UNIT-1

What is scripting Language?

A Scripting language is a programming language that employs a high-level construct to interpret and execute one command at a time.

Scripting Languages are often used for short scripts over full Computer programs.

Examples: Javascript, Python and Ruby are all Examples of Scripting Languages.

What is the difference between script and program?

A scripting Language does not include the Compilation step - it is interpreted instead.

Characteristics of Scripting Languages?

A scripting Language is a programming Language that executes tasks with in a special run-time environment by an interpreter instead of compiler. They are usually short, fast and interpreted from source code or byte code.

a) Flexible dynamic typing.
b) Easy access to other programs.
c) sophisticated pattern matching and string manipulation.
d) High Level data types.
e) lack of declarations; Simple Scoping rules.
f) economy of expression.
g) Integrated compile and run.
h) Low overheads and ease of use.
i) Enhanced functionality

Uses of Scripting Languages:

2 Scripting is primarily used to automate tasks for websites and web applications while using an existing program.

It is useful for extracting information from a dataset.

Computer programmers, Software developers, as well as front-end and back-end developers, use scripting skills in their careers.

Task automation locally or remotely. As with any other kind of programming language, scripting helps you automate repetitive tasks based on patterns.

Beginner-friendly short scripts

Create and manage dynamic content.

Back-end programming for Complex Systems.

1 Traditional Scripting:

The activities that comprise traditional scripting include:

a) System Administration
b) Controlling Applications Remotely
c) System and application extensions.
d) 'Experimental' programming
e) Building Command-line interfaces to applications based on C-Libraries
f) Server-Side form processing on the web using CGI.

1 Script: In Computer programming, a script is a program (or) sequence of instructions that is interpreted (or) carried out by another program [illegible]

Modern scripting languages:

modern scripting include

a) Visual Scripting

b) using Scriptable Components

c) client-Side and Server Side web scripting

Best scripting Languages:

a) Javascript / ECMA Script

b) PHP

c) python

d) Ruby

e) Groovy

f) perl

g) Lua

h) Bash

ECMA (European Computer Manufacturers Association Script) is a Scripting Language based on Javascript.

web scripting

- web script, a computer programming language for adding dynamic capabilities to world wide web pages.
- Web pages marked up with HTML (Hypertext markup Language) or XML (Extensible markup Language) are largely static documents.

Ex: PHP, Python, JavaScript and jