Praise for Design Patterns in Ruby

"***Design Patterns in Ruby*** documents smart ways to resolve many problems that Ruby developers commonly encounter. Russ Olsen has done a great job of selecting classic patterns and augmenting these with newer patterns that have special relevance for Ruby. He clearly explains each idea, making a wealth of experience available to Ruby developers for their own daily work."

—Steve Metsker, Managing Consultant
with Dominion Digital, Inc.

"This book provides a great demonstration of the key 'Gang of Four' design patterns without resorting to overly technical explanations. Written in a precise, yet almost informal style, this book covers enough ground that even those without prior expo sure to design patterns will soon feel confident applying them using Ruby. Olsen has done a great job to make a book about a classically 'dry' subject into such an engag ing and even occasionally humorous read."

—Peter Cooper

"This book renewed my interest in understanding patterns after a decade of good intentions. Russ picked the most useful patterns for Ruby and introduced them in a straightforward and logical manner, going beyond the GoF's patterns. This book has improved my use of Ruby, and encouraged me to blow off the dust covering the GoF book."

—Mike Stok

"***Design Patterns in Ruby*** is a great way for programmers from statically typed object oriented languages to learn how design patterns appear in a more dynamic, flexible language like Ruby."

—Rob Sanheim, Ruby Ninja, Relevance

This page intentionally left blank

Design Patterns in Ruby

Addison-Wesley Professional Ruby Series **Obie Fernandez, Series Editor**

The Addison-Wesley Professional Ruby Series provides readers with practical, people-oriented, and in-depth information about applying the Ruby platform to create dynamic technology solutions. The series is based on the premise that the need for expert reference books, written by experienced practitioners, will never be satisfi ed solely by blogs and the

Internet.

Books currently in the series

RailsSpace: Building a Social Networking Website with Ruby on Rails™
Michael Hartl, Aurelius Prochazka • 0321480791 • ©2008

The Ruby Way: Solutions and Techniques in Ruby Programming, Second Edition Hal Fulton • 0672328844 • ©2007

Professional Ruby Collection: Mongrel, Rails Plugins, Rails Routing, Refactoring to REST, and Rubyisms CD1
James Adam, David A. Black, Trotter Cashion, Jacob Harris, Matt Pelletier, Zed Shaw 0132417995 • ©2007

The Rails Way
Obie Fernandez • 0321445619 • ©2008

Mongrel: Developing, Extending and Deploying Your Ruby Applications *Coming Spring 2008* Zed Shaw • 0321503090 • ©2008

Short Cuts

Rails Routing
David A. Black • 0321509242 • ©2007

Rails Refactoring to Resources: Using CRUD and REST in Your Rails Application Trotter Cashion • 0321501748 • ©2007

Mongrel: Serving, Deploying and Extending Your Ruby Applications
Matt Pelletier and Zed Shaw • 0321483502 • ©2007

Rails Plugins: Extending Rails Beyond the Core
James Adam • 0321483510 • ©2007

Rubyism in Rails
Jacob Harris • 0321474074 • ©2007

Troubleshooting Ruby Processes: Leveraging System Tools when the Usual Ruby Tricks Stop Working Philippe Hanrigou • 0321544684 • ©2008

Writing Effi cient Ruby Code
Dr. Stefan Kaes • 0321540034 • ©2008

Video

RailsSpace Ruby on Rails Tutorial (Video LiveLessons)
Aurelius Prochazka • 0321517067 • ©2008

www.awprofessional.com/ruby

DESIGN PATTERNS IN RUBY

Russ Olsen

Upper Saddle River, NJ • Boston • Indianapolis • San
Francisco New York • Toronto • Montreal • London • Munich •
Paris • Madrid Capetown • Sydney • Tokyo • Singapore •
Mexico City

Visit us on the Web: www.awprofessional.com
Library of Congress Cataloging-in-Publication Data
Olsen, Russ.
Design patterns in Ruby/Russ Olsen.
p. cm.
Includes index.
ISBN 978-0-321-49045-2 (hbk. : alk. paper)
1. Ruby on rails (Electronic resource) 2. Software patterns. I. Title.

QA76.64.O456 2007
005.1'17—dc22 2007039642

ISBN-13: 978-0-321-29045-2
ISBN-10: 0-321-49045-2

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts. First printing, December 2007

Editor-in-Chief: Karen Gettman Acquisitions Editor: Chris Guzikowski

Managing Editor: John Fuller Project Editor: Lara Wysong Copy Editor: Jill Hobbs Indexer: Ted Laux

Proofreader: Deborah Prato Composition: International Typesetting and Composition

For Karen,
who makes it all possible,

and Jackson,
who makes it all worthwhile.

This page intentionally left blank

Contents

Foreword

Design Patterns: Elements of Reusable Object-Oriented Software, affectionately known by many as the "Gang of Four book" (GoF) is the first reference work on the topic to be published in a mainstream book. It has sold over half a million copies since 1995 and undoubtedly influenced the thoughts and code of millions of programmers worldwide. I still vividly remember buying my first copy of the book in the late nineties. Due in part to the enthusiasm with which it was recommended to me by my peers, I treated it as part of my coming-of-age as a programmer. I tore through the book in a few days, eagerly thinking up practical applications for each pattern.

It's commonly agreed that the most useful thing about patterns is the way in which they form a vocabulary for articulating design decisions during the normal course of development conversations among programmers. This is especially true during pair programming, a cornerstone of Extreme Programming and other Agile processes, where design is an ongoing and shared activity. It's fantastically convenient to be able to say to your pair, "I think we need a strategy here" or "Let's add this functionality as an observer."

Knowledge of design patterns has even become an easy way to screen programming job candidates in some shops, where it's common to hear:

"What's your favorite pattern?"

"Um . . . factory?"

"Thanks for coming, there's the door."

Then again, the whole notion of having a *favorite* pattern is kind of strange isn't it? Our favorite pattern should be the one that applies to a given circumstance. One of the

classic mistakes made by inexperienced programmers just beginning to learn about pat terns is to choose to implement a pattern as an end of its own, rather than as a means. Why do people get wrapped up in

implementing patterns “just for fun” anyway?

At least in the statically typed world, there are a fair amount of technical chal lenges to tackle when you implement design patterns. At best, you use some ninja techniques that really show your coding prowess. Worst case scenario you end up with a bunch of boilerplate gunk. It makes the topic of design patterns a fun one, at least for programming geeks like me.

Are the GoF design patterns difficult to implement in Ruby? Not really. For starters, the absence of static typing lowers the code overhead involved in our programs overall. The Ruby standard library also makes some of the most common patterns available as one-line includes, and others are essentially built into the Ruby language itself. For instance, a Command object in the GoF sense is essentially a wrapper around some code that knows how to do one specific thing, to run a particular bit of code at some time. Of course, that is also a fairly accurate description of a Ruby code block object or a Proc.

Russ has been working with Ruby since 2002 and he knows that most experienced Rubyists already have a good grasp of design patterns and how to apply them. Thus his main challenge, as far as I can tell, was to write this book in such a way that it would be relevant and essential for professional Ruby programmers, yet still benefit newcomers to our beloved language. I think he has succeeded, and you will, too. Take the Command object example again: In its simple form it may be implemented with simply a block, but add state and a bit of behavior to it and now the implementation is not so simple anymore. Russ gives us proven advice that is specific to Ruby and instantly useful.

This book also has the added benefit of including new design patterns specific to Ruby that Russ has identified and explained in detail, including one of my favorite ones: Internal Domain Specific Languages. I believe that his treatment of the subject, as an evolution of the Interpreter pattern, is the first significant reference work in pub lication on the topic.

Finally, I think this book will hugely benefit those that are just beginning their professional careers in Ruby or migrating from languages such as PHP, where there isn’t as much of a cultural emphasis on OO design and patterns. In the process of describing design patterns, Russ has captured the essence of solving many of the common programming hurdles that we face in day-to-day programming of significant Ruby programs—priceless information for newbies. So much so that I’m sure that this book will be a staple of my gift-list for new programmer colleagues and friends.

—Obie Fernandez, Professional Ruby Series Editor

Preface

A former colleague of mine used to say that thick books about design patterns were evidence of an inadequate programming language. What he meant was that, because design patterns are the common idioms of code, a good programming language should make them very easy to implement. An ideal language would so thoroughly integrate the patterns that they would almost disappear from sight.

To take an extreme example, in the late 1980s I worked on a project that pro duced object-oriented code in C. Yes, C, not C++. We pulled off this feat by having each “object” (actually a C structure) point to a table of function pointers. We oper ated on our “objects” by chasing the pointer to the table and calling functions out of the table, thereby simulating a method call on an object. It was awkward and messy, but it worked. Had we thought of it, we might have called this technique the “object-oriented” pattern. Of course, with the advent of C++ and then Java, our object-oriented pattern disappeared, absorbed so thoroughly into the language that it vanished from sight. Today, we don’t usually think of object orientation as a pattern— it is too easy.

But many things are still not easy enough. The justly famous Gang of Four book (*Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, and Vlissides) is required reading for every software engineer today. But actu ally implementing many of the patterns described in *Design Patterns* with the lan guages in widespread use today (Java and C++ and perhaps C#) looks and feels a lot like my 1980s-vintage handcrafted object system. Too painful. Too verbose. Too prone to bugs.

The Ruby programming language takes us a step closer to my old friend’s ideal, a language that makes implementing patterns so easy that sometimes they fade into the background. Building patterns in Ruby is

easier for a number of reasons:

- Ruby is dynamically typed. By dispensing with static typing, Ruby dramatically reduces the code overhead of building most programs, including those that imple ment patterns.
- Ruby has code closures. It allows us to pass around chunks of code and associated scope without having to laboriously construct entire classes and objects that do nothing else.
- Ruby classes are real objects. Because a class in Ruby is just another object, we can do any of the usual runtime things to a Ruby class that we can do to any other object: We can create totally new classes. We can modify existing classes by adding or deleting methods. We can even clone a class and change the copy, leaving the original alone.
- Ruby has an elegant system of code reuse. In addition to supporting garden-variety inheritance, Ruby allows us to define mixins, which are a simple but flexible way to write code that can be shared among several classes.

