clarity for *performance*. Although this is a common trade-off, it's one I'd r

In his "Simple Made Easy" keynote at the Strange Loop conference, R
creator of Clojure, reintroduced an arcane word, *complect*: to join by wea
together; to interweave. Imperative programming often forces you to

tasks so that you can fit them all within a single loop, for efficiency. F
gramming via higher-order functions such as `map()` and `filter()` allow
your level of abstraction, seeing problems with better clarity. I show ma
functional thinking as a powerful antidote to incidental complecting.

Aligning with Language Trends

Michael Feathers, author of *Working with Legacy Code*, captured a key
tween functional and object-oriented abstractions in 140 lowly charact

> OO makes code understandable by encapsulating moving parts. FP makes
> standable by minimizing moving parts.
>
> — Mich

Think about the things you know about object-oriented programmin

"moving parts." Rather than build mechanisms to *control* mutable state, r
languages try to *remove* mutable state, a "moving part." The theory fol
language exposes fewer potentially error-prone features, it is less likely
to make errors. I will show numerous examples throughout of functiona
eliminating variables, abstractions, and other moving parts.

In object-oriented imperative programming languages, the units of re
and the messages they communicate with, captured in a class diagram
work in that space, *Design Patterns: Elements of Reusable Object-Orient*
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides), inclu
class diagram with each pattern. In the OOP world, developers are encou
unique data structures, with specific operations attached in the form of r
tional programming languages don't try to achieve reuse in quite th
functional programming languages, the preference is for a few key data s
as `list`, `set`, and `map`) with highly optimized operations on those data
utilize this machinery, developers pass data structures plus higher-ord
plug into the machinery, customizing it for a particular use.

Consider a simplified portion of the code from Example 1-2:

```
regexToList(words, "\\b\\w+\\b").stream()
```

tactically sugared (`w → !NON_WORDS.contains(w)`)). The machinery a
criteria in an efficient way, returning the filtered list.

Encapsulation at the function level allows reuse at a more granular, fun
than building new class structures for every problem. Dozens of XML l
the Java world, each with its own internal data structures. One advantag
higher-level abstractions is already appearing in the Clojure space. Rece
vations in Clojure's libraries have managed to rewrite the map function

ically parallelizable, meaning that all map operations get a performanc
developer intervention.

Functional programmers prefer a few core data structures, building op
nery to understand them. Object-oriented programmers tend to create
tures and attendant operations constantly—building new classes and m
them is the predominant object oriented paradigm. Encapsulating all
within classes discourages reuse at the method level, preferring larger f
reuse. Functional programming constructs make it easier to reuse code a
level.

Consider the `indexOfAny()` method, from the popular Java framework
mons, which provides a slew of helpers for Java, in Example 1-3.

```
    Some(result.head)
}
```

In Example 1-5, I create an indexed version of the input string. Scala's takes a collection (the range of numbers up to the length of my input st bines it with the collection of String characters, creating a new collecti pairs from the original collection. For example, if my input string is za dInput contains Vector ((0,z), (1,a), (2,b), (3,y), (4,c), ((7,x)). The zip name comes from the result; it looks as if the two collec joined like the teeth of a zipper.

Once I have the indexed collection, I use Scala's for() comprehension the collection of search characters, then access each pair from the ind Scala allows shorthand access to collection elements, so I can compare th character to the second item for the collection (if (char == pair._2 acters match, I return the index portion of the pair (pair._1).

A common source of confusion in Java is the presence of null: is it a le value or does it represent the absence of a value? In many functional la included), that ambiguity is avoided by the Option class, which conta indicating no return, or Some, containing the returned values. For Ex problems asks for only the first match, so I return result.head, the firs results collection.

Learning a new programming language is easy: you merely learn the familiar concepts. For example, if you decide to learn JavaScript, your f a resource that explains how JavaScript's if statement works. Typically, d new languages by applying what they know about existing languages. new paradigm is difficult—you must learn to see different soluti problems.

Writing functional code doesn't require a shift to a functional program such as Scala or Clojure but rather a shift in the way you approach pro

A Common Example

Once garbage collection became mainstream, it simultaneously elimin
egories of hard-to-debug problems and allowed the runtime to manag
is complex and error-prone for developers. Functional programming
same thing for the algorithms you write, allowing you to work at a hig
straction while freeing the runtime to perform sophisticated optimizatio
receive the same benefits of lower complexity and higher performan
collection provides, but at a more intimate level, in the way you devise

Imperative Processing

Imperative programming describes a style of programming modeled a
commands (imperatives) that modify state. A traditional `for loop` is a
ample of the imperative style of programming: establish an initial stat
series of commands for each iteration of the loop.

