trees in the next installment.

In Java, errors are traditionally handled by exceptions and the language's support for creating and propagating them. But what if structured exception handling didn't exist? Many functional languages don't support the exception paradigm, so they must find alternate ways of expressing error conditions. In this article, I show type-safe, error-handling mechanisms in Java that bypass

the normal exception-propagation mechanism (with assistance in some examples from the Functional Java framework).

Functional error handling

If you want to handle errors in Java without using exceptions, the fundamental obstacle is the language limitation of a single return value from methods. But methods *can*, of course, return a single `Object` (or subclass) reference, which can hold multiple values. So I could enable multiple return values by using a `Map`. Consider the `divide()` method in Listing 1:

Listing 1. Using `Map` to handle multiple returns

```
public static Map<String, Object> divide(int x, int y) {
    Map<String, Object> result = new HashMap<String, Object>();
    if (y == 0)
        result.put("exception", new Exception("div by zero"));
    else
        result.put("answer", (double) x / y);
    return result;
}
```

In Listing 1, I create a `Map` with `String` as the key and `Object` as the value. In the `divide()` method, I put either `exception` to indicate failure or `answer` to indicate success. Both modes are tested in Listing 2:

Listing 2. Testing success and failure with `Maps`

```
@Test
public void maps_success() {
    Map<String, Object> result = RomanNumeralParser.divide(4, 2);
    assertEquals(2.0, (Double) result.get("answer"), 0.1);
}

@Test
public void maps_failure() {
    Map<String, Object> result = RomanNumeralParser.divide(4, 0);
    assertEquals("div by zero", ((Exception) result.get("exception")).getMessage());
}
```

In Listing 2, the `maps_success` test verifies that the correct entry exists in the returned `Map`. The `maps_failure` test checks for the exception case.

This approach has some obvious problems. First, whatever ends up in the `Map` isn't type-safe, which disables the compiler's ability to catch certain errors. Enumerations for the keys would improve this situation slightly, but not much. Second, the method invoker doesn't know if the method call was a success, placing the burden on the caller to check the dictionary of possible results. Third, nothing prevents both keys from having values, which makes the result ambiguous.

What I need is a mechanism that allows me to return two (or more) values in a type-safe way.

The `Either` class

The need to return two distinct values occurs frequently in functional languages, and a common data structure used to model this behavior is the `Either` class. Using generics in Java, I can create a simple `Either` class, as shown in Listing 3:

Listing 3. Returning two (type-safe) values via the `Either` class

```
public class Either<A,B> {
    private A left = null;
    private B right = null;

    private Either(A a,B b) {
        left = a;
        right = b;
    }

    public static <A,B> Either<A,B> left(A a) {
        return new Either<A,B>(a,null);
    }

    public A left() {
        return left;
    }

    public boolean isLeft() {
        return left != null;
    }

    public boolean isRight() {
        return right != null;
    }

    public B right() {
        return right;
    }

    public static <A,B> Either<A,B> right(B b) {
        return new Either<A,B>(null,b);
    }

   public void fold(F<A> leftOption, F<B> rightOption) {
        if(right == null)
            leftOption.f(left);
        else
            rightOption.f(right);
    }
}
```

In Listing 3, `Either` is designed to hold either a `left` or `right` value (but never both). This data structure is referred to as a *disjoint union*. Some C-based languages contain the `union` data type, which can hold one instance of several different types. A disjoint union has slots for two types but holds an instance for only one of them. The `Either` class has a `private` constructor, making construction the responsibility of either of the two static methods `left(A a)` or `right(B b)`. The remaining methods in the class are helpers that retrieve and investigate the class members.

Armed with `Either`, I can write code that returns *either* an exception or a legitimate result (but never both) while retaining type safety. The common functional convention is that the *left* of an `Either` class contains an exception (if any), and the *right* contains the result.