All of this makes code in Ruby compressible. In Ruby, as in Java and C++, you can implement very sophisticated ideas, but with Ruby it becomes possible to hide the details of your implementations much more effectively. As you will see on the pages that follow, many of the design patterns that require many lines of endlessly repeated boilerplate code in traditional static languages require only one or two lines in Ruby. You can turn a class into a singleton with a simple include Singleton command. You can delegate as easily as you can inherit. Because Ruby enables you to say more inter esting things in each line of code, you end up with less code.

This is not just a question of keyboard laziness; it is an application of the DRY (Don't Repeat Yourself) principle. I don't think anyone today would mourn the pass ing of my old object-oriented pattern in C. It worked for me, but it made me work for it, too. In the same way, the traditional implementations of many design patterns work, but they make you work, too. Ruby represents a real step forward in that you become able to do work only once and compress it out of the bulk of your code. In short, Ruby allows you to concentrate on the real problems that you are trying to solve instead of the plumbing. I hope that this book will help you see how.

Who Is This Book For?

Simply put, this book is intended for developers who want to know how to

build sig nificant software in Ruby. I assume that you are familiar with object-oriented pro gramming, but you don't really need any knowledge of design patterns—you can pick that up as you go through the book.

You also don't need a lot of Ruby knowledge to read this book profitably. You will find a quick introduction to the language in Chapter 2, and I try to explain any Ruby specific language issues as we go.

How Is This Book Organized?

This book is divided into three parts. First come a couple of introductory chapters, starting with the briefest outline of the history and background of the whole design patterns movement, and ending with a quick tour of the Ruby language at the "just enough to be dangerous" level.

Part 2, which takes up the bulk of these pages, looks at a number of the original Gang of Four patterns from a Ruby point of view. Which problem is this pattern try ing to solve? What does the traditional implementation of the pattern—the imple mentation given by the Gang of Four—look like in Ruby? Does the traditional implementation make sense in Ruby? Does Ruby provide us with any alternatives that might make solving the problem easier?

Part 3 of this book looks at three patterns that have emerged with the introduc tion and expanded use of Ruby.

A Word of Warning

I cannot sign my name to a book about design patterns without repeating the mantra that I have been muttering for many years now: Design patterns are little spring-loaded solutions to common programming problems. Ideally, when the appropriate problem comes along, you should trigger the design pattern and your problem is solved. It is that first part—the bit about waiting for the appropriate problem to come along—that some engineers have trouble with. You cannot say that you are correctly applying a design pattern *unless you are confronting the problem that the pattern is supposed to solve*.

The reckless use of every design pattern on the menu to solve nonexistent prob lems has given design patterns a bad name in some circles. I would contend that Ruby

makes it easier to write an adapter that uses a factory method to get a proxy to the builder, which then creates a command, which will coordinate the operation of adding two plus two. Ruby will make that process easier,

but even in Ruby it will not make any sense.

Nor can you look at program construction as a simple process of piecing together some existing design patterns in new combinations. Any interesting program will always have unique sections, bits of code that fit that specific problem perfectly and no other. Design patterns are meant to help you recognize and solve the common problems that arise repeatedly when you are building software. The advantage of design patterns is that they let you rapidly wing your way past the problems that some one has already solved, so that you can get on to the hard stuff, the code that is unique to your situation. Design patterns are not the universal elixir, the magic potion that will fix all of your design problems. They are simply one technique—albeit a very use ful technique—that you can use to build programs.

About the Code Style Used in This Book

One thing that makes programming in Ruby so pleasant is that the language tries to stay out of your way. If there are several sensible ways of saying something, Ruby will usually support them all:

```
# One way to say it

if (divisor == 0)
   puts 'Division by zero'
end

# And another

puts 'Division by zero' if (divisor == 0)

# And a third

(divisor == 0) && puts 'Division by zero'
```

Ruby also tries not to insist on syntax for syntax's sake. Where possible, it will let you omit things when the meaning is clear. For example, you can usually omit the parentheses around the argument list when calling a method:


```
puts('A fine way to call puts')
puts 'Another fine way to call puts'
```

You can even forget the parentheses when you are defining the argument list of a method and around the conditional part of an if statement:

```
def method_with_3_args a, b, c
   puts "Method 1 called with #{a} #{b} #{c}"
   if a == 0
      puts 'a is zero'
   end
end
```

The trouble with all of these shortcuts, convenient as they are in writing real Ruby programs, is that when liberally used, they tend to confuse beginners. Most program mers who are new to Ruby are going to have an easier time with

```
if file.eof?
   puts( 'Reached end of file' )
end
```

or even

```
puts 'Reached end of file' if file.eof?
```

than with

```
file.eof? || puts('Reached end of file')
```

Because this book is more about the deep power and elegance of Ruby than it is about the details of the language syntax, I have tried to strike a balance between mak ing my examples actually look like real Ruby code on the one hand while still being beginner friendly on the other hand. In practice, this means that while I take advan tage of some obvious shortcuts, I have deliberately avoided the more radical tricks. It is not that I am unaware of, or disapprove of, the Ruby syntactical shorthand. It is just that I am more interested getting the conceptual elegance of the language across to readers who are new to Ruby. There will be plenty of time to learn the syntactical shortcuts after you have fallen hopelessly in love with the language.

This page intentionally left blank

Acknowledgments

I have always thought that whoever said that no man is an island got it pretty much backward. Most islands are the very tops of underwater mountains, the tiny green part that you see supported by a massive and invisible structure just below the waves. So, if the point is that no one does anything unaided, and that credit for everything we do should be spread among our friends and families and colleagues, then we are all islands, propped up by folks who help but who are not seen. Certainly this book would have never gotten started or completed without the help of a mountain of very generous people.

I would like to particularly thank my good friend Bob Kiel, who probably told me no less than 17,827 times that I should write a book. Thanks also to Xandy Johnson, for his generous support and encouragement throughout the writing of this book.

I would also like to say “thank you” to everyone who reviewed this book in its various drafts, including the aforementioned Bob and Xandy, along with Mike Abner, Geoff Adams, Peter Cooper, Tom Corbin, Bill Higgins, Jason Long, Steve Metsker, Glenn Pruitt, Rob Sanheim, Mike Stok, and Gary Winklosky. And special thanks to Andy Lynn and Arild Shirazi, who both went over early drafts of the man uscript with an invaluable, if sometimes painful, fine-toothed comb. Special thanks to Rob Cross for finding that “last” typo.

Thanks also to Heli Roosild, a very professional writer who took the time to look over some things I had written and said, “Yes, this will do.”

Thanks also to Lara Wysong, Raina Chrobak, and Christopher Guzikowski, all of Addison-Wesley—especially Chris, who read a 900-word blog article and saw a 384-page

book. Thanks also to Jill Hobbs, who copyedited this book with a sharp eye and an even sharper pen.

I’d also like to thank the fine folks at FGM for creating the kind of intellectual environment that makes efforts like this book possible.

Thanks, too, to Steve McMaster and his band of merry men at SamSix

for their support and encouragement.

On a more personal level, I would like to thank my wife Karen for her encour agement and suggestions, and for lending me the end of the kitchen table for all those months. Thanks, too, to my son Jackson for letting me tell stories about him here and there in these pages. Many thanks to Diana Greenberg, a good friend in the best of times, and a great friend in the worst of times.

Thanks to my brother Charles for setting an example of courage and persistence that I try to live up to every day.

Finally, thanks to my sister Dolores for awakening my interest in reading. I clearly remember the day she dragged me across the library, away from the trash I had been browsing and over to a shelf of serious books. She pulled one down and said, "*This* is the kind of thing you should be reading." I can still picture the book. It was *The Rise and Fall of the Third Reich* by William L. Shirer, all 1,264 pages of it. I think I was seven at the time.

About the Author

Russ Olsen is a software engineer with more than twenty-five years of experience. Russ has built software in such diverse areas as CAD/CAM, GIS, document management, and enterprise integration. Currently, he is involved in building SOA service discovery and security solutions for large enterprises.

Russ has been involved in Ruby since 2002 and was the author of ClanRuby, an early attempt to bring multimedia capabilities to Ruby. He is currently a committer on a number of open source projects, including UDDI4R.

Russ's technical articles have been featured on Javalobby, O'Reilly's On Java, and the Java Developer's Journal Web site.

Russ holds a B.S. from Temple University and lives with his family, two turtles, and an indeterminate number of guppies outside of Washington, D.C. You can reach Russ via e-mail at russ@russolsen.com.

This page intentionally left blank

PART I
Patterns and Ruby

This page intentionally left blank

CHAPTER 1
Building Better Programs with Patterns

It's funny, but design patterns always remind me of a certain grocery store. You see, my very first steady job was a part-time gig that I landed in high school. For a couple of hours every weekday and all day on Saturday, I would help out at the local mom and-pop grocery store. I stocked the shelves, swept the floor, and generally did what ever unskilled labor

needed doing. At first, the goings-on at that little store were a complex blur of sights (I have never liked the look of raw liver), sounds (my boss had been a Marine Corps drill instructor and knew how to express himself for effect), and smells (better left to the imagination).

But the longer I worked at Conrad Market, the more all those disconnected events began to form themselves into understandable chunks. In the morning you unlocked the front door, disarmed the alarm, and put out the "Yes! We're Open" sign. At the end of the day you reversed the process. In between there were a million jobs to be done—everything from stocking the shelves to helping customers find the ketchup. As I got to know my colleagues in other grocery stores, it turned out that those other markets were run pretty much the same way.

This is how people deal with the problems they confront, with the complexity that life forces on them. The first few times we see a problem we may improvise and invent a solution on the spot, but if that same problem keeps reappearing, we will come up with a standard operating procedure to cover it. Don't, as the old saying goes, keep reinventing the wheel.

The Gang of Four

Wheel reinvention is a constant problem for software engineers. It is not that we like doing things over and over. It is just that sometimes when you are designing systems it is hard to realize that the circular friction reduction device with a central axis that you have just built is, in fact, a wheel. Software design can be so numbingly complex that it is easy to miss the patterns of challenges and solutions that pop up repeatedly.

In 1995, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides set out to redirect all the effort going into building redundant software wheels into some thing more useful. That year, building on the work of Christopher Alexander, Kent Beck, and others, they published *Design Patterns: Elements of Reusable Object-Oriented Software*. The book was an instant hit, with the authors rapidly becoming famous (at least in software engineering circles) as the Gang of Four (GoF).

The GoF did two things for us. First, they introduced most of the software engi neering world to the idea of design patterns, where each pattern is a prepackaged solu tion to a common design problem. We should look around, they wrote, and identify common solutions to common problems. We should give each solution a name, and we should talk about

what that solution is good for, when to use it, and when to reach for something else. And we should write all of this information down, so that over time our palette of design solutions will grow.

Second, the GoF identified, named, and described 23 patterns. The original 23 solutions were the recurring patterns that the GoF saw as key to building clean, well designed object-oriented programs. In the years since *Design Patterns* was published, people have described patterns in everything from real-time micro-controllers to enter prise architectures. But the original 23 GoF patterns stuck to the middle ground of object-oriented design—bigger than a hash table, smaller than a database—and focused on some key questions: How do objects like the ones you tend to find in most systems relate to one another? How should they be coupled together? What should they know about each other? How can we swap out parts that are likely to change frequently?

Patterns for Patterns

In trying to answer these questions, the GoF opened their book with a discussion of some general principles, a set of meta-design patterns. For me, these ideas boil down to four points:

- Separate out the things that change from those that stay

the same. • Program to an interface, not an

implementation.

- Prefer composition over inheritance.
- Delegate, delegate, delegate.

To these, I would like to add one idea that is not really mentioned in *Design Patterns,* but that guides much of my own approach to building programs:

- You ain’t gonna need it.

In the following sections, we will look at each of these principles in turn, to see what they can tell us about building software.

Separate Out the Things That Change from Those That Stay the Same

Software engineering would be a lot easier if only things would stay the

same. We could build our classes serene in the knowledge that, once finished, they would con tinue to do exactly what we built them to do. Of course, things never stay the same, not in the wider world and certainly not in software engineering. Changes in com puting hardware, operating systems, and compilers, combined with ongoing bug fixes and ever-migrating requirements, all take their toll.

A key goal of software engineering is to build systems that allow us to contain the damage. In an ideal system, all changes are local: You should never have to comb through *all* of the code because A changed, which required you to change B, which triggered a change in C, which rippled all the way down to Z. So how do you achieve— or at least get closer to—that ideal system, the one where all changes are local?

You get there by separating the things that are likely to change from the things that are likely to stay the same. If you can identify which aspects of your system design are likely to change, you can isolate those bits from the more stable parts. When requirements change or a bug fix comes along, you will still have to modify your code, but perhaps, just perhaps, the changes can be confined to those walled-off, change prone areas and the rest of your code can live on in stable peace.

But how do you effect this quarantine? How do you keep the changing parts from infecting the stable parts?

Program to an Interface, Not an Implementation

A good start is to write code that is less tightly coupled to itself in the first place. If our classes are to do anything significant, they need to know about each other. But *what*

exactly do they need to know? The following Ruby fragment[1] creates a new instance of the `Car` class and calls the `drive` method on that instance:

```
my_car = Car.new
my_car.drive(200)
```

Clearly, this bit of code is married to the `Car` class. It will work as long as the requirement is that we need to deal with exactly one type of vehicle: a car. Of course, if the requirement changes and the code needs to deal with a second type of transport (such as an airplane), we suddenly have a problem. With only two flavors of vehicle to deal with, we might be able to get away with the following monstrosity:

```
# Deal with cars and planes

if is_car
   my_car = Car.new
   my_car.drive(200)
else
   my_plane = AirPlane.new
   my_plane.fly(200)
end
```

Not only is this code messy, but it is now also coupled to both cars and airplanes. This fix may just about hold things together . . . until a boat comes along. Or a train. Or a bike. A better solution, of course, is to return to Object-Oriented Programming 101 and apply a liberal dose of polymorphism. If cars and planes and boats all implement a common interface, we could improve things by doing something like the following:

```
my_vehicle = get_vehicle
my_vehicle.travel(200)
```

In addition to being good, straightforward object-oriented programming, this second example illustrates the principle of **programming to an interface.** The original

1. If you are new to Ruby, don't fret: The code in this chapter is very, very basic, and the next chapter offers a reasonably full introduction to the language. So just sit back for the next few pages and try to hum along.

code worked with exactly one type of vehicle—a car—but the new and improved ver sion will work with any Vehicle.

Java and C# programmers sometimes take the advice to "program to an interface" literally, as the **interface** is an actual construct in those languages. They carefully abstract out all the important functionality into many separate interfaces, which their classes then implement. This is usually a good idea, but it is not really what the prin ciple of programming to an interface is suggesting. The idea here is to program to the most general type you can—not to call a car a car if you can get away with calling it a vehicle, regardless of whether Car and Vehicle are real classes or abstract interfaces. And if you can get away with calling your car something more general still, such as a movable object, so much the

better. As we shall see in the pages that follow, Ruby (a language that lacks interfaces in the built-in syntax sense)[2] actually encourages you to program to your interfaces in the sense of programming to the most general types.

By writing code that uses the most general type possible—for example, by treat ing all of our planes and trains and cars like vehicles whenever we can—we reduce the total amount of coupling in our code. Instead of having 42 classes that are all tied to cars and boats and airplanes, perhaps we end up with 40 classes that know only about vehicles. Chances are that the remaining two classes will still give us trouble if we have to add another kind of vehicle, but at least we have limited the damage. In fact, if we have to change only a couple of classes, we have succeeded in separating out the parts that need to change (the two classes) from the parts that stay the same (the other 40 classes). The cumulative effect of turning down the coupling volume is that our code tends to be less likely to shatter in a horrendous chain reaction in the face of change.

Even so, programming to an interface is not the only step that we can take to make our code more change resistant. There is also composition.

Prefer Composition over Inheritance

If your introduction to object-oriented programming was like mine, you spent 10 minutes on information hiding, 22 minutes on scope and visibility issues, and the rest of the semester talking about inheritance. Once you got past the basic ideas of objects, fields, and methods, inheritance was the interesting thing, the most object-oriented part of object-oriented programming. With inheritance you could get implementation

2. The Ruby language does support modules, which bear a “first cousin” relationship to Java inter faces. We will have a lot more to say about Ruby modules in Chapter 2.

 MovableObject

Vehicle

MotorBoat AirPlane

Car

Figure 1-1 Getting the maximum mileage out of inheritance

for free: Just subclass Widget and you magically can take advantage of all the good stuff in the Widget class.

Inheritance sometimes seems like the solution to every problem. Need to model a car? Just subclass Vehicle, which is a kind of MovableObject. Similarly, as shown in Figure 1-1, an AirPlane might branch off to the right under Vehicle while MotorBoat might go off to the left. At each level we have succeeded in taking advan tage of all the workings of the higher-level superclasses.

The trouble is that inheritance comes with some unhappy strings attached. When you create a subclass of an existing class, you are not really creating two sepa rate entities: Instead, you are making two classes that are bound together by a com mon implementation core. Inheritance, by its very nature, tends to marry the subclass to the superclass. Change the behavior of the superclass, and there is an excellent chance that you have also changed the behavior of the subclass. Further, subclasses have a unique view into the guts of the superclass. Any of the interior workings of the superclass that are not carefully hidden away are clearly visible to the subclasses. If our goal is to build systems that are not tightly coupled together, to build systems where a single change does not ripple through the code like a sonic boom, breaking the glassware as it goes, then probably we should not rely on inheritance as much as we do.

If inheritance has so many problems, what is the alternative? We can assemble the behaviors we need through **composition.** Instead of creating classes that inherit most of their talents from a superclass, we can assemble functionality from the bottom up. To do so, we equip our objects with references to other objects—namely, objects that supply the functionality that we need. Because the functionality is encapsulated in

these other objects, we can call on it from whichever class needs that functionality. In short, we try to avoid saying that an object *is a kind of* something and instead say that it *has* something.

Returning to our car example, assume we have a method that simulates taking a Sunday drive. A key part of taking that drive is to start and stop the engine:

```
class Vehicle
```

```
    # All sorts of vehicle-related code...

    def start_engine
      # Start the engine
    end

    def stop_engine
      # Stop the engine
    end
  end

  class Car < Vehicle
    def sunday_drive
      start_engine
      # Cruise out into the country and return
      stop_engine
    end
  end
```

The thinking behind this code goes something like this: Our car needs to start and stop its engine, but so will a lot of other vehicles, so let's abstract out the engine code and put it up in the common Vehicle base class (see Figure 1-2).

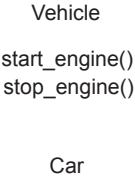


Figure 1-2 Abstracting out the engine code into the base class

That is great, but now all vehicles are required to have an engine. If we come across an engine-less vehicle (for example, a bicycle or a sailboat), we will need to perform some serious surgery on our classes. Further, unless we take extraordinary care in building the Vehicle class, the details of the engine are probably exposed to the Car class—after all, the engine is being managed by Vehicle and a Car is noth ing but a flavor of a Vehicle. This is hardly the stuff of separating out the change able from the static.

We can avoid all of these issues by putting the engine code into a

class all of its own—a completely stand-alone class, not a superclass of Car:

```
class Engine

  # All sorts of engine-related code...

  def start
    # Start the engine
  end

  def stop
    # Stop the engine
  end
end
```

Now, if we give each of our Car objects a reference to its own Engine, we are ready for a drive, courtesy of composition.

```
class Car
  def initialize
    @engine = Engine.new
  end

  def sunday_drive
    @engine.start
    # Cruise out into the country and return...
    @engine.stop
  end
end
```

Assembling functionality with composition (Figure 1-3) offers a whole trunk load of advantages: The engine code is factored out into its own class, ready for reuse

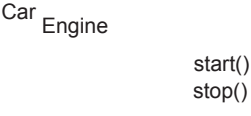


Figure 1-3 Assembling a car with composition

(via composition, of course!). As a bonus, by untangling the engine-related code from Vehicle, we have simplified the Vehicle class.

We have also increased encapsulation. Separating out the engine-related code from Vehicle ensures that a firm wall of interface exists between the car and its engine. In the original, inheritance-based version, all of the details of the engine implementation were exposed to all of the methods in Vehicle. In the new version, the only way a car can do anything to its engine is by working through the public— and presumably well-thought-out—interface of the Engine class.

We have also opened up the possibility of other kinds of engines. The Engine class itself could actually be an abstract type and we might have a variety of engines, all available for use by our car, as shown in Figure 1-4.

On top of that, our car is not stuck with one engine implementation for its whole life. We can now swap our engines at runtime:

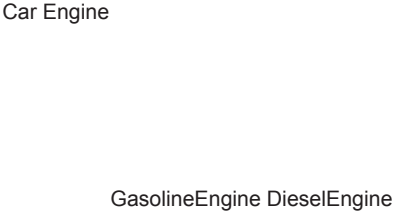


Figure 1-4 A car can now have different kinds of engines


```
class Car
  def initialize
    @engine = GasolineEngine.new
  end

  def sunday_drive
    @engine.start
    # Cruise out into the country and return...
    @engine.stop
  end

  def switch_to_diesel
    @engine = DieselEngine.new
```

```
    end
end
```

Delegate, Delegate, Delegate

There is a subtle functional difference between our current Car class (the one with the separate Engine object) and the original inheritance-based implementation. The original Car class exposed the start_engine and stop_engine methods to the world at large. Of course, we can do the same thing in our latest implementation of Car by simply foisting off the work onto the Engine object:

```
class Car
    def initialize
        @engine = GasolineEngine.new
    end

    def sunday_drive
        @engine.start
        # Cruise out into the country and return...
        @engine.stop
    end

    def switch_to_diesel
        @engine = DieselEngine.new
    end

    def start_engine
        @engine.start
    end
```



```
    def stop_engine
        @engine.stop
    end
end
```

This simple “pass the buck” technique goes by the somewhat pretentious name of **delegation.** Someone calls the start_engine method on our Car. The car object says, “Not my department,” and hands the whole problem off to the engine.

The combination of composition and delegation is a powerful and flexible alter native to inheritance. We get most of the benefits of inheritance, much more flexibil ity, and none of the unpleasant side effects. Of course, nothing comes for free. Delegation requires an extra method call, as the delegating object passes the buck along. This extra method

call will have some performance cost—but in most cases, it will be very minor.

Another cost of delegation is the boilerplate code you need to write—all of those boring delegating methods such as start_engine and stop_engine that do noth ing except pass the buck on to the object that really knows what to do. Fortunately, this is a book about design patterns in Ruby and, as we shall see in Chapters 10 and 11, in Ruby we don't have to write all of those dull methods.

You Ain't Gonna Need It

So much for the principles originally cited by the GoF in 1995. To this formidable set, I would like to add one more principle that I think is critical to building, and actually finishing, real systems. This design principle comes out of the Extreme Programming world and is elegantly summed up by the phrase **You Ain't Gonna Need It** (**YAGNI** for short). The YAGNI principle says simply that you should not implement features, or design in flexibility, that you don't need *right now*. Why? Because chances are, *you ain't gonna need it later,* either.

Look at it this way: A well-designed system is one that will flex gracefully in the face of bug fixes, changing requirements, the ongoing march of technology, and inevitable redesigns. The YAGNI principle says that you should focus on the things that you need right now, building in only the flexibility that you know you need. If you are not sure that you need it right now, postpone implementing the functionality until you really do need it. If you do not need it now, do not implement it now; instead, spend your time and energy implementing the things that you definitely need right now.

At the heart of the YAGNI idea is the simple realization that we tend to be wrong when we try to anticipate exactly what we will need in the future. When you put in

some feature or add some new bit of flexibility to your system before you really need it, you are making a two-pronged bet.

First, you are betting that you will eventually need this new feature. If you make your persistence layer database independent today, you are betting that someday you will need to port to a new database. If you internationalize your GUI now, you are betting that someday you will have users in some foreign land. But as Yogi Berra is supposed to have said, predictions are hard, especially about the future. If it turns out that you never need to work with another database, or if your application never

makes it out of its homeland, then you have done all of that up-front work and lived with all of the additional complexity for naught.

The second prong of the bet that you make when you build something before you actually need it is perhaps even more risky. When you implement some new feature or add some new flexibility before its time, you are betting that you can get it right, that you know how to correctly solve a problem that you haven't really encountered yet. You are betting that the database independence layer you so lovingly installed last year will be able to handle the database the system actually does move to: "What! Marketing wants us to support xyzDB? I've never heard of it!" You are betting that your GUI internation alization will be able to deal with the specific languages that you need to support: "Gee, I didn't realize that we would need to support a language that reads from right to left . . ."

Look at it this way: Barring a sharp blow to the head, as you stand here today you are as dumb as you ever will be. We are all learning, getting smarter every day. This is especially true in software projects: You can be sure that you will have a better grasp of the requirements, technology, and design of any software system that you work on at the end of the project than at the beginning. Whenever you put in a feature before you really need it, you are guilty of programming while stupid; if you wait until you really need the thing, you are likely to have a better understanding of what you need to do and how you should go about doing it.

Design patterns are all about making your systems more flexible, more able to roll smoothly with change. But the use of design patterns has somehow become associated with a particularly virulent strain of over-engineering, with code that tries to be infi nitely flexible at the cost of being understandable, and maybe even at the cost of just plain working. The proper use of design patterns is the art of making your system just flexible enough to deal with the problems you have today, but no more. Patterns are useful techniques, rather than ends in themselves. They can help you construct a working system, but your system will not work better because you used all 23 of the GoF design patterns in every possible combination. Your code will work better only if it focuses on the job it needs to do right now.

Fourteen Out of Twenty-Three

The patterns presented in *Design Patterns* are tools for building software. Like the real world tools that you can buy in a hardware store, they are not all equally useful in all sit uations. Some are like your trusty hammer,

absolutely required for every job. Others are more like that laser level that I got for my last birthday—great when you need it, which is not really all that often. In this book we will look at 14 of the original 23 GoF pat terns. In picking which patterns to discuss, I have tried to concentrate on the most widely used and useful. For example, I find it hard to imagine coding without iterators (Chapter 7), so that one is definitely in. I have also leaned toward the patterns that morph in the translation to Ruby. Here again, the iterator is a good example: I can't live without iterators, but iterators in Ruby are not quite the same as iterators in Java or C++.

To give you a preview of what lies in store for you, here is a quick overview of the GoF patterns covered in this book:

- Every pattern has a problem that it is trying to solve. For example, perhaps your code always wants to do exactly the same thing, except at step 44. Sometimes step 44 wants to do this, and sometimes it wants to do that. Perhaps you need a **Template Method.**
- Maybe it is not step 44 that wants to vary but the whole algorithm. You have a well-defined job, something that needs to get done, but there are a lot of ways to do it. You might need to remove the outer covering from a feline creature, and there is more than one technique you might employ. You might want to wrap those techniques—those algorithms—in a **Strategy** object.
- What if you have a class A, which needs to know what is happening over there in class B? But you don't want to couple the two classes together because you never know when class C (or even class D!) might come along. You might want to con sider using the **Observer** pattern.
- Sometimes you need to treat a collection of objects just like a single object—I can delete or copy or move an individual file, but I can also delete or copy or move a whole directory of files. If you need to build a collection of objects that looks just like one of the individual objects, you probably need the **Composite** pattern.
- Now suppose you are writing some code that hides a collection of objects, but you don't want the collection hidden *too* well: You would like your client to be able to access the objects in your collection in sequence without knowing how or where you have stored those objects. You definitely need the **Iterator** pattern.

- Sometimes we need to wrap instructions in a kind of a postcard: *Dear database, when you get this, delete row number 7843.* Postcards are

hard to come by in code, but the **Command** pattern is tailor made for this situation.

- What do you do when an object does what you need it to do, but its interface is wrong? Your interface mismatch might be very deep and complex, or it might be as simple as needing an object that can write but having an object that calls it save. The GoF would recommend an **Adapter** to you.
- Maybe you have the right object, but it is over there, someplace else on the net work, and you don't want the client code to care about its location. Or perhaps you want to delay creating your object as long as possible, or control access to it. In this circumstance, you may need a **Proxy.**
- Sometimes you need to add responsibilities to your objects on the fly, at runtime. If you have a need for objects that implement some core capabilities but must some times take on additional responsibilities, perhaps you need the **Decorator** pattern.
- Perhaps you have an instance of a class, and it needs to be the only instance of that class—that is, the single instance that everybody uses. Sounds like you have a **Singleton.**
- Now suppose you are writing a base class, one that is meant to be extended. As you are happily coding away at your base class, you realize that it needs to pro duce a new object and only the subclass will know exactly which kind of object to produce. You may need a **Factory Method** here.
- How do you create families of compatible objects? Suppose you have a system that models various types of cars, but not all engines are compatible with all fuel or cooling systems. How do you ensure that you don't end up with the automotive equivalent of Frankenstein's monster? You might build a class devoted to creating all of those objects and call it an **Abstract Factory.**
- Perhaps you are building an object so complex that its construction requires a sig nificant bit of code. Even worse, the process of construction needs to vary accord ing to the circumstances. Perhaps you need the **Builder** pattern?
- Ever have the feeling that you are using the wrong programming language to solve your problem? As crazy as it sounds, perhaps you should stop trying to solve your problem directly and build an **Interpreter** for a language that solves your prob lem more easily.

Patterns in Ruby?

So much for the theory of patterns.[3] But why Ruby? Isn't Ruby just some scripting language, only suitable for system administration tasks and building Web GUIs? In a word, no.

Ruby is an elegant, general-purpose, object-oriented programming language. It is not suited for every situation—for example, if you need very high performance, perhaps you should look elsewhere (at least for the moment). But for many programming jobs Ruby is more than suitable. Ruby code is terse but expressive. Ruby brings with it a rich and sophisticated model of programming.

As we shall see in the coming chapters, Ruby has its own way of doing things, a way that changes how we approach many programming problems, including the problems addressed by the classic GoF design patterns. Given that, it would be sur prising if the combination of Ruby and the classic design patterns did not lead to something a little bit different, a new twist on the traditional. Sometimes Ruby is dif ferent enough that its use permits completely new solutions to programming prob lems. In fact, three new patterns have come into focus with the recent popularity of Ruby. So, the catalog of patterns in this book will close out with the following:

- The **Internal Domain-Specific Language (DSL),** a very dynamic twist on build ing specialized little languages
- **Meta-programming,** a technique for creating just the classes and objects that you need, dynamically at runtime
- **Convention Not Configuration,** a cure for the (mostly XML)

  configuration blues Let's get started . . .

3. Of course, I have just barely made the faintest of scratches on the surface of this huge and inter esting topic. Take a look at Appendix B, Digging Deeper, to learn more.

This page intentionally left blank

CHAPTER 2
Getting Started with Ruby

I first found Ruby because of my eight-year-old son and his love of a certain electrically charged, but very sweet yellow mouse. Back in 2002, my son was spending his free time playing a certain computer game that involved finding and taming various magical creatures, including the energetic rodent.[1] In my mind's eye, I see the glowing light bulb suddenly appear over his head. "My dad," I imagine him thinking, "is a programmer. This thing that I'm playing, this thing with the magical islands and wonderful crea tures, is a program. My dad makes programs. My dad can teach me to make a game!"

Well, maybe. After a blast of begging and pleading that only the parents of young children can truly comprehend, I set out to teach my son to program. The first thing I needed, I thought, was a simple, clear, and easy-to-understand programming language. So I went looking—and I found Ruby. My son, as kids do, rapidly moved on to other things. But in Ruby I had found a keeper. Yes, it was clean and clear and simple—a good language for learning. But the professional software engineer in me, the guy who has built systems in everything from assembly language to Java, saw something else: Ruby is concise, sophisticated, and wickedly powerful.

But Ruby is also solidly mainstream. The basic moving parts of the language are the old gears and pulleys familiar to all programmers. Ruby has all of the data types with which you are familiar: strings, integers, and floating-point numbers, along with arrays and our old friends, `true` and `false`. The basics of Ruby are familiar and commonplace. It is the way they are assembled and the higher-level stuff that makes the language such an unexpected joy.

1. Thankfully, the Pokemon craze has abated a bit since then.

If you already know the basics of Ruby, if you know how to pull the

third char acter out of a string and how to raise 2 to the 437th power, and if you have written a class or two, then you can probably dive right into Chapter 3. This chapter will still be here if you get stuck.

If you are very new to Ruby, then this chapter is for you. In this chapter I will take you through the basics of the language as quickly as possible. After all, Alan Turing was right: Past a certain level of complexity, all programming languages are equivalent. If you already know another mainstream language, then the basics of Ruby will pres ent no problem to you; you will just be relearning all of the things that you already know. My hope is that by the end of this chapter you will know just enough of the language to be dangerous.

Interactive Ruby

The easiest way to run Ruby [2] is to use the interactive Ruby shell, irb. Once you start up irb, you can type in Ruby code interactively and see the results immediately. To run irb, you simply enter irb at the command prompt. The following example starts up irb and then uses Ruby to add 2 plus 2:

```
$ irb
irb(main):001:0> 2 + 2
=> 4
irb(main):002:0>
```

The interactive Ruby shell is great for trying out things on a small scale. There is nothing like immediate feedback for exploring a new programming language.

Saying Hello World

Now that you have Ruby safely up and running, the next step is obvious: You need to write the familiar hello world program. Here it is in Ruby:

```
#
# The traditional first program in any language
#
puts('hello world')
```

2. See Appendix A if you need guidance in installing Ruby on your system.

You could simply start up irb and type in this code interactively. Alternatively, you could use a text editor and type the hello world program into a text file with a name such as hello.rb. You can then use the Ruby interpreter, appropriately called ruby, to run your program:

```
$ ruby hello.rb
```

Either way, the result is the same:

```
hello world
```

You can learn an awful lot about a programming language from the hello world program. For example, it looks like the puts method prints things out, and so it does. You can also see that Ruby comments start with the # character and continue until the end of the line. Comments may fill the entire line, as in the earlier example, or you can tack on a comment after some code:

```
puts('hello world') # Say hello
```

Another thing to notice about our Ruby hello world program is the absence of semicolons to end the statements. In Ruby, a program statement generally ends with the end of a line. Ruby programmers tend to use semicolons only to separate multi ple statements on the same line on those rare occasions when they decide to jam more than one statement on a line:

```
#
# A legal, but very atypical Ruby use of the semicolon
#
puts('hello world');
#
# A little more de rigueur, but still rare use of semicolons #
puts('hello '); puts('world')
```

The Ruby parser is actually quite smart and will look at the next line if your state ment is obviously unfinished. For example, the following code works just fine because the Ruby parser deduces from the dangling plus sign that the calculation continues on the second line:

```
x = 10 +
    20 + 30
```

statement onto the next line with a backslash:

```
x = 10 \
      + 10
```

There is a theme here: The Ruby design philosophy is to have the language help you whenever possible but to otherwise stay out of the way. In keeping with this phi losophy, Ruby lets you omit the parentheses around argument lists when the meaning is clear without them:

```
puts 'hello world'
```

For clarity, most of the examples in this book include the parentheses when call ing a method, unless there are no arguments, in which case the empty parentheses are omitted.

While our hello world program wrapped its string with single quotes, you can also use double quotes:

```
puts("hello world")
```

Single or double quotes will result in exactly the same kind of string object, albeit with a twist. Single-quoted strings are pretty much a case of “what you see is what you get”: Ruby does very little interpretation on single-quoted strings. Not so with double quoted strings. These strings are preprocessed in some familiar ways: `\n` becomes a single linefeed (also known as a newline) character, while `\t` becomes a tab. So while the string `'abc\n'` is five characters long (the last two characters being a backslash and the letter “n”), the string `"abc\n"` is only four characters long (the last character being a linefeed character).

Finally, you may have noticed that no `\n` appeared at the end of the `'hello world'` string that we fed to `puts`. Nevertheless, `puts` printed our message with a newline. It turns out that `puts` is actually fairly clever and will stick a newline char acter on the end of any output that lacks one. This behavior is not necessarily desir able for precision formatting, but is wonderful for the kind of examples you will find throughout this book.

Variables

Ordinary Ruby variable names—we will meet several extraordinary variables in a bit— start with a lowercase letter or an underscore.[3] This first character can be followed by uppercase or lowercase letters, more underscores, and numbers. Variable names are case sensitive, and other than your patience there is no limit on the length of a Ruby variable name. All of the following are valid Ruby variable names:

- max_length
- maxLength
- numberPages
- numberpages
- a_very_long_variable_name
- _flag
- column77Row88
- ___

The first two variable names in this list, max_length and maxLength, bring up an important point: While camelCase variable names are perfectly legal in Ruby, Ruby programmers tend not to use them. Instead, the well-bred Ruby programmer uses words_separated_by_underscores. Also, because variable names are case sensitive, numberPages and numberpages are different variables. Finally, the last variable name listed above consists of three underscores, which is legal in the way that many very bad ideas are legal.[4]

Okay, let's put our strings and variables together:

```
first_name = 'russ'
last_name = 'olsen'
full_name = first_name + ' ' + last_name
```

3. In fact, for most parsing purposes in Ruby, an underscore actually counts as a lowercase letter. This, of course, raises the question of what an uppercase underscore would look like, and whether it would be called an overscore.

4. Another good reason to avoid using all-underscore variable names is that irb has already beat you to the punch. irb sets the variable _ (that is, one underscore) to the last expression it evaluated.

What we have here are three basic assignments: The variable

first_name is assigned the string 'russ', last_name is assigned a value of 'olsen', and full_name gets the concatenation of my first and last names separated by a space.

You may have noticed that none of the variables were declared in the previous example. There is nothing declaring that the variable first_name is, and always will be, a string. Ruby is a dynamically typed language, which means that variables have no fixed type. In Ruby, you simply pull a variable name out of thin air and assign a value to it. The variable will take on the type of the value it happens to hold. Not only that, but at different times in the same program a variable might hold radically dif ferent types. Early in a program, the value of pi might be the number 3.14159; later in the same program, the value might be a reference to a complex mathematical algo rithm; still later, it might be the string "apple". We will revisit dynamic typing a number of times in this book (starting with the very next chapter, if you can't wait), but for now just remember that variables take on the types of their values.

Aside from the garden-variety variables that we have examined so far, Ruby sup ports constants. A constant is similar to a variable, except that its name starts with an uppercase letter:

```
POUNDS_PER_KILOGRAM = 2.2
StopToken = 'end'
FACTS = 'Death and taxes'
```

The idea of a constant is, of course, that the value should not change. Ruby is not overly zealous about enforcing this behavior, however. You *can* change the value of a constant, but you will get a warning for your trouble:

```
StopToken = 'finish'
(irb):2: warning: already initialized constant StopToken
```

For the sake of sanity, you should probably avoid changing the values of constants.

Fixnums and Bignums

To no one's great surprise, Ruby supports arithmetic. You can add, subtract, multiply, and divide in Ruby with the usual results:

```
x = 3
y = 4
sum = x+y
product = x*y
```

Ruby enforces some standard rules about numbers. There are two basic flavors, integers and floating-point numbers. Integers, of course, lack a fractional part: 1, 3, 6, 23, –77, and 42 are all integers, whereas 7.5, 3.14159, and –60000.0 are all floating point numbers.

Integer division in Ruby comes as no surprise: Divide two integers and your answer will be another integer, with any decimal part of the quotient silently truncated (not rounded!):

```
6/3 # Is 2
7/3 # Is still 2
8/3 # 2 again
9/3 # Is 3 finally!
```

Reasonably sized integers—anything you can store in 31 bits—are of type `Fixnum`, while larger integers are of type `Bignum`; a `Bignum` can hold an arbitrarily gigantic number. Given that the two types convert back and forth more or less seam lessly, however, you can usually forget that there is any distinction between them:

```
2 # A Fixnum
437 # A Fixnum
2**437 # Very definitely a big Bignum
1234567890 # Another Bignum
1234567890/1234567890 # Divide 2 Bignums, and get 1, a Fixnum
```

Ruby also supports the familiar assignment shortcuts, which enable you to shorten expressions such as `a = a+1` to `a += 1`:

```
a = 4
a += 1 # a is now 5
a -= 2 # a is now 3
a *= 4 # a is now 12
a /= 2 # a is now 6
```

Sadly, there are no increment (`++`) and decrement (`--`) operators in Ruby.

Floats

If only the world were as exact as the integers. To deal with messy reality, Ruby also supports floating-point numbers or, in Ruby terminology, floats. A float is easy to spot—it is a number with a decimal point:

```
3.14159
-2.5
6.0
0.0000000111
```

You can add, subtract, and multiply floats with the results you would expect. Floats even obey the more familiar rules of grammar-school division:

```
2.5+3.5 # Is 6.0
0.5*10 # Is 5.0
8.0/3.0 # Is 2.66666666
```

There Are No Primitives Here

You do not have to take my word about the types of these different flavors of num bers. You can simply ask Ruby by using the `class` method:

```
7.class # Gives you the class Fixnum
888888888888.class # Gives you the class Bignum
3.14159.class # Gives you the class Float
```

The slightly strange-looking syntax in this code is actually a tip-off to something deep and important: In Ruby, everything—and I mean *everything*—is an object. When we say `7.class`, we are actually using the familiar object-oriented syntax to call the `class` method on an object, in this case the object representing the number seven. In fact, Ruby numbers actually have quite a wide selection of methods available:

```
3.7.round # Gives us 4.0
3.7.truncate # Gives us 3
-123.abs # Absolute value, 123
1.succ # Successor, or next number, 2
```


Unlike Java, C#, and many other widely used languages, Ruby does not have any primitive data types. It is objects all the way down. The fact that everything is an object in Ruby drives much of the elegance of the

language. For example, the univer sal object orientation of Ruby is the secret that explains how Fixnums and Bignums can be so effortlessly converted back and forth.

If you follow the class inheritance hierarchy of any Ruby object upward, from its class up through the parent class or superclass, and on to its super-duper-class, eventu ally you will reach the Object class. Because every Ruby object can trace its class ances try back to Object, all Ruby objects inherit a minimum set of methods, a sort of object-oriented survival kit. That was the source of the class method that we encoun tered earlier. We can also find out whether an object is an instance of a given class:

```
'hello'.instance_of?(String) # true
```

We can also see if it is nil:

```
'hello'.nil? # false
```

Perhaps the Object method that gets the most use is to_s, which returns a string representation of the object—a suitably brief name for the Ruby equivalent of the Java toString method:

```
44.to_s # Returns a two-character string '44'
'hello'.to_s # A fairly boring conversion, returns 'hello'
```

The total object orientation of Ruby also has some implications for variables. Because everything in Ruby is an object, it is not really correct to say that the expres sion x = 44 assigns the value 44 to the variable x. Instead, what is really happening is that x receives a reference to an object that happens to represent the number after 43.

But Sometimes There Is No Object

If everything is an object, what happens when you do not really have an object? For exactly those occasions, Ruby supplies us with a special object that represents the idea of not having an object, of being utterly object-less, sans object. This special value is nil.

In the last section, we said that everything in Ruby is an object, and so it is: nil is every bit as much a Ruby object as "hello world" or 43. For

example, you can get the class of nil:

```
puts(nil.class)
```

That turns out to be something very predictable:

```
NilClass
```

Sadly, nil is destined to live out its life alone: There is only one instance of NilClass (called nil) and you cannot make any new instances of NilClass.

Truth, Lies, and nil

Ruby supports the usual set of Boolean operators. For example, we can find out if two expressions are equal, if one expression is less than the other, or if one expression is greater than the other.

```
1 == 1 # true
1 == 2 # false
'russ' == 'smart' # sadly, false
(1 < 2) # true
(4 > 6) # nope

a = 1
b = 10000
(a > b) # no way
```

We also have less than or equal and its cousin, greater than or equal:

```
(4 >= 4) # yes!
(1 <= 2) # also true
```

All of these comparison operators evaluate to one of two objects—true or false. Like nil, the true and false objects are the only instances of their respective classes: true is the only instance of TrueClass and false is the sole instance of (you guessed it)

FalseClass. Oddly, both TrueClass and FalseClass are direct subclasses of Object. I keep expecting a BooleanClass to slip in somewhere, but alas, no.

Ruby also has an and operator—in fact, several of them. For example, you might say

```
(1 == 1) and (2 == 2) # true
(1 == 1) and (2 == 3) # false
```

You might also say

```
(1 == 1) && (2 == 2) # true
(1 == 1) && (2 == 3) # false
```

Both amount to the same thing. Essentially, `and` and `&&` are synonyms.[5] Matched up with `and` and `&&` is `or` and `||`, which do about what you would expect:[6]

```
(1 == 1) or (2 == 2) # yup
(2 == 1) || (7 > 10) # nope
(1 == 1) or (3 == 2) # yup
(2 == 1) || (3 == 2) # nope
```

Finally, Ruby has the usual `not` operator and its twin `!`:

```
not (1 == 2) # true
! (1 == 1) # false
not false # true
```

One thing to keep in mind is that Ruby can come up with a Boolean value for any expression. We can, therefore, mix strings, integers, and even dates into Boolean

5. Okay, not quite. The `&&` operator has a higher precedence—it is stickier in expressions—than `and`. The same is true of the `||` and `or` operators.
6. Ruby also has `&` and `|` operators—note that they are single characters. These guys are bitwise logi cal operators, useful in their own right but probably not what you would want to use in your average `if` statement.

expressions. The evaluation rule is very simple: `false` and `nil` evaluate to false; every thing else evaluates to true. So the following are perfectly legal expressions:

```
true and 'fred' # true, because 'fred' is neither nil nor false 'fred' && 44 # true, because
```

```
both 'fred' and 44 are true nil || false # false, because both nil and false are false
```

If you come from the world of C or C++, you will be shocked to learn that in Ruby, zero, being neither false nor nil, evaluates to true in a Boolean expression. Surprisingly, this expression

```
if 0
   puts('Zero is true!')
end
```

will print out

```
Zero is true!
```

Decisions, Decisions

That last example was a sneak preview of the if statement, which has the usual optional else:

```
age = 19

if (age >= 18)
   puts('You can vote!')
else
   puts('You are too young to vote.')
end
```

As you can see from the example, each if statement always has its own terminat ing end. If you have more than one condition, you can use an elsif:


```
if(weight < 1)
   puts('very light')
elsif(weight < 10)
   puts('a bit of a load')
elsif(weight < 100)
   puts('heavy')
else
   puts('way too heavy')
end
```

Note that the keyword is elsif—five letters, one word. It is not else if, elseif, and certainly not elif.

As usual, Ruby tries its best to make the code as concise as possible. Because the parentheses after the if and elsif really do not add much to the meaning, they are optional:

```
if weight < 1
   puts('very light')
elsif weight < 10
   puts('a bit of a load')
elsif weight < 100
   puts('heavy')
else
   puts('way too heavy')
end
```

There is also an idiom for those times when you need to decide whether you want to execute a single statement. Essentially, you can hang the if on the end of a statement:

```
puts('way too heavy') if weight >= 100
```

There is also an unless statement, which reverses the sense of an if statement: The body of the statement executes only if the condition is false. As with the if state ment, you can have a long form of unless:

```
unless weight < 100
   puts('way too heavy')
end
```


A short form is also available:

```
puts('way too heavy') unless weight < 100
```

Loops

Ruby has two flavors of loops. First, we have the classic while loop, which, like the if statement, is always terminated with an end. Thus this loop

```
i = 0
while i < 4
   puts("i = #{i}")
```

```
  i = i + 1
end
```

will print out this:

```
i = 0
i = 1
i = 2
i = 3
```

The evil twin of the while loop is the until loop, which is more or less identi cal to the while loop except that it keeps looping until the condition becomes true. Thus we might have written the preceding example as follows:

```
i = 0
until i >= 4
  puts("i = #{i}")
  i = i + 1
end
```

Ruby also has a for loop, which you can use, among other things, to sequence through arrays:

```
array = ['first', 'second', 'third']
array.each do |x|
  puts(x)
end
```


Surprisingly, for loops are rare in real Ruby programs. A Ruby programmer is much more likely to write this equivalent code instead:

```
array.each do |x|
  puts(x)
end
```

We will have much more to say about this odd-looking loop thing in Chapter 7. For now, just think of the each syntax as another way to write a for loop. If you need to break out of a loop early, you can use a break statement:

```
names = ['george', 'mike', 'gary', 'diana']

names.each do |name|
    if name == 'gary'
        puts('Break!')
        break
    end
    puts(name)
end
```

Run this code and it will never print out gary:

```
george
mike
Break!
```

Finally, you can skip to the next iteration of a loop with the next statement:

```
names.each do |name|
    if name == 'gary'
        puts('Next!')
        next
    end
    puts(name)
end
```

This code will skip right over gary but keep going:

```
george
mike
Next!
diana
```

More about Strings

Since we are already using Ruby strings, let's get to know them a little better. As we have seen, we can build string literals with both single and double quotes:

```
first = 'Mary had'
second = " a little lamb"
```

We have also seen that the plus sign is the concatenation

operator, so that `poem = first + second`

will give us this:

```
Mary had a little lamb
```

Strings have a whole range of methods associated with them. You can, for exam ple, get the length of a string:

```
puts(first.length) # Prints 8
```

You can also get an all-uppercase or all-lowercase version of a string:

```
puts(poem.upcase)
puts(poem.downcase)
```

This code will print out

```
MARY HAD A LITTLE LAMB
mary had a little lamb
```


In many ways, Ruby strings act like arrays: You can set individual characters in a string by indexing the string in a very array-like fashion. Thus, if you execute

```
poem[0] = 'G'
puts(poem)
```

you will get a very different sort of poem:

```
Gary had a little lamb
```

You can also get at individual characters in a string in the same way, albeit with a slightly annoying twist: Because Ruby lacks a special character type, when you pull an individual character out of a Ruby string

you get a number—namely, the integer char acter code. Consider this example:

```
second_char = poem[1] # second_char is now 97, the code for 'a'
```

Fortunately, you can also put individual characters back via the code, so perhaps there is not much harm done:

```
poem[0] = 67 # 67 is the code for 'C'
```

Now Cary is the former owner of a young sheep.

Double-quoted strings in Ruby have another feature, one that you are going to run into frequently in the examples in this book. While it is turning the familiar \n's into newlines and \t's into tabs, if the Ruby interpreter sees #{expression} inside a double-quoted string, it will substitute the value of the expression into the string. For example, if we set the variable n

```
n = 42
```

we can smoothly insert it into a string

```
puts("The value of n is #{n}.")
```

to get

```
The value of n is 42.
```


This feature (called string interpolation) is not just limited to one expression per string, nor is the expression limited to a simple variable name. For example, we can say

```
city = 'Washington'
temp_f = 84
        puts("The city is #{city} and the temp is #{5.0/9.0 * (temp_f-32)} C")
```

which will print

```
The city is Washington and the temp is 28.8888888888889 C
```

While traditional quotes are great for relatively short, single-line strings, they tend to be awkward for expressing longer, multiple-line strings. To ease this pain, Ruby has another way of expressing string literals:

```
a_multiline_string = %Q{
The city is #{city}.
The temp is #{5.0/9.0 * (temp_f-32)} C
}
```

In this example, everything between the %Q{ and the final } is a string literal. If you start your string literal with a %Q{ as we did above, Ruby will treat your string as double quoted and do all of the usual double-quoted interpretation on it. If you use %q{ (note the lowercase "q"), your text will receive the same minimal processing as a single-quoted string.[7]

Finally, if you are coming from the Java or C# world, there is a serious conceptual landmine waiting for you in Ruby. In C# and Java, strings are **immutable:** Once you create a string in either of these languages, that string can never be changed. Not so in Ruby. In Ruby, any string is liable to change just about any time. To illustrate, let us create two references to the same string:

```
name = 'russ'
first_name = name
```

7. Actually, you have a lot more options. You can, for example, use matched pairs of parentheses "()" or angle brackets "<>" instead of the braces that I show here to delimit your string. Thus %q<a string> is a fine string. Alternatively, you can use any other special character to start and end your string— for example, %Q-a string-.

If this were Java or C# code, I could use first_name essentially forever, serene in the knowledge that its value could never change out from under me. By contrast, if I change the contents of name:

```
name[0] = 'R'
```

I also change first_name, which is just a reference to the same string object. If we print out either variable

```
puts(name)
puts(first_name)
```

we will get the same, changed value:

```
Russ
Russ
```

Symbols

The merits of making strings immutable have been the subject of long debate. Strings were mutable in C and C++, were immutable in Java and C#, and went back to muta ble in Ruby. Certainly there are advantages to mutable strings, but making strings mutable does leave an obvious gap: What do we do when we need to represent some thing that is less about data and more like an internal identifier in our program?

Ruby has a special class of object just for this situation—namely, the symbol. A Ruby symbol is essentially an immutable identifier type thing. Symbols always start with a colon:

- :a_symbol
- :an_other_symbol
- :first_name

If you are not used to them, symbols may seem a bit strange at first. Just remem ber that symbols are more or less immutable strings and that Ruby programmers use them as identifiers.

Arrays

Creating arrays in Ruby is as easy as typing a pair of square braces or Array.new:

```
x = [] # An empty array
y = Array.new # Another one
a = [ 'neo', 'trinity', 'tank'] # A three-element array
```

Ruby arrays are indexed from zero:

```
a[0] # neo
a[2] # tank
```

You can get the number of elements in an array with the length or size method. Both do the same thing:

```
puts(a.length) # Is 3
puts(a.size) # Is also 3
```

Keep in mind that Ruby arrays do not have a fixed number of elements. You can extend arrays on the fly by simply assigning a new element beyond the end of the array:

```
a[3] = 'morpheus'
```

Our array now contains four elements.

If you add an element to an array far beyond the end, then Ruby will automati cally add the intervening elements and initialize them to nil. Thus, if we execute the code

```
a[6] = 'keymaker'
puts(a[4])
puts(a[5])
puts(a[6])
```

 we will get

```
nil
nil
keymaker
```

A convenient way to append a new element to the end of an array is with the << operator:

```
a << 'mouse'
```

Ruby also sticks to the spirit of dynamic typing with arrays. In Ruby, arrays are not limited to a single type of element. Instead, you can mix and match any kind of object in a single array:

```
mixed = ['alice', 44, 62.1234, nil, true, false]
```

Finally, because arrays are just regular objects,[8] they have a rich and varied set of methods. You can, for example, sort your array:

```
a = [ 77, 10, 120, 3]
a.sort # Returns [3, 10, 77, 120]
```

You can also reverse the elements in an array:

```
a = [1, 2, 3]
a.reverse # Returns [ 3, 2, 1]
```

Keep in mind that the sort and reverse methods leave the original array untouched: They actually return a *new* array that is sorted or reversed. If you want to sort or reverse the original array, you can use sort! and reverse!:

```
a = [ 77, 10, 120, 3]
a.sort! # a is now [3, 10, 77, 120]
a.reverse! # a is now [120, 77, 10, 3]
```

8. We know arrays are just objects in Ruby because (all together now!) *everything* in Ruby is an object.

This convention of method leaving the original object untouched while method! modifies the original object is not limited to arrays. It is applied frequently (but sadly not quite universally) throughout Ruby.

Hashes

A Ruby hash is a close cousin to the array—you can look at a hash as an array that will take anything for an index. Oh yes, and unlike arrays, hashes are unordered. You can create a hash with a pair of braces:

```
h = {}
h['first_name'] = 'Albert'
h['last_name'] = 'Einstein'

h['first_name'] # Is 'Albert'
h['last_name'] # Is Einstein
```

Hashes also come complete with a shortcut initialization syntax. We could define the same hash with

h = {'first_name' => 'Albert', 'last_name' => 'Einstein'} Symbols make good hash keys, so our example might be improved with h =

{:first_name => 'Albert', :last_name => 'Einstein'}

Regular Expressions

The final built-in Ruby type that we will examine is the regular expression. A regular expression in Ruby sits between a pair of forward slashes:

```
/old/
/Russ|Russell/
/.*/
```

While you can do incredibly complex things with regular expressions, the basic ideas underlying them are really very simple and a little knowledge will take you a

long way.[9] Briefly, a regular expression is a pattern that either does or does not match any given string. For example, the first of the three regular expressions above will match only the string 'old', while the second will match two variations of my first name. The third expression will match anything.

In Ruby, you can use the =~ operator to test whether a given regular expression matches a particular string. The =~ operator will return either nil (if the expression does not match the string) or the index of the first matching character (if the pattern does match):

```
/old/ =~ 'this old house' # 5 - the index of 'old'
/Russ|Russell/ =~ 'Fred' # nil – Fred is not Russ nor Russell /.*/ =~ 'any old string' # 0
- the RE will match anything
```

There is also a !~ operator for testing whether a regular expression does *not* match a given string.

A Class of Your Own

Ruby would not be much of an object-oriented language if you could not create classes of your own:

```
class BankAccount
  def initialize( account_owner )
    @owner = account_owner
    @balance = 0
  end

  def deposit( amount )
    @balance = @balance + amount
  end

  def withdraw( amount )
    @balance = @balance - amount
  end
end
```

9. If you are one of those people who have avoided learning regular expressions, let me recommend that you take some time to explore this wonderfully useful tool. You could start with some of the books listed in Appendix B, Digging Deeper.

Clearly, the Ruby class definition syntax has the same unadorned brevity as the rest of the language. We start a class definition with the keyword `class` followed by the name of the class:

```
class BankAccount
```

Recall that in Ruby all constants start with an uppercase letter. In Ruby's world view, a class name is a constant. This makes sense because the name of a class always refers to the same thing—the class. Thus, in Ruby, all class names need to start with an uppercase letter, which is why our class is `BankAccount` with a capital "B". Also note that while the only hard requirement is that the name of a class begin with an uppercase letter, Ruby programmers typically use camel case for class names.

The first method of our `BankAccount` class is the `initialize` method:

```
  def initialize( account_owner )
    @owner = account_owner
    @balance = 0
  end
```

The `initialize` method is both special and ordinary. It is ordinary in the way it is built—the line introducing the method consists of the keyword `def`

followed by the name of the method, followed by the argument list, if there is one. Our initialize method does have a single argument, account_owner.

Next, we have the body of the method—in this case, a couple of assignment statements. The first thing our initialize method does is to grab the value passed in through account_owner and assign it to the very strange-looking variable, @owner:

```
@owner = account_owner
```

Names that start with an @ denote **instance variables**—each instance of the BankAccount class will carry around its own copy of @owner. Likewise, each instance of BankAccount will carry around its copy of @balance, which we initialize to zero. As usual, there is no up-front declaration of @owner or @balance; we simply make up the names on the spot.

Although initialize is defined in the same, ordinary way as all other methods, it is special because of its name. Ruby uses the initialize method to set up new objects. When Ruby creates a new instance of a class, it calls the initialize method to set up the new object for use. If you do not define an initialize method on your

class, Ruby will do the typical object-oriented thing: It will look upward through the class hierarchy until either it finds an initialize method or it reaches Object. Given that the Object class defines an initialize method (which does nothing), the search is guaranteed to quietly end there. Essentially, initialize is the Ruby version of a constructor.

To actually construct a new instance of our class, we call the new method on the class. The new method takes the same parameters as the initialize method:

```
my_account = BankAccount.new('Russ')
```

This statement will allocate a new BankAccount object, call its initialize method with the arguments passed into new, and assign the newly initialized BankAccount instance to my_account.

Our BankAccount class has two other methods, deposit and withdraw, which grow and shrink the size of our account, respectively. But how do we get at our account balance?

Getting at the Instance Variables

While our BankAccount class seems like it is almost ready for use, there is

one prob lem: In Ruby, an object's instance variables cannot be accessed from outside the object. If we made a new BankAccount object and tried to get at @balance, we are in for an unpleasant shock. Running this code

```
my_account = BankAccount.new('russ')
puts(my_account.balance)
```

produces the following error:

```
account.rb:8: undefined method 'balance' ... (NoMethodError)
```

Nor does my_account.@balance, with the at sign, work. No, the instance vari ables on a Ruby object are just not visible outside the object. What is a coder to do? We might simply define an accessor method:

```
def balance
  @balance
end
```


One thing to note about the balance method is that it lacks a return statement. In the absence of an explicit return statement, a method will return the value of the last expression computed, which in this case is simply @balance. We can now get at our balance:

```
puts(my_account.balance)
```

The fact that Ruby allows us to omit the parentheses for an empty argument list gives us the satisfying feeling that we are accessing a value instead of calling a method. We might also like to be able to set the account balance directly. The obvious thing to do is to add a setter method to BankAccount:

```
def set_balance(new_balance)
  @balance = new_balance
end
```

Code with a reference to a BankAccount instance could then set the account balance: my_account.set_balance(100)

One problem with the set_balance method is that it is fairly ugly. It would be much clearer if you could just write

```
my_account.balance = 100
```

Fortunately, you can. When Ruby sees an assignment statement like this one, it will translate it into a plain old method call. The method name will be variable name, followed by an equals sign. The method will have one parameter, the value of the right-hand side of the assignment statement. Thus the assignment above is translated into the following method call:

```
my_account.balance=(100)
```

Take a close look at the name of that method. No, that is not some special syn tax; the method name really does end in an equals sign. To make this all work for our BankAccount object, we simply rename our setter method:

```
def balance=(new_balance)
  @balance = new_balance
end
```


We now have a class that looks good to the outside world: Code that uses BankAccount can set and get the balance with abandon, without caring that it is really calling the balance and balance= methods. Sadly, our class is a bit verbose on the inside: We seem doomed to have all of these boring name and name= methods littered throughout our class definition. Unsurprisingly, Ruby comes to our rescue yet again.

It turns out that getter and setter methods are so common that Ruby supplies us with a great shortcut to create them. Instead of going through all of the def name . . . motions, we can simply add the following line to our class:

```
attr_accessor :balance
```

This statement will create a method called balance whose body does nothing more than return the value of @balance. It will also create the balance= (new_value) setter method. We can even create multiple accessor methods in a single statement:

```
attr_accessor :balance, :grace, :agility
```

The preceding code adds no less than six new methods for the enclosing class: get ter and setter methods for each of the three named instance variables.[10] Instant acces sors, no waiting.

You also have help if you want the outside world to be able to read your instance variables but not write them. Just use attr_reader instead of attr_accessor:

```
attr_reader :name
```

Now your class has a getter method for name, but no setter method. Similarly, attr_writer will create only the setter method, name=(new_value).

10. There is a subtle twist of terminology going on here: The things with the at signs on the inside of the class are instance variables. When you create the getter and setter methods and expose them to the outside world, they become **attributes** of the object—hence *attr*_reader and *attr*_writer. In practice, the finer points of the terminology seem to be honored more in the breach than the observance and no one is confused.

An Object Asks: Who Am I?

Sometimes a method needs a reference to the current object, the instance to which the method is attached. For that purpose we can use self, which is always a reference to the current object:

```
class SelfCentered
  def talk_about_me
    puts("Hello I am #{self}")
  end
end

conceited = SelfCentered.new
conceited.talk_about_me
```

If you run this code, you will get something like this:

```
Hello I am #<SelfCentered:0x40228348>
```

Of course, your instance of SelfCentered is unlikely to reside at the same hex address as mine, so your output may look a little different.

Inheritance, Subclasses, and Superclasses

Ruby supports single inheritance—all the classes that you create have exactly one par ent or superclass. If you do not specify a superclass, your

class automatically becomes a subclass of Object. If you want your superclass to be something other than Object, you can specify the superclass right after the class name:

```
class InterestBearingAccount < BankAccount
  def initialize(owner, rate)
    @owner = owner
    @balance = 0
    @rate = rate
  end

  def deposit_interest
    @balance += @rate * @balance
  end
end
```


Take a good look at the InterestBearingAccount initialize method. Like the initialize method of BankAccount, the InterestBearingAccount ini tialize method sets @owner and @balance along with the new @rate instance vari able. The key point is that the @owner and @balance instance variables in InterestBearingAccount are the same as the ones in the BankAccount class. In Ruby, an object instance has only one set of instance variables, and those variables are visible all the way up and down the inheritance tree. If we went BankAccount mad and built a subclass of BankAccount and a sub-subclass of that, and so on for 40 classes and 40 subclasses, there would still be only one @owner instance variable per instance.

One unfortunate aspect of our InterestBearingAccount class is that the InterestBearingAccount initialize method sets the @owner and @balance fields, essentially duplicating the contents of the initialize method in BankAccount. We can avoid this messy code duplication by calling the Account initialize method from the InterestBearingAccount initialize method:

```
def initialize(owner, rate)
  super(owner)
  @rate = rate
end
```

Our new initialize method replaces the duplicate code with a call to super. When a method calls super, it is saying, “Find the method with the same name as me in my superclass, and call that.” Thus the effect of the call to super in the initial ize method is to call the initialize method in the BankAccount class. If there is no method of the same name in the superclass, Ruby will continue looking upward through the inheritance tree

until it finds a method or runs out of classes, in which case you will get an error.

Unlike many object-oriented languages, Ruby does not automatically ensure that initialize is called for all your superclasses. In this sense, Ruby treats initialize like an ordinary method. If the InterestBearingAccount initialize method did not make the call to super, the BankAccount rendition of initialize would never be called on InterestBearingAccounts.

Argument Options

So far, the methods with which we have adorned our classes have sported pretty bor ing lists of arguments. It turns out that Ruby actually gives us a fair number of options when it comes to method arguments. We can, for example, specify default values for our arguments:


```
def create_car( model, convertible=false)
   # ...
end
```

You can call create_car with one argument—in which case convertible defaults to false—or two arguments. Thus all of the following are valid calls to create_car:

```
create_car('sedan')
create_car('sports car', true)
create_car('minivan', false)
```

If you do write a method with default values, all of the arguments with default values must come at the end of the argument list.

While default values give you a lot of method-defining flexibility, sometimes even more freedom is handy. For those occasions you can create methods with an arbitrary number of arguments:

```
def add_students(*names)
   for student in names
      puts("adding student #{student}")
   end
end

add_students( "Fred Smith", "Bob Tanner" )
```

Run the code above and you will see

```
adding student Fred Smith
adding student Bob Tanner
```

The add_students method works because all of the arguments are rolled up in the names array—that's what the asterisk indicates. You can even mix and match regular arguments with the variable arguments array, as long as the array appears at the end of the argument list:


```
def describe_hero(name, *super_powers)
   puts("Name: #{name}")
   for power in super_powers
      puts("Super power: #{power}")
   end
end
```

The preceding method requires at least one argument but will take as many addi tional arguments as you care to give it. Thus all of the following are valid calls to describe_hero:

```
describe_hero("Batman")
describe_hero("Flash", "speed")
describe_hero("Superman", "can fly", "x-ray vision", "invulnerable")
```

Modules

Along with classes, Ruby features a second code-encapsulating entity called a module. Like a class, a module is a package of methods and constants. Unlike a class, however, you can never create an instance of a module. Instead, you include a module in a class, which will pull in all of the module's methods and constants and make them available to instances of the class. If you are a Java programmer, you might think of modules as being a bit like interfaces that carry a chunk of implementation code.

A module definition bears an uncanny resemblance to a class definition, as we can see from this simple, one-method module:

```
module HelloModule
```

```
    def say_hello
      puts('Hello out there.')
    end
end
```

Once we have defined our little module, we can pull it into any of our classes with the include statement:[11]

11. There is another way to use modules, which we will see in Chapter 12: You can just call methods directly out of the module without including the module in any class.


```
class TryIt
    include HelloModule
end
```

The effect of the include statement is to make all of the methods in the module available to instances of the class:

```
tryit = TryIt.new
tryit.say_hello
```

The accessibility also works in the other direction: Once a module is included in a class, the module methods have access to all of the methods and instance variables of the class. For example, the following module contains a method that prints various bits of information about the object in which it finds itself—values that it gets by call ing the name, title, and department methods supplied by its host class:

```
module Chatty
    def say_hi
        puts("Hello, my name is #{name}")
        puts("My job title is #{title}")
        puts("I work in the #{department} department")
    end
end

class Employee
    include Chatty

    def name
        'Fred'
    end
```

```
    def title
      'Janitor'
    end

    def department
      'Maintenance'
    end
  end
```

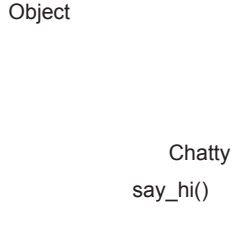
 Running this code produces

```
Hello, my name is Fred
My job title is Janitor
I work in the Maintenance department
```

When you include a module in your class, the module becomes a sort of special, secret superclass of your class (see Figure 2-1). But while a class can have only one superclass, it can include as many modules as it likes.

When someone calls a method on an instance of your class, Ruby will first deter mine whether that method is defined directly in your class. If it is, then that method will be called. For example, if you call the `name` method on an `Employee` instance, Ruby will look first in the `Employee` class, see that a `name` method is available right there, and call it. If there is no such method defined directly by the class, Ruby will next look through all the modules included by the class. For example, if you call the `say_hi` methods, Ruby—after failing to find it in the `Employee` class itself—will look in the modules included by `Employee`. If the class includes more than one mod ule, Ruby will search the modules from the last one included back to the first. But our `Employee` class includes only one module; right there in the `Chatty` module Ruby will find and call the `say_hi` method. If Ruby had not found the method in the `Employee` class or in any of its modules, it would have continued the search on to `Employee` superclass—and its modules.

Modules, when used in the way described here, are known as mixins—because they live to be *mixed in* (that is, to add their methods) to classes. Conceptually, mixin



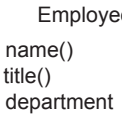


Figure 2-1 A module mixed into a class