To illustrate the difference between imperative and functional program
with a common problem and its imperative solution. Let's say that you

```
        for(int i = 0; i < listOfNames.size(); i++) {
            if (listOfNames.get(i).length() > 1) {
                result.append(capitalizeString(listOfNames.get(i))).ap
            }
        }
        return result.substring(0, result.length() - 1).toString();
    }
```

result, along with a trailing comma. The last name in the final string sh
the comma, so I strip it off the final return value.

Imperative programming encourages developers to perform operation
In this case, I do three things: *filter* the list to eliminate single character
list to capitalize each name, then *convert* the list into a single string. F
these three operations *Useful Things* to do to a list. In an imperative la
use the same low-level mechanism (iteration over the list) for all thre
cessing. Functional languages offer specific helpers for these operation

Functional Processing

Functional programming describes programs as expressions and tr
modeling mathematical formulas, and tries to avoid mutable state. F
gramming languages categorize problems differently than imperative
logical categories listed earlier (*filter*, *transform*, and *convert*) are repre
tions that implement the low-level transformation but rely on the de
tomize the low-level machinery with a higher-order function, supplie
parameters. Thus, I could conceptualize the problem as the pseudocode


```
    -> transform(x.capitalize)
    -> convert(x + "," + y)
```

Functional languages allow you to model this conceptual solution wi
about the details.

Consider the company process example from Example 2-1 implemented
in Example 2-3.

```
val employees = List("neal", "s", "stu", "j", "rich", "bob", "aiden",
          "liam", "mason", "noah", "lucas", "jacob", "jayden", "jack")

val result = employees
  .filter(_.length() > 1)
  .map(_.capitalize)
  .reduce(_ + "," + _)
```

The use of Scala in Example 2-3 reads much like the pseudocode in Ex
necessary implementation details. Given the list of names, I first filter
single characters. The output of that operation is then fed into the `map` f
executes the supplied code block on each element of the collection, retur
formed collection. Finally, the output collection from `map` flows to the
tion, which combines each element based on the rules supplied in the
this case, to combine the first two elements, I concatenate them with a
three of these small functions, I don't care what the parameters are n
allows me to skip the names and use an underscore instead. In the case
still pass two parameters, which is the expected signature even though
same generic indicator, the underscore.

I chose Scala as the first language to show this implementation because o
familiar syntax and the fact that Scala uses industry-consistent names for
In fact, Java 8 has these same features, and closely resembles the Scala ve

Example 2-4. Java 8 version of the Company Process

```
    public String cleanNames(List<String> names) {
        if (names == null) return "";
        return names
                .stream()
                .filter(name -> name.length() > 1)
                .map(name -> capitalize(name))
                .collect(Collectors.joining(","));
```

In Example 2-4, I use the collect() method rather than reduce() bec
efficient with the Java String class; collect() is a special case for redu
Otherwise, it reads remarkably similarly to the Scala code in Example 2

If I was concerned that some of the items in the list might be null, I
another criterium to the stream:

```
return names
        .stream()
        .filter(name -> name != null)
        .filter(name -> name.length() > 1)
        .map(name -> capitalize(name))
        .collect(Collectors.joining(","));
```

The Java runtime is intelligent enough to combine the null check and le
a single operation, allowing you to express the idea succinctly yet still h
code.

Groovy has these features but names them more consistently than scrip
such as Ruby. The Groovy version of the "company process" from Exam
in Example 2-5.

Example 2-5. Processing in Groovy

```
public static String cleanUpNames(listOfNames) {
  listOfNames
      .findAll { it.length() > 1 }
      .collect { it.capitalize() }
      .join ','
}
```

While Example 2-5 is structurally similar to the Scala code in Example 2
names and substitution identifier differ. Groovy's findAll on a collect

tions and stop going immediately for detailed implementations.

What are the benefits of thinking at a higher level of abstraction? Firs
you to categorize problems differently, seeing commonalities. Second, it
time to be more intelligent about optimizations. In some cases, reord
stream makes it more efficient (for example, processing fewer items) if it
the ultimate outcome. Third, it allows solutions that aren't possible whe
is elbow deep in the details of the engine. For example, consider the a
required to make the Java code in Example 2-1 run across multiple th
you control the low-level details of iteration, you must weave the thread
In the Scala version, I can make the code parallel by adding par to the st
in Example 2-8.

Example 2-8. Scala processing in parallel

```
val parallelResult = employees
  .par
  .filter(_.length() > 1)
  .map(_.capitalize)
  .reduce(_ + "," + _)
```

```
public String cleanNamesP(List<String> names) {
    if (names == null) return "";
    return names
            .parallelStream()
            .filter(n -> n.length() > 1)
            .map(e -> capitalize(e))
            .collect(Collectors.joining(","));
```

required to write the `sum()` method—in Java 8, it is one of the stream t
generates values.

In physics, energy is differentiated into *potential*, energy stored and re
kinetic, energy expenditure. For collections in languages such as Java b
all collections acted as kinetic energy: the collection resolved values imm
ing no intermediate state. Streams in functional languages work mor
energy, which is stored for later use. The stream holds the origin
Example 2-12, the origin comes from the `range()` method) and whatev
been attached to the stream, such as filtering operations. The stream
from potential to kinetic until the developer "asks" for values, using a t
eration such as `forEach()` or `sum()`. The stream is passable as a paramet
additional criteria added later to its potential until it becomes kinetic. Th
of *lazy evaluation*, which I cover in detail in Chapter 4.

This style of coding was possible, with difficulty, in previous versions of J
useful frameworks.

Functional Java Number Classifier

While all modern languages now include higher-order functions, man
stay on older versions of runtimes such as Java for many years for no
sons. *Functional Java* is an open source framework whose mission inc
many functional idioms to Java post version 1.5 with as little intrusiven
For example, because the Java 1.5-era JDK don't include higher-order fu
tional Java mimics their use via generics and anonymous inner classe
classifier takes on a different look when implemented using Functional J
shown in Example 2-13.

Example 2-13. Number classifier using the Functional Java framework

In Example 2-13, I used the `foldLeft()` method, which collapses ele
toward the first element. For addition, which is commutative, the di
matter. If, on the other hand, I need to use an operation in which orde
is also a `foldRight()` variant.

> Higher-order abstractions eliminate friction.

we were talking about the waning of Smalltalk versus the waxing of Ja
extensive work in both and says that he initially viewed the switch fro
Java as a syntactic inconvenience, but eventually as an impediment
thinking afforded in the previous world. Placing syntactic hurdles arou
abstractions adds needless friction to the thought process.

> Don't add needless friction.

Filter

A common operation on lists is filtering: creating a smaller list by filte
list based on some user-defined criteria. Filtering is illustrated in Figur

stitution keyword `it` as a placeholder, and the last line of the method
return value, which is the list of factors in this case.

> Use `filter` to produce a subset of a collection based on s
> filtering criteria.

Map

The *map* operation transforms a collection into a new collection by appl
to each of the elements, as illustrated in Figure 2-2.

`aliquot-sum` in each case and returns the appropriate keyword (an en
lineated with a leading colon).

> Use *map* to transform collections in situ.

Fold/Reduce

The third common function has the most variations in name, and mar
ences, among popular languages. `foldLeft` and `reduce` are specific vari
manipulation concept called *catamorphism*, which is a generalization o

The `reduce` and `fold` operations have overlapping functionality, with su
from language to language. Both use an *accumulator* to gather valu
function is generally used when you need to supply an initial value for th
whereas `fold` starts with nothing in the accumulator. The order of op
collection is specified by the specific method name (for example, `fol`
`Right`). Neither of these operations mutates the collection.

I show the `foldLeft()` function in Functional Java. In this case, a "fold

At first it's not obvious how the one-line body in Example 2-19 performs a
to calculate the `aliquotSum`. In this case, the *fold* operation refers to a
that combines each element of the list with the next one, accumulating
for the entire list. A fold left combines the list elements leftward, start
value and accumulating each element of the list in turn to yield a final re
illustrates a fold operation.

Because addition is commutative, it doesn't matter if you do a `foldL`
`Right()`. But some operations (including subtraction and division) ca
so the `foldRight()` method exists to handle those cases. In purely functi
left and right folds have implementation differences. For example, right fo
on infinite lists whereas left folds cannot.

Example 2-13 uses the Functional Java-supplied `add` enumeration; the
cludes the most common mathematical operations for you. But what

to produce a different-sized (usually smaller but not necessarily) value,
a collection or a single value.

Use `reduce` or `fold` for piecewise collection processing.

`collect()`, `map()`, and `inject()` appear. Once you become accustomed
tools in your toolbox, you'll find yourself turning to them again and ag

One of the challenges of learning a different paradigm such as functiona
is learning the new building blocks and "seeing" them peek out of probler
solution. In functional programming, you have far fewer abstractions,
generic (with specificity added via higher-order functions). Because
gramming relies heavily on passed parameters and composition, you h
to learn about the interactions among moving parts, making your job e

Synonym Suffering

This second version is less verbose because Scala allows substitution of p
underscores. Both versions yield the same result.

