programming has become an important tool for the challenges of our
you, a Java developer, can use it to your advantage.

The recent interest in functional programming started as a response
pervasiveness of *concurrency* as a way of scaling horizontally, throu
Multithreaded programming (see, e.g., [Goetz2006]) is difficult to do
developers are good at it. As we'll see, functional programming offers b
for writing robust, concurrent software.

An example of the greater need for horizontal scalability is the growth
sets requiring management and analysis, the so-called *big data* trend.
sets that are too large for traditional database management systems. Th

Because this is a short introduction and because it is difficult to repres
tional concepts in Java, there will be several topics that I won't discu
although I have added glossary entries, for completeness. These topics
ing, *partial application*, and *comprehensions*. I'll briefly discuss severa
such as *combinators*, *laziness*, and *monads*, to give you a taste of the
However, fully understanding these topics isn't necessary when you'r
tional programming.

A few years ago, when many developers started talking about functiona
(FP) as the best way to approach *concurrency*, I decided it was time to
judge for myself. I expected to learn some new ideas, but I assumed I
object-oriented programming (OOP) as my primary approach to sof
ment. I was wrong.

As I learned about functional programming, I found good ideas for imp
currency, as I expected, but I also found that it brought new *clarity*
about the design of types* and functions. It also allowed me to write mo
Functional programming made me rethink where module boundaries
how to make those modules better for reuse. I found that the functiona
community was building innovative and more powerful type systems th
correctness. I also concluded that functional programming is a better
the unique challenges of our time, like working with massive data sets
agile as requirements change ever more rapidly and schedules grow eve

Instead of remaining an OOP developer who tosses in some FP for sea
write functional programs that use objects judiciously. You could say th
for the concurrency, but I stayed for the "paradigm shift."

The funny thing is, we've been here before. A very similar phenomen

then put the *same object model* in the code! Even implementation deta
forms of input and output, seemed ideal for object modeling.

But let's be clear, both FP and OOP are tools, not panaceas. Each has a
disadvantages. It's easy to stick with the tried and true, even when th
better way available. Even so, it's hard to believe that objects, which h

well in the past, could be any less valuable today, isn't it? For me, my g
in functional programming isn't a repudiation of objects, which have p
Rather, it's a recognition that the drawbacks of objects are harder to ign
with the programming challenges of today. The times are different than t
objects were ascendant several decades ago.

Here, in brief, is why I became a functional programmer and why I beli

learn about it, too. For me, functional programming offers the best ap
the following challenges, which I face every day.

I Have to Be Good at Writing Concurrent Programs

It used to be that a few of the "smart guys" on the team wrote most of
code, using multithreaded concurrency, which requires carefully synch

Today, even your phone has several CPU cores (or your next one will)
to write robust concurrent software is no longer optional. Fortunately,
gramming gives you the right principles to think about concurrency and
several higher-level concurrency abstractions that make the job far easi

> Multithreaded programming, requiring synchronized access to sh
> *mutable* state, is the *assembly language of concurrency*.

Most Programs Are Just Data Management Probler

I work a lot with *big data* these days, mostly using the Apache Hadoo
tools, built around *MapReduce* [Hadoop]. When you are ingesting *te*

I've come to believe that faithfully representing the domain object model
be questioned. *Object-relational mapping* (ORM) and similar forms of
ware add overhead for transforming relational data into objects, movin
around the application, then ultimately transforming them back to rel
updates. Of course, all this extra code has to be tested and maintained

I know this practice arose in part because we love objects and we often
data, or maybe we just hate working with relational databases. (I speak
experience.) However, relational data, such as the result sets for querie
collections that can be manipulated in a functional way. Would it be
directly with that data?

I'll show you how working directly with more fundamental collection

mizes the overhead of working with object models, while still avoiding c
promoting reuse.

Functional Programming Is More Modular

Years ago, I had a large client that struggled to get work done with the
base. Their competition was running circles around them. One day I

up protocol layers that make possible all the wonderful things that we
with computers.

There are no similar standards for object-based components. Various
CORBA and COM had modest success, but ultimately failed for the sam
that objects are at the wrong level of abstraction. Concepts like "custo
new, yet we can't find a way to stop inventing a new representation fo

However, if we notice that an object is fundamentally just an aggregatio
we can see a way to define better standardized abstractions at lower leve
analogous to digit circuits. These standards are the fundamental colle
map, and *set*, along with "primitive" types like numbers and few well-c
concepts (e.g., Money in a financial application).

A further aid to modularity is the nature of *functions* in functional progra
avoid side effects, making them free of dependencies on other objects
easier to reuse in many contexts.

The net result is that a functional program defines abstractions where
useful, easier to reuse, compose, and also test.

> Any arbitrarily complex object can be decomposed into "atomic" v
> (like primitives) and collections containing those values and
> collections.

I Have to Work Faster and Faster

Development cycles are going asymptotically to zero length. That sour
cially if you started professional programming when I did, when projects
months, even years. However, today there are plenty of Internet sites th
code *several times a day* and all of us are feeling the pressure to get w
quickly, without sacrificing quality, of course.

When schedules were longer, it made more sense to model your domai
to *implement* that domain in code. If you made a mistake, it would t
correct with a new release. Today, for most projects, understanding th
cisely is less important than delivering some value quickly. Our under
domain will change rapidly anyway, as we and our customers discove
with each deployment. If we misunderstand some aspect of the dom
those mistakes quickly when we do frequent deployments.

If careful modeling seems less important, faithfully *implementing* the c even more suspect today than before. While Agile Software Developm improved our quality and our ability to respond to change, we need to r keep our code "minimally sufficient" for the requirements today, yet flex Functional programming helps us do just that.

Functional Programming Is a Return to Simplicity

against accidental complexity, the kind we add ourselves by our i choices, as opposed to the inherent complexity of the problem domain. ple, much of the object-oriented middleware in our applications today and wasteful, in my opinion.

I know that some of these claims are provocative. I'm not trying to c abandon objects altogether or to become an FP zealot. I'm trying to gi toolbox and a broadened perspective, so you can make more informed

Functional programming, in its "purest" sense, is rooted in how funct
and values actually work in mathematics, which is different from how
work in most programming languages.

Functional programming got its start before digital computers even ex
the theoretical underpinnings of computation were developed in the 1
maticians like Alonzo Church and Haskell Curry.

In the 1930s, Alonzo Church developed the Lambda Calculus, which
for defining and invoking functions (called *applying* them). Today, the
havior of most programming languages reflect this model.

Haskell Curry (for whom the Haskell language is named) helped develo
Logic, which provides an alternative theoretical basis for computation
Logic examines how *combinators*, which are essentially functions, comb
a computation. One practical application of combinators is to use th
blocks for constructing parsers. They are also useful for representing
planned computation, which can be analyzed for possible bugs and op
portunities.

More recently, Category Theory has been a fruitful source of ideas for
gramming, such as ways to structure computations so that side effects
and output), which change the state of the "world," are cleanly separa
with no side effects.

A lot of the literature on functional programming reflects its mathematic
can be overwhelming if you don't have a strong math background. In c
oriented programming seems more intuitive and approachable. Fortu

The first language to incorporate functional programming ideas was Li
developed in the late 1950s and is the second-oldest high-level program
after Fortran. The ML family of programming languages started in the 1
Caml, OCaml (a hybrid object-functional language), and Microsoft's F
best known functional language that comes closest to functional "pur
which was started in the early 1990s. Other recent functional languages
and Scala, both of which run on the JVM but are being ported to the .NE
Today, many other languages are incorporating ideas from functional

The Basic Principles of Functional Programming

languages considered *functional* languages? Functional languages sha
principles.

Avoiding Mutable State

The first principle is the use of *immutable* values. You might recall the *gorean equation* from school, which relates the lengths of the sides of a

$x^2 + y^2 = z^2$

If I give you *values* for the *variables* `x` and `y`, say `x=3` and `y=4`, you can co for `z` (`5` in this case). The key idea here is that values are *never* modifi crazy to say `3++`, but you could start over by assigning *new* values to the

Most programming languages don't make a clear distinction between a contents of memory) and a variable that refers to it. In Java, we'll use fi *variable* reassignment, so we get objects that are immutable *values*.

Why should we avoid mutating values? First, allowing mutable values multithreaded programming so difficult. If multiple threads can mo shared value, you have to synchronize access to that value. This is qu error-prone programming that even the experts find challenging [Goe make a value immutable, the synchronization problem disappears. Con is harmless, so multithreaded programming becomes far easier.

A second benefit of immutable values relates to program correctness in is harder to understand and exhaustively test code with mutable values mutations aren't localized to one place. Some of the most difficult bugs systems occur when state is modified non-locally, by client code that where in the program.

Consider the following example, where a mutable List is used to hol
orders:

```
public class Customer {
  // No setter method
  private final List<Order> orders;
  public List<Order> getOrders() { return orders; }
  public Customer(...) {...}
}
```

It's reasonable that clients of Customer will want to view the list of Or
nately, by exposing the list through the getter method, getOrders, we
over them! A client could modify the list without our knowledge. We c
setter for orders and it is declared final, but these protections only pr
a new list to orders. The list itself can still be modified.

We could work around this problem by having getOrders return a co
by adding special accessor methods to Customer that provide contr
orders. However, copying the list is expensive, especially for large lists.

What happens when the list of orders is supposed to change, but it has
Should we relent and make it mutable to avoid the overhead of mak
Fortunately, we have an efficient way to copy large data structures; we'll
that aren't changing! When we add a new order to our list of orders, w
rest of the list. We'll explore how in Chapter 3.

Some mutability is unavoidable. All programs have to do IO. Otherwise
nothing but heat up the CPU, as a joke goes. However, functional pr
courages us to think strategically about when and where mutability is n
encapsulate mutations in well-defined areas and keep the rest of the co
tation, we improve the robustness and modularity of our code.

We still need to handle mutations in a thread-safe way. Software Transac
and the Actor Model give us this safety. We'll explore both in Chapter

> Make your objects immutable. Declare fields `final`. Only provide g
> for fields and then only when necessary. Be careful that mutable
> objects can still be modified. Use mutable collections carefully
> "Minimize Mutability" in [Bloch2008] for more tips.

Functions as First-Class Values

In Java, we are accustomed to passing objects and primitive values t
turning them from methods, and assigning them to variables. This mea
and primitives are *first-class values* in Java. Note that classes themsel
class values, although the reflection API offers information about class

Functions are not first-class values in Java. Let's clarify the differe
method and a *function*.

> A *method* is a block of code attached to a particular class. It can o
> called in the context of the class, if it's defined to be `static`, or
> context of an instance of the class. A *function* is more general. It
> attached to any particular class or object. Therefore, all instance
> *ods* are *functions* where one of the arguments is the object.

Java only has methods and methods aren't *first-class* in Java. You can't
as an argument to another method, return a method from a method, or a
as a value to a variable.

However, most *anonymous inner classes* are effectively function "wrappe
methods take an instance of an interface that declares one method. He

to the left of the "arrow" (->) and the body of the function is to the righ
Notice how much boilerplate code this syntax removes!

> The term *lambda* is another term for *anonymous function*. It comes
> the use of the Greek lambda symbol λ to represent functions in *la*
> *calculus*.

For completeness, here is another example function type, one that takes
of types A1 and A2, respectively, and returns a non-void value of type R

Closures

A *closure* is formed when the body of a function refers to one or more *f*
variables that aren't passed in as arguments or defined locally, but are d
enclosing scope where the function is defined. The runtime has to "clos
variables so they are available when the function is actually executed,
happen long after the original variables have gone out of scope! Java has lin
for closures in inner classes; they can only refer to final variables in the enc

Higher-Order Functions

There is a special term for functions that take other functions as argu
them as results: *higher-order functions*. Java methods are limited to prin
jects as arguments and return values, but we can mimic this feature wit
interfaces.

Higher-order functions are a powerful tool for building abstractions a
behavior. In Chapter 3, we'll show how higher-order functions allow
customization of standard library types, like `Lists` and `Maps`, and also
bility. In fact, the *combinators* we mentioned at the beginning of this cha
order functions.

Side-Effect-Free Functions

Being able to replace a function call for a particular set of parameters w
returns is called *referential transparency*. It has a fundamental implicatio
with no side effects; the function and the corresponding return values
onymous, as far as the computation is concerned. You can represent the
any such function with a value. Conversely, you can represent any value
call!

Side-effect-free functions make excellent building blocks for reuse, si
depend on the context in which they run. Compared to functions with s
are also easier to design, comprehend, optimize, and test. Hence, they a
have bugs.

which encapsulates the required loop counting. We'll see other iterati
in the next chapter, when we discuss operations on functional collectio

The classic functional alternative to an iterative loop is to use *recursi*
pass through the function operates on the next item in the collection unt
point is reached. Recursion is also a natural fit for certain algorithms, su
a tree where each branch is itself a tree.

Consider the following example, where a unit test defines a simple tr
value at each node, and left and right subtrees. The `Tree` type defi

Lazy vs. Eager Evaluation

Mathematics defines some *infinite* sets, such as the *natural numbers* (a
gers). They are represented symbolically. Any particular finite subset o
uated only on demand. We call this *lazy evaluation*. *Eager evaluation*
to represent all of the infinite values, which is clearly impossible.

Some languages are lazy by default, while others provide lazy data stru

```
  public static List<Integer> take(int count) {
    return doTake(emptyList(), count);
  }

  private static List<Integer> doTake(List<Integer> accumulator, int c
    if (count == ZERO)
      return accumulator;
    else
      return doTake(list(next(count - 1), accumulator), count - 1);
  }
}
```

We start with a definition of zero, then use next to compute each natur

its predecessor. The take(n) method is a pragmatic tool for extracting a the integers. It returns a List of the integers from 1 to n. (The List type discussed in Chapter 3. It isn't java.util.List.) Note that the helper m *tail-call recursive*.

We have replaced values, integers in this case, with functions that co demand, an example of the *referential transparency* we discussed earl sentation of infinite data structures wouldn't be possible without thi referential transparency and lazy evaluation require side-effect-free fur mutable values.

Finally, lazy evaluation is useful for deferring expensive operations u never executing them at all.

Declarative vs. Imperative Programming

Finally, functional programming is *declarative*, like mathematics, where relationships are defined. The runtime figures out how to compute fi definition of the factorial function provides an example:

```
factorial(n) = 1                    if n = 1
```

The definition relates the value of `factorial(n)` to `factorial(n-1)`, a r
tion. The special case of `factorial(1)` terminates the recursion.

Object-oriented programming is primarily *imperative*, where we *tell* the
specific steps to do.

To better understand the differences, consider this example, which pr
ative and an imperative implementation of the factorial function:

ments a calculation of factorials, but its structure is more declarative th
I formatted the method to look similar to the definition of factorial.

The `imperativeFactorial` method uses mutable values, the loop co
`result` that accumulates the calculated value. The method explicitly im
ticular algorithm. Unlike the declarative version, this method has lots of
steps, making it harder to understand and keep bug free.

Declarative programming is made easier by *lazy evaluation*, because la
runtime the opportunity to "understand" all the properties and relatio
mine the optimal way to compute values on demand. Like *lazy evaluat*
programming is largely incompatible with mutability and functions wi

Designing Types

Whether you prefer *static* or *dynamic* typing, functional programming h
lessons to teach us about good type design. First, all functional langua
the use of core container types, like *lists*, *maps*, *trees*, and *sets* for captu
forming data, which we'll explore in Chapter 3. Here, I want to dis
benefits of functional thinking about types, enforcing valid values fo
applying rigor to type design.

of Lisp (as its name suggests). Don't confuse the following classic defini
built-in `List` type.

As you read this code, keep a few things in mind. First, `List` is an Algeb
with structural similarities to `Option<T>`. In both cases, a common inter
protocol of the type, and there are two concrete subtypes, one that repr
and one that represents "non-empty."

Second, despite the similarities of structure, we'll introduce a few more i
idioms that get us closer to the requirements of a true algebraic dat

prevent this as they can only create lists terminated by `EMPTY`. Hence, th
good behavior.

> *Pure* functional programming uses recursion instead of loops, s
> loop counter would have to be mutable.

We used a few idioms to enforce the algebraic data type constraint tha
archy must be closed, with only two concrete types to represent all l

Maps

Let's talk briefly about maps, which associate keys with values, as in th
example:

```
Map<String,String> languageToType = new HashMap<String,String>();
languageToType.put("Java",     "Object Oriented");
```

```
languageToType.put("Ruby",     "Object Oriented");
languageToType.put("Clojure", "Functional");
languageToType.put("Scala" ,  "Hybrid Object-Functional");
```

Maps don't make good *algebraic data types*, because the value of defin
vs. a "non-empty" type (or similar decomposition) is less useful. In pa
the fact that the "obvious" implementation of `List` is strongly implied l
`tail` design.

There is no such obvious implementation of `Map`. In fact, we need flexib
different implementations for different performance goals. Instead, `Ma`

Combinator Functions: The Collection Power Tools

You already think of lists, maps, etc. as "collections," all with a set of com
Most collections support Java `Iterators`, too. In functional programm
three core operations that are the basis of almost all work you do with

Filter
: Create a new collection, keeping only the elements for which a filter
`true`. The size of the new collection will be less than or equal to
original collection.

Map
: Create a new collection where each element from the original coll
formed into a new value. Both the original collection and the new
have the same size. (Not to be confused with the `Map` data structur

Fold
: Starting with a "seed" value, traverse through the collection and us
to build up a new final value where each element from the original c
tributes" to the final value. An example is summing a list of intege

Many other common operations can be built on top of these three. To
the basis for implementing concise and *composable* behaviors. Let's se

```java
package datastructures2;
...
public class ListModule {
  public static interface List<T> {
    ...
    public       List<T>  filter    (Function1<T,Boolean> f);
    public <T2> List<T2> map       (Function1<T,T2> f);
    public <T2> T2       foldLeft  (T2 seed, Function2<T2,T,T2> f);
    public <T2> T2       foldRight (T2 seed, Function2<T,T2,T2> f);
    public       void     foreach   (Function1Void<T> f);
  }

  public static final class NonEmptyList<T> implements List<T> {
    ...
    public List<T> filter (Function1<T,Boolean> f) {
       if (f.apply(head())) {
        return list(head(), tail().filter(f));
      } else {
        return tail().filter(f);
      }
    }

    public <T2> List<T2> map (Function1<T,T2> f) {
      return list(f.apply(head()), tail().map(f));
    }

    public <T2> T2 foldLeft (T2 seed, Function2<T2,T,T2> f) {
      return tail().foldLeft(f.apply(seed, head()), f);
    }

    public <T2> T2 foldRight (T2 seed, Function2<T,T2,T2> f) {
      return f.apply(head(), tail().foldRight(seed, f));
    }

    public void foreach (Function1Void<T> f) {
      f.apply(head());
      tail().foreach(f);
```

There are five new methods declared in the `List` interface. We need t
`fold`, `foldLeft` and `foldRight`, for reasons we'll discuss in a moment. A
a `foreach` method for convenience.

Each implementation for the five new methods in `NonEmptyList` is recu
are no checks for the end of the recursion! The corresponding imp
`EMPTY` terminates the recursion. This means we have eliminated the need
tests, replacing them with object-oriented polymorphism!

Recall that the `filter` method will return a new `List`. It takes a `Functi`
`f` and applies `f` to each element. In `Empty`, `filter` just returns `EMPTY`. In `N`
the result of applying `f` to `head` (`f.apply(head())`) is `true`, then `filter` co
list with `head` and the result of calling `filter` on the tail. Otherwise, `fil`
the result of applying `filter` to the `tail`, thereby discarding `head`. So, `fil`
and it terminates when it is called on an empty list.

The `map` method is slightly simpler, since it never discards an eleme
recursion to traverse the list, applying `f` to each element and building up
the results. Note that `f` is now of type `Function1<T,T2>`, because the goa
original elements of type `T` to be transformed into instances of the new
time, `EMPTY`'s `map` method calls `emptyList`, because it must return an
`List<T2>`, instead of an object of the original type.

The `foldLeft` and the `foldRight` methods are the hardest to understan
actually the most important, as all other methods could be implement
We'll start with a general discussion of how these methods work, the
implementation details.

The reason there are two versions is because they traverse the collectio
function in different orders. In some cases, the ordering doesn't matter
results will be different. There are other important differences we'll see

In a nutshell, `foldLeft` groups the elements from left to right, while `fo`
them from right to left. It might help to start with an illustration of
methods work. Suppose I have a list of the integers 1 through 4. I wa
using fold. Consider the following example:

So, we can see that `foldRight` can be used with infinite data structures,
`n` elements will be evaluated.