Parsing Roman numerals

I have a class named `RomanNumeral` (whose implementation I'll leave to the reader's imagination) and a class named `RomanNumeralParser` that calls the `RomanNumeral` class. The `parseNumber()` method and illustrative tests appear in Listing 4:

Listing 4. Parsing Roman numerals

```
public static Either<Exception, Integer> parseNumber(String s) {
    if (! s.matches("[IVXLXCDM]+"))
        return Either.left(new Exception("Invalid Roman numeral"));
    else
        return Either.right(new RomanNumeral(s).toInt());
}

@Test
public void parsing_success() {
    Either<Exception, Integer> result = RomanNumeralParser.parseNumber("XLII");
    assertEquals(Integer.valueOf(42), result.right());
}

@Test
public void parsing_failure() {
    Either<Exception, Integer> result = RomanNumeralParser.parseNumber("FOO");
    assertEquals(INVALID_ROMAN_NUMERAL, result.left().getMessage());
}
```

In Listing 4, the `parseNumber()` method performs an astoundingly naive validation (for the purpose of showing an error), placing the error condition in the `Either`'s *left* or the result in its *right*. Both cases are shown in the unit tests.

This is a big improvement over passing `Map`s around. I retain type safety (note that I can make the exception as specific as I like); the errors are obvious in the method declaration via the generics; and my results come back with one extra level of indirection, unpacking the result (whether exception or answer) from `Either`. And the extra level of indirection enables *laziness*.

Lazy parsing and Functional Java

The `Either` class appears in many functional algorithms and is so common in the functional world that the Functional Java framework (see Resources) contains an implementation of `Either` that would work in the Listing 3 and Listing 4 examples. But it was made to work with other Functional Java constructs. Accordingly, I can use a combination of `Either` and Functional Java's `P1` class to create *lazy* error evaluation. A lazy expression is one that executes on demand (see Resources).

In Functional Java, a `P1` class is a simple wrapper around a single method named `_1()` that takes no parameters. (Other variants — `P2`, `P3`, and so on — hold multiple methods.) A `P1` is used in Functional Java to pass a code block around without executing it, enabling you to execute the code in a context of your choosing.

In Java, exceptions are instantiated as soon as you `throw` an exception. By returning a lazily evaluated method, I can defer the creation of the exception until a later time. Consider the example and associated tests in Listing 5:

Listing 5. Using Functional Java to create a lazy parser

```
public static P1<Either<Exception, Integer>> parseNumberLazy(final String s) {
    if (! s.matches("[IVXLXCDM]+"))
        return new P1<Either<Exception, Integer>>() {
            public Either<Exception, Integer> _1() {
                return Either.left(new Exception("Invalid Roman numeral"));
            }
        };
```

```
    else
        return new P1<Either<Exception, Integer>>() {
            public Either<Exception, Integer> _1() {
                return Either.right(new RomanNumeral(s).toInt());
            }
        };
}

@Test
public void parse_lazy() {
    P1<Either<Exception, Integer>> result = FjRomanNumeralParser.parseNumberLazy("XLII");
    assertEquals((long) 42, (long) result._1().right().value());
}

@Test
public void parse_lazy_exception() {
    P1<Either<Exception, Integer>> result = FjRomanNumeralParser.parseNumberLazy("FOO");
    assertTrue(result._1().isLeft());
    assertEquals(INVALID_ROMAN_NUMERAL, result._1().left().value().getMessage());
}
```

The code in Listing 5 is similar to Listing 4, with an extra `P1` wrapper. In the `parse_lazy` test, I must unpack the result via the call to `_1()` on the result, which returns `Either`'s *right*, from which I can retrieve the value. In the `parse_lazy_exception` test, I can check for the presence of a *left*, and I can unpack the exception to discern its message.

The exception (along with its expensive-to-generate stack trace) isn't created until you unpack `Either`'s *left* with the `_1()` call. Thus, the exception is lazy, letting you defer execution of the exception's constructor.