Many examples of filtering operations use numbers, but filter() app
lection. This example applies filter() to a list of words to determine
words:

```
val words = List("the", "quick", "brown", "fox", "jumped",
                 "over", "the", "lazy", "dog")
words filter (_.length == 3)
// List(the, fox, the, dog)
```

Another filtering variant in Scala is the partition() function, which r
nate version of a collection by splitting it into multiple parts; the origi
unchanged. The split is based on the higher-order function that you pa
the separation criteria. Here, the partition() function returns two lis
according to which list members are divisible by 3:

```
numbers partition (_ % 3 == 0)
// (List(3, 6, 9),List(1, 2, 4, 5, 7, 8, 10))
```

The filter() function returns a collection of matching elements, w
returns only the first match:

```
numbers find (_ % 3 == 0)
// Some(3)
```

However, the return value for find() isn't the matched value itself, but r
wrapped in an Option class. Option has two possible values: Some or N
some other functional languages, uses Option as a convention to avoid
in the absence of a value. The Some() instance wraps the actual return
3 in the case of numbers find (_ % 3 == 0). If I try to find someth
exist, the return is None:

I discuss Option and similar classes in depth in Chapter 5.

Scala also includes several functions that process a collection based on a
tion and either retain values or discard them. The takeWhile() funct
largest set of values from the front of the collection that satisfy the pred

```
List(1, 2, 3, -4, 5, 6, 7, 8, 9, 10) takeWhile (_ > 0)
// List(1, 2, 3)
```

The dropWhile() function skips the largest number of elements t
predicate:

```
words dropWhile (_ startsWith "t")
// List(quick, brown, fox, jumped, over, the, lazy, dog)
```

Map

The second major functional transformation that's common to all funct
is *map*. A map function accepts a higher-order function and a collectio
the passed function to each element and returns a collection. The retu
(unlike with filtering) is the same size as the original collection, but with

Scala

Scala's `map()` function accepts a code block and returns the transforme

```
List(1, 2, 3, 4, 5) map (_ + 1)
// List(2, 3, 4, 5, 6)
```

The `map()` function works on all applicable types, but it doesn't nece transformed version of each element of the collection. In this example, the sizes of all elements in a string:

```
words map (_.length)
// List(3, 5, 5, 3, 6, 4, 3, 4, 3)
```

Nested lists occur so often in functional programming languages that for denesting—typically called *flattening*—is common. Here is an examp a nested list:

```
List(List(1, 2, 3), List(4, 5, 6), List(7, 8, 9)) flatMap (_.toLis
```

differences.

Scala

Scala has the richest set of fold operations, in part because it facilitate scenarios that don't appear in the dynamically typed Groovy and Clo commonly used to perform sums:

The `reduceLeft()` function assumes that the first element is the left si tion. For operators such as plus, the placement of operands is irrele matters for operations such as divide. If you want to reverse the ord operator is applied, use `reduceRight()`:

```
List.range(1, 10) reduceRight(_ - _)
```

Thus, it applies 8 - 9 first, then uses that result as the *second* paramete
calculations.

Understanding when you can use higher-level abstractions such as redu
keys to mastery of functional programming. This example uses reduc
termine the longest word in a collection:

```
words.reduceLeft((a, b) => if (a.length > b.length) a else b)
// jumped
```

The reduce and fold operations have overlapping functionality, with sub
that I discussed earlier. One obvious difference suggests common use
signature of `reduceLeft[B >: A](op: (B, A) => B): B` shows that the
that's expected is the function to combine elements. The initial value is
the first value in the collection. In contrast, the signature of `foldLeft`
`(B, A) => B): B` indicates an initial seed value for the result, so you c
that are different from the type of the list elements.

Here's an example of summing a collection by using `foldLeft()`:

```
List.range(1, 10).foldLeft(0)(_ + _)
// 45
```

Scala supports operator overloading, so the two common fold operation
`foldRight`, have corresponding operators: `/:` and `:\`, respectively. Thus
a terser version of sum by using `foldLeft`:

```
(0 /: List.range(1, 10)) (_ + _)
// 45
```

commonplace. Many of the features of functional languages were proh
decade ago but make perfect sense now because they optimize developer

One of the values of functional thinking is the ability to cede control of l
(such as garbage collection) to the runtime, eliminating a swath of bugs
While many developers are accustomed to blissful ignorance for bedro
such as memory, they are less accustomed to similar abstractions appea
level. Yet these higher-level abstractions serve the same purpose: handlin

In this chapter, I show five ways developers in functional languages can
the language or runtime, freeing themselves to work on more relevant

Iteration to Higher-Order Functions

I already illustrated the first example of surrendering control in Exampl
iteration with functions such as `map`. What's ceded here is clear: if you ca
operation you want to perform in a higher-order function, the langua
efficiently, including the ability to parallelize the operation with the add
modifier.

high performance, even with the new `Stream` API in Java 8. Once you
however, you can apply that power in a more succinct way.

> Always understand one level below your normal abstraction l

Programmers rely on abstraction layers to be effective: no one program

Closures

All functional languages include closures, yet this language feature is oft
almost mystical terms. A *closure* is a function that carries an implicit bi
variables referenced within it. In other words, the function (or method)
text around the things it references.

Here is a simple example, written in Groovy, of creating a closure block
shown in Example 3-1.

Example 3-1. Simple closure binding in Groovy

```
class Employee {
  def name, salary
}

def paidMore(amount) {
  return {Employee e -> e.salary > amount}
}

isHighPaid = paidMore(100000)
```

In Example 3-1, I define a simple Employee class with two fields. T
paidMore function that accepts a parameter amount. The return of this fu
block, or *closure*, accepting an Employee instance. The type declaration
serves as useful documentation in this case. I can assign this code blo
isHighPaid, supplying the parameter value of 100,000. When I make t
I *bind* the value of 100,000 to this code block forever. Thus, when I eval
via this code block, it will opine about their salaries by using the perm
value, as shown in Example 3-2.

Example 3-2. Executing the closure block

```
def Smithers = new Employee(name:"Fred", salary:120000)
def Homer = new Employee(name:"Homer", salary:80000)
println isHighPaid(Smithers)
println isHighPaid(Homer)
// true, false
```

In Example 3-2, I create a couple of employees and determine if their sa
criterion. When a closure is created, it creates an enclosure around the
enced within the scope of the code block (thus the name *closure*). Each
closure block has unique values, even for private variables. For examp
another instance of my paidMore closure with another binding(and the
assignment), as shown in Example 3-3.

Example 3-3. Another closure binding

```
isHigherPaid = paidMore(200000)
println isHigherPaid(Smithers)
println isHigherPaid(Homer)
def Burns = new Employee(name:"Monty", salary:1000000)
println isHigherPaid(Burns)
// false, false, true
```

Closures are used quite often as a portable execution mechanism in funct
and frameworks, passed to higher-order functions such as map() as the
code. Functional Java uses anonymous inner classes to mimic some o
behavior, but they can't go all the way because Java before version 8 did
sures. But what does that mean?

Example 3-4 shows an example of what makes closures so special.

Example 3-4. When the closure block is garbage collected, all the refere
reclaimed as well.

It's a bad idea to create a closure just so that you can manipulate its int
done here to illustrate the inner workings of closure bindings. Bindi
immutable values (as shown in Example 3-1) is more common.

The closest you could come to the same behavior in Java prior to Java 8
language that has functions but not closures, appears in Example 3-5.

Several variants of the `Counter` class are possible (creating anonymous
generics, etc.), but you're still stuck with managing the state yourself.
why the use of closures exemplifies functional thinking: allow the runt
state. Rather than forcing yourself to handle field creation and babying
the horrifying prospect of using your code in a multithreaded environm
control and let the language or framework invisibly manage that state f

> Let the language manage state.

Closures are also an excellent example of *deferred execution*. By binding c
block, you can wait until later to execute the block. This turns out to be
scenarios. For example, the correct variables or functions might not be
inition time but are at execution time. By wrapping the execution cont
you can wait until the proper time to execute it.

Imperative languages use *state* to model programming, exemplified by p
ing. Closures allow us to model *behavior* by encapsulating both code a
a single construct, the closure, that can be passed around like traditional
and executed at exactly the correct time and place.

> Capture the *context*, not the *state*.

Currying and Partial Application

Currying and partial application are language techniques derived fro
(based on work by twentieth-century mathematician Haskell Curry and
techniques are present in various types of languages and are omniprese
languages in one form or another. Both currying and partial applicati
ability to manipulate the number of arguments to functions or metho
supplying one or more default values for some arguments (known as fixi
Most functional languages include currying and partial application, but
them in different ways.

To the casual observer, currying and partial application appear to hav
fect. With both, you can create a version of a function with presupplied
of the arguments:

- *Currying* describes the conversion of a multiargument function
  single-argument functions. It describes the transformation proces
  cation of the converted function. The caller can decide how man
  apply, thereby creating a derived function with that smaller numbe
- *Partial application* describes the conversion of a multiargument fu
  that accepts fewer arguments, with values for the elided argume
  advance. The technique's name is apt: it partially applies some argu
  tion, returning a function with a signature that consists of the remai

With both currying and partial application, you supply argument valu
function that's invokable with the missing arguments. But currying a f
the next function in the chain, whereas partial application binds argu
values that you supply during the operation, producing a function with
(number of arguments). This distinction becomes clearer when you con
with arity greater than two.

For example, the fully curried version of the `process(x, y, z)` function
`(y)(z)`, where both `process(x)` and `process(x)(y)` are functions that
argument. If you curry only the first argument, the return value of p
function that accepts a single argument that in turn accepts a single arg
trast, with partial application, you are left with a function of smaller arit
application for a single argument on `process(x, y, z)` yields a funct
two arguments: `process(y, z)`.

The two techniques' results are often the same, but the distinction is
often misconstrued. To complicate matters further, Groovy implemen
application and currying but calls both currying. And Scala has both p
functions and the `PartialFunction` class, which are distinct concepts
ilar names.

In Groovy

Scala

Scala supports currying and partial application, along with a trait tha
ability to define constrained functions.