However, `foldRight` has a drawback; it is not *tail recursive*. Why? No
an addition *after* the recursive call returns. The recursive call isn't the
the *tail* of the algorithm. The *tail-call optimization* can't be applied to f

However, `foldLeft` *is* tail recursive. Let's write the left-recursive vers
`next` example:

happens *before* the call, to construct the argument passed to the nex
`foldLeft`. Hence the recursion is a tail call, the last calculation done.

However, `foldLeft` can't be used for infinite data structures. There is
we can replace a call to `next` with the seed, as for `foldRight`. So, `foldL`
evaluate the expression, blowing up on an infinite data structure.

Now let's return to the implementations, starting with `foldLeft`. First,
is of type `Function2<T2,T,T2>`. The first `T2` type parameter represents t
that we are building up a new value that could be just about anythin
String, an Integer (for sums), etc. So, we have to pass a starter or "seed"
conventional name for this argument is `accumulator`, since it will con
mulation" of the work done up to a given point.

The second type parameter `T` for `f` is the type of the elements in the or
last type parameter `T2` is the final return type of the call to `foldLeft`. No

call to `tail().foldRight` *after* the latter has returned. As we discussed
why `foldRight` is not tail recursive.

> Consider these concise and precise definitions: `foldLeft` "is the f
> mental list iterator" and `foldRight` "is the fundamental list recu
> operator" [Shivers].

To end our discussion of `fold`, note that there is a similar operation calle
is like `fold`, but the initial value of the collection is used as the `seed`. Hen

---

general, because the type of the result doesn't have to be the same as
collection elements. Also, unlike `fold`, `reduce` will fail if used on an em
since there is no "first" value!

Finally, we have `foreach`, the simplest of all these methods. Technically,
be disallowed in "pure" functional programming, because it performs o

putations from simpler pieces. Combinators are arguably *the most reus
we have in programming*.

> The `filter`, `map`, and `fold` functions are *combinators*, composable l
> ing blocks that let us construct complex computations from si
> pieces. They are highly *reusable*. The combination of `map` and `reduc`
> the inspiration for the *MapReduce* approach to processing massiv
> sets [Hadoop].

Finally, recall that I implemented these functions using recursion, but
them avoids recursion, as in our `FunctionCombinatorTest` example. Th

a value. While this may be fine for small objects, it will be too expensive f
like long lists and large maps.

Fortunately, we can have both immutability and acceptable performa
allocate memory for what is actually changing and we share the unchan
the original object. This approach is called *structure sharing*. Tree data
vide the balance of performance and space efficiency required to do t
abstraction might still be a `List`, a `Map`, or another data structure. The t
for the internal storage. Note that the trees themselves and their nod
mutable. Otherwise, structure sharing will be dangerous, as mutatio
object will be seen by others that share the same substructure!

To simplify the discussion, let's use unbalanced binary trees. They p
O($\log_2(N)$) search times (unless the tree is *really* unbalanced). Real

Software Transactional Memory

Chances are you've worked on an application with a database backen
of most relational databases is support for *ACID transactions*, an acron
ity, *consistency*, *isolation*, and *durability*.* The goal of ACID transacti
logical inconsistencies in a given set of related records, for example wh
taneous updates leave the set of records in an inconsistent state, or up
that are based on stale data, which could effectively erase more recent

Software Transactional Memory (STM) brings transactions to locations
are referenced by variables [STM] (see also [PeytonJones2007]). STM
durability, because memory isn't durable (e.g., if the power is lost), bu
vide the ACI, *atomicity*, *consistency*, and *isolation* in ACID.

The model in STM is to separate references to values from the values t
saw this principle at work in Akka actors. In STM, a program has a refe
of interest. The STM framework provides a protocol for changing the
the reference "points."

However, values themselves are *not* changed. They remain immutable.
ences change to point to new values. We saw in "Persisten
tures" on page 36 how the appropriate choice of implementation can
cient way to make a new value from a large object without copying the
aren't changing. Rather, those parts are shared between the old and ne