Providing defaults

Laziness isn't the only benefit of using `Either` for error handling. Another is that you can provide default values. Consider the code in Listing 6:

Listing 6. Providing reasonable default return values

```
public static Either<Exception, Integer> parseNumberDefaults(final String s) {
    if (! s.matches("[IVXLXCDM]+"))
        return Either.left(new Exception("Invalid Roman numeral"));
    else {
        int number = new RomanNumeral(s).toInt();
        return Either.right(new RomanNumeral(number >= MAX ? MAX : number).toInt());
    }
}

@Test
public void parse_defaults_normal() {
    Either<Exception, Integer> result = FjRomanNumeralParser.parseNumberDefaults("XLII");
    assertEquals((long) 42, (long) result.right().value());
}

@Test
public void parse_defaults_triggered() {
    Either<Exception, Integer> result = FjRomanNumeralParser.parseNumberDefaults("MM");
    assertEquals((long) 1000, (long) result.right().value());
}
```

In Listing 6, let's assume that I never allow any Roman numerals greater than `MAX`, and any attempt to set one greater will default to `MAX`. The `parseNumberDefaults()` method ensures that the default is placed in `Either`'s *right*.

Wrapping exceptions

I can also use `Either` to wrap exceptions, converting structured exception handling to functional, as shown in Listing 7:

Listing 7. Catching other people's exceptions

```
public static Either<Exception, Integer> divide(int x, int y) {
    try {
        return Either.right(x / y);
    } catch (Exception e) {
        return Either.left(e);
    }
}

@Test
public void catching_other_people_exceptions() {
    Either<Exception, Integer> result = FjRomanNumeralParser.divide(4, 2);
    assertEquals((long) 2, (long) result.right().value());
    Either<Exception, Integer> failure = FjRomanNumeralParser.divide(4, 0);
    assertEquals("/ by zero", failure.left().value().getMessage());
}
```

In Listing 7, I attempt division, which potentially raises an `ArithmeticException`. If an exception occurs, I wrap it in `Either`'s *left*; otherwise I return the result in *right*. Using `Either` enables you to convert traditional exceptions (including checked ones) into a more functional style.

Of course, you can also lazily wrap exceptions thrown from called methods, as shown in Listing 8:

Listing 8. Lazily catching exceptions

```
public static P1<Either<Exception, Integer>> divideLazily(final int x, final int y) {
    return new P1<Either<Exception, Integer>>() {
        public Either<Exception, Integer> _1() {
            try {
                return Either.right(x / y);
            } catch (Exception e) {
                return Either.left(e);
            }
        }
    };
}

@Test
public void lazily_catching_other_people_exceptions() {
    P1<Either<Exception, Integer>> result = FjRomanNumeralParser.divideLazily(4, 2);
    assertEquals((long) 2, (long) result._1().right().value());
    P1<Either<Exception, Integer>> failure = FjRomanNumeralParser.divideLazily(4, 0);
    assertEquals("/ by zero", failure._1().left().value().getMessage());
}
```

Nesting exceptions

One of the nice features of Java exceptions is the ability to declare several different potential exception types as part of a method signature. `Either` can do that as well, albeit with increasingly

convoluted syntax. For example, what if I need a method on `RomanNumeralParser` that allows me to divide two Roman numerals, but I need to return two different possible exception conditions — either a parsing error or a division error? Using standard Java generics, I can nest exceptions, as illustrated in Listing 9:

Listing 9. Nested exceptions

```
public static Either<NumberFormatException, Either<ArithmeticException, Double>>
        divideRoman(final String x, final String y) {
    Either<Exception, Integer> possibleX = parseNumber(x);
    Either<Exception, Integer> possibleY = parseNumber(y);
    if (possibleX.isLeft() || possibleY.isLeft())
        return Either.left(new NumberFormatException("invalid parameter"));
    int intY = possibleY.right().value().intValue();
    Either<ArithmeticException, Double> errorForY =
            Either.left(new ArithmeticException("div by 1"));
    if (intY == 1)
        return Either.right((fj.data.Either<ArithmeticException, Double>) errorForY);
    int intX = possibleX.right().value().intValue();
    Either<ArithmeticException, Double> result =
            Either.right(new Double((double) intX) / intY);
    return Either.right(result);
}

@Test
public void test_divide_romans_success() {
    fj.data.Either<NumberFormatException, Either<ArithmeticException, Double>> result =
        FjRomanNumeralParser.divideRoman("IV", "II");
    assertEquals(2.0,result.right().value().right().value().doubleValue(), 0.1);
}

@Test

public void test_divide_romans_number_format_error() {
    Either<NumberFormatException, Either<ArithmeticException, Double>> result =
        FjRomanNumeralParser.divideRoman("IVooo", "II");
    assertEquals("invalid parameter", result.left().value().getMessage());
}

@Test
public void test_divide_romans_arthmetic_exception() {
    Either<NumberFormatException, Either<ArithmeticException, Double>> result =
        FjRomanNumeralParser.divideRoman("IV", "I");
    assertEquals("div by 1", result.right().value().left().value().getMessage());
}
```

In Listing 9, the `divideRoman()` method first unpacks the `Either` returned from the original `parseNumber()` method from Listing 4. If an exception has occurred in either of the two number conversions, I return an `Either` *left* with the exception. Next, I must unpack the actual integer values, then execute my other validation criteria. Roman numerals don't have the concept of zero, so I've made a rule that disallows division by 1: If the denominator is 1, I package my exception and place it in the *right*'s *left*.

In other words, I have three slots, delineated by types: `NumberFormatException`, `ArithmeticException`, and `Double`. The first `Either`'s *left* holds the potential `NumberFormatException`, and its *right* holds another `Either`. The second `Either`'s *left* contains a potential `ArithmeticException`, and its *right* contains the payload, the result. Thus, to get the actual answer, I must traverse `result.right().value().right().value().doubleValue()`!

Obviously, the practicality of this approach breaks down quickly, but it does provide a type-safe way to nest exceptions as part of the class signature.

The `Option` class

`Either` is a handy concept that I'll use to build tree-shaped data structures in the next installment. A similar class in Scala called `Option`, replicated in Functional Java, provides a simpler exception case: either *none*, indicating no legitimate value, or *some*, indicating successful return. `Option` is demonstrated in Listing 10:

Listing 10. Using `Option`

```
public static Option<Double> divide(double x, double y) {
    if (y == 0)
        return Option.none();
    return Option.some(x / y);
}

@Test
public void option_test_success() {
    Option result = FjRomanNumeralParser.divide(4.0, 2);
    assertEquals(2.0, (Double) result.some(), 0.1);
}

@Test
public void option_test_failure() {
    Option result = FjRomanNumeralParser.divide(4.0, 0);
    assertEquals(Option.none(), result);

}
```

As illustrated in Listing 10, an `Option` contains either `none()` or `some()`, similar to *left* and *right* in `Either` but specific to methods that may not have a legitimate return value.

Both `Either` and `Option` in Functional Java are *monads* — special data structures that represent *computation* and are used heavily in functional languages. In the next installment, I'll explore the monadic concepts relating to `Either` and show how it enables Scala-style pattern matching in isolated cases.

Conclusion

When you learn a new paradigm, you need to reconsider all the familiar ways of solving problems. Functional programming uses different idioms to report error conditions, most of which can be replicated in Java, with some admittedly convoluted syntax.

In the next installment, I will show how the lowly `Either` can be used to build trees.

Resources

Learn

- *The Productive Programmer* (Neal Ford, O'Reilly Media, 2008): Neal Ford's book discusses tools and practices that help you improve your coding efficiency.
- Functional Java: Functional Java is a framework that adds many functional language constructs to Java.
- Lazy evaluation: Read more about this strategy for evaluating expressions.
- Monads: Monads, a legendarily difficult subject in functional languages, will be covered in a future installment of this series.
- Scala: Scala is a modern, functional language on the JVM.
- Browse the technology bookstore for books on these and other technical topics.
- The blog and particularly this series on Throwing Away Throws provided inspiration and source material for this installment.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Download IBM product evaluation versions or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Neal Ford

Neal Ford is a software architect and Meme Wrangler at **Thought**Works, a global IT consultancy. He also designs and develops applications, instructional materials, magazine articles, courseware, and video/DVD presentations, and he is the author or editor of books spanning a variety of technologies, including the most recent *The Productive Programmer*. He focuses on designing and building large-scale enterprise applications. He is also an internationally acclaimed speaker at developer conferences worldwide. Check out his Web site.