Currying

In Scala, functions can define multiple argument lists as sets of parenth
call a function with fewer than its defined number of arguments, the retu
that takes the missing argument lists as its arguments. Consider the ex
Scala documentation that appears in Example 3-10.

Example 3-10. Scala's currying of arguments

In Example 3-11, I first create a price() function that returns a ma
product and price. Then I create a withTax() function that accepts c
arguments. However, within a particular source file, I know that I will w
with one state's taxes. Rather than "carry" the extra argument for ever
partially apply the state argument and return a version of the function in
value is fixed. The locallyTaxed() function accepts a single argument

Partial (constrained) functions

The Scala PartialFunction trait is designed to work seamlessly with pa
which is covered in detail in Chapter 6. Despite the similarity in nam
Function trait does not create a partially applied function. Instead, yo
define a function that works only for a defined subset of values and typ

Case blocks are one way to apply partial functions. Example 3-12 us
without the traditional corresponding match operator.

Example 3-12. Using `case` without `match`

In Example 3-12, I create a map of city and state correspondence. Th
map() function on the collection, and map() in turn pulls apart the ke
print them. In Scala, a code block that contains case statements is one
an anonymous function. You can define anonymous functions more co
using case, but the case syntax provides the additional benefits tha
illustrates.

MatchError as the function tries to increment the "seven" string. But co
correctly. Why the disparity and where did the error go?

Case blocks define partial functions, but not partially applied function
tions have a limited range of allowable values. For example, the mathem
1/x is invalid if x = 0.

Partial functions offer a way to define constraints for allowable valu
lect() invocation in Example 3-13, the case is defined for Int, but no
the "seven" string isn't collected.

In Example 3-16, I define a partial function to accept any type of input (A
to react to a subset of types. However, notice that I can also call the
function for the partial function. Implementers of the PartialFunctio
case can call isDefinedAt(), which is implicitly defined. Example 3-13
map() and collect() behave differently. The behavior of partial functio

difference: collect() is designed to accept partial functions and to ca
dAt() function for elements, ignoring those that don't match.

Partial functions and partially applied functions in Scala are similar in
offer a different set of orthogonal features. For example, nothing pre
partially applying a partial function.

Common Uses

application do have a place in real-world programming.

Function factories

Currying (and partial application) work well for places where you impl
function in traditional object-oriented languages. To illustrate, Examp
ments a simple adder function in Groovy.

Example 3-17. Adder and incrementer in Groovy

In Example 3-17, I use the adder() function to derive the incrementer

Template Method design pattern

One of the Gang of Four design patterns is the Template Method patter
to help you define algorithmic shells that use internal abstract methods
implementation flexibility. Partial application and currying can solve the
Using partial application to supply known behavior and leaving the o
free for implementation specifics mimics the implementation of this
design pattern.

I show an example of using partial application and other functional tec
recate several design patterns (including Template Method) in Chapter

Implicit values

When you have a series of function calls with similar argument valu
currying to supply implicit values. For example, when you interact wit
framework, you must pass the data source as the first argument. By usi
cation, you can supply the value implicitly, as shown in Example 3-18.

Recursion

Recursion, which (according to Wikipedia) is the "process of repeating
similar way," is another example of ceding details to the runtime, and i
ciated with functional programming. In reality, it's a computer-science
over things by calling the same method from itself, reducing the collec
and always carefully ensuring you have an exit condition. Many times,
to easy-to-understand code because the core of your problem is the need
thing over and over to a diminishing list.

Seeing Lists Differently

Figure 3-1. Lists as indexed slots

Many functional languages have a slightly different perspective on lists,
Groovy shares this perspective. Instead of thinking of a list as indexed s
as a combination of the first element in the list (the head) plus the rema
(the tail), as shown in Figure 3-2.

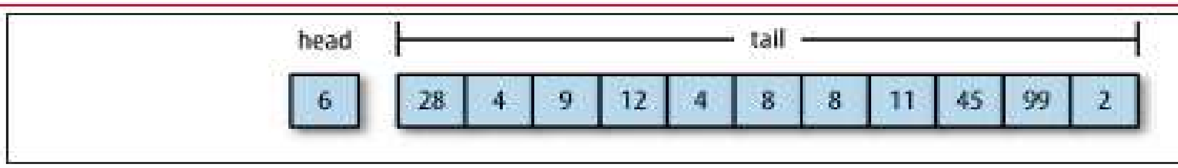

Figure 3-2. A list as its head and tail

Thinking about a list as head and tail allows me to iterate through it usi
shown in Example 3-20.

If not, I print out the first element in the list, available via Groovy's head
then recursively call the recurseList() method on the remainder of th

Recursion often has technical limits built into the platform, so this te
panacea. But it should be safe for lists that contain a small number of it

The difference between Examples 3-21 and 3-22 highlights an impo
Who's minding the state? In the imperative version, *I* am. *I* must create
named new_list, *I* must add things to it, and *I* must return it when I'
recursive version, the *language* manages the return value, building it up
the recursive return for each method invocation. Notice that every ex
filter() method in Example 3-22 is a return call, which builds up th
value on the stack. You can cede responsibility for new_list; the langua
it for you.

Recursion allows you to cede state management to the runtin

The list-construction operators in Scala make the return conditions for
readable and easy to understand. The code in Example 3-23 is one of t
both recursion and currying from the Scala documentation. The fil
recursively filters a list of integers via the parameter p, a predicate functi
term in the functional world for a Boolean function. The filter() me
see if the list is empty and, if it is, simply returns; otherwise, it checks th
in the list (xs.head) via the predicate to see if it should be included in

Tail-Call Optimization

One of the major reasons that recursion isn't a more commonplace ope
growth. Recursion is generally implemented to place intermediate result
and languages not optimized for recursion will suffer stack overflow. La
as Scala and Clojure have worked around this limitation in various ways.
way that developers can help runtimes handle this problem is *tail-call*
When the recursive call is the last call in the function, runtimes can oft
results on the stack rather than force it to grow.

Many functional languages (such as Erlang), implement tail recursion
growth. Tail recursion is used to implement long-running Erlang proces

My guess is that you don't use recursion at all now—it's not even a part o
However, part of the reason lies in the fact that most imperative languages
support for it, making it more difficult to use than it should be. By addi
and support, functional languages make recursion a candidate for simp

Streams and Work Reordering

before filter(). When thinking imperatively, the instinct is to place t
eration before the mapping operation, so that the map has less work to pe
a smaller list. However, many functional languages (including Java 8
Functional Java framework) define a Stream abstraction. A Stream act
like a collection, but it has no backing values, and instead uses a stream
a source to a destination. In Example 3-24, the source is the names col
destination (or "terminal") is collect(). Between these operations, b
filter() are *lazy*, meaning that they defer execution as long as possib
don't try to produce results until a downstream terminal "asks" for the

For the lazy operations, intelligent runtimes can reorder the result of th
you. In Example 3-24, the runtime can flip the order of the lazy operat
more efficient, performing the filtering before the mapping. As with m
additions to Java, you must ensure that the lambda blocks you pass to
as filter() don't have side effects, which will lead to unpredictable res

Allowing the runtime to optimize when it can is another great example of
giving away mundane details and focusing on the problem domain r
implementation of the problem domain.

I discuss laziness in more detail in Chapter 4 and Java 8 streams in Cha

Memoization

The word memoization was coined by Donald Michie, a British artific researcher, to refer to function-level caching for repeating values. Toda is common in functional programming languages, either as a built-in that's relatively easy to implement.

Memoization helps in the following scenario. Suppose you have a intensive function that you must call repeatedly. A common solution internal cache. Each time you calculate the value for a certain set of para that value in the cache, keyed to the parameter value(s). In the future, if invoked with previous parameters, return the value from the cache ra culate it. Function caching is a classic computer science trade-off: it uses (which we frequently have in abundance) to achieve better performanc

Functions must be *pure* for the caching technique to work. A *pure* func has no side effects: it references no other mutable class fields, doesn't set a than the return value, and relies only on the parameters for input. All the `java.lang.Math` class are excellent examples of pure functions. Obv reuse cached results successfully only if the function reliably returns t for a given set of parameters.

Adding Memoization

Functional programming strives to minimize moving parts by building anisms into the runtime. Memoization is a feature built into a program that enables automatic caching of recurring function-return values. In automatically supplies the code I've written in Examples 4-1 and 4-3. languages support memoization, including Groovy.

In order to memoize a function in Groovy, you define it as a closure, t `memoize()` method to return a function whose results will be cached.

Memoizing a function is a *metafunction* application: doing something itself rather than the function results. Currying, discussed in Chapter 3 ample of a metafunction technique. Groovy built memoization into its other languages implement it differently.

Memoized	956 ms
Memoized (2nd)	19 ms

Memoizing everything slows down the first run but has the fastest sub any case—but only for small sets of numbers. As with the imperative ca tested in Example 4-3, large number sets impede performance drastica memoized version runs out of memory in the 8,000-number case. But fo approach to be robust, safeguards and careful awareness of the execut required—another example of imperative moving parts. With memoiza tion occurs at the function level. Look at the memoization results for 1 found in Table 4-5.

Table 4-5. Results for range 1–10,000

`memoizeBetween()`	Creates a caching variant of the closure with automatic cache size adjustment a and upper limits on the cache size

In the imperative version, the developer owns the code (and responsibil languages build generic machinery—sometimes with customization kn of alternate functions or parameters)—that you can apply to standard co tions are a fundamental language element, so optimizing at that level give functionality for free. The memoization versions in this chapter with sm outperform the handwritten caching code handily. In fact, I'll never be cache as efficient as the language designers can because they can bend t language designers have access to low-level parts that developers don't timization opportunities beyond the grasp of mere mortals. Not only ca handle caching more efficiently, I want to cede that responsibility to th can think about problems at a higher level of abstraction.

> Language designers will always build more efficient mechanis cause they are allowed to bend rules.

Building a cache by hand is straightforward, but it adds statefulness and the code. Using functional-language features like memoization, I can ad function level, achieving better results (with virtually no change to my imperative version. Functional programming eliminates moving parts, focus your energy on solving real problems.

Of course, you don't have to rely on an existing class to layer memoizat the memoized function via an explicit invocation to `call()`.

Most functional languages either include memoization or make it trivia
For example, memoization is built into Clojure; you can memoize any fu
the built-in `(memoize )` function. For example, if you have an existing `(h
you can memoize it via `(memoize (hash "homer"))` for a caching versio
implements the name-hashing algorithm from Example 4-6 in Clojure

call() method. In the Clojure version, the memoized method call is e
on the surface, with the added indirection and caching invisible to the

Scala doesn't implement memoization directly but has a collection r
getOrElseUpdate() that handles most of the work of implementing
Example 4-9.

Example 4-9. Memoization implementation in Scala

```
def memoize[A, B](f: A => B) = new (A => B) {
    val cache = scala.collection.mutable.Map[A, B]()
    def apply(x: A): B = cache.getOrElseUpdate(x, f(x))
  }

def nameHash = memoize(hash)
```

The getOrElseUpdate() function in Example 4-9 is the perfect operato
cache: it either retrieves the matching value or creates a new entry whe

It is worth reiterating the importance of immutability for anything you m
memoized function relies on anything other than parameters to generat
will receive unpredictable outcomes. If your memoized function has s
won't be able to rely on that code executing when the cached value is re

Make sure all memoized functions:

- Have no side effects
- Never rely on outside information

As runtimes become more sophisticated and we have plenty of machi
our disposal, advanced features such as memoization become commo
every mainstream language. For example, although Java 8 doesn't includ
ization, it is easy to implement it atop the new lambda features.

Laziness

Lazy evaluation—deferral of expression evaluation for as long as possib
of many functional programming languages. Lazy collections deliver th
needed rather than precalculating them, offering several benefits. First
expensive calculations until they're absolutely needed. Second, you can
collections, which keep delivering elements as long as they keep rec
Third, lazy use of functional concepts such as map and filter enable y
more efficient code. Java doesn't natively support laziness until Java
frameworks and successor languages do.

Consider the snippet of pseudocode for printing the length of a list in E

Example 4-10. Pseudocode illustrating nonstrict evaluation

```
print length([2+1, 3*2, 1/0, 5-4])
```

If you try to execute this code, the result will vary depending on the type o
language it's written in: *strict* or *nonstrict* (also known as lazy). In a stric
language, executing (or perhaps even compiling) this code results in a
ception because of the list's third element. In a nonstrict language, the re
accurately reports the number of items in the list. After all, the metho
`length()`, not `lengthAndThrowExceptionWhenDivByZero()`! Haskell is
nonstrict languages in common use. Alas, Java doesn't support nonstrict
you can still take advantage of the concept of laziness in Java by deferr
and some next-generation languages are lazier than Java by default.

Functional programming languages approach code reuse differentl
oriented languages. Object-oriented languages tend to have many data
many operations, whereas functional languages exhibit few data struct
operations. Object-oriented languages encourage you to create class-sp
and you can capture recurring patterns for later reuse. Functional lang
achieve reuse by encouraging the application of common transformatio
tures, with higher-order functions to customize the operation for speci

In this chapter, I cover various ways that languages have *evolved* solutio
recurring problems in software. I discuss the attitudinal change in funct

seminal work in that space, *Design Patterns: Elements of Reusable Objec*
ware (Addison-Wesley, 1994), includes at least one class diagram with
the OOP world, developers are encouraged to create unique data struc
cific operations attached in the form of methods. Functional program
don't try to achieve reuse in the same way. They prefer a few key data s
as `list`, `set`, and `map`) with highly optimized operations on those data
pass data structures plus higher-order functions to "plug into" t

---

customizing it for a particular use. For example, the `filter()` functio
several languages accepts a code block as the "plug-in" higher-order fu
termines the filter criteria, and the machinery applies the filter criteri
way, returning the filtered list.

Encapsulation at the function level enables reuse at a more granular, fu
than building custom class structures. One advantage of this approac
pearing in Clojure. For example, consider the case of parsing XML. A
frameworks exist for this task in Java, each with custom data structu
semantics (for example, SAX versus DOM). Clojure parses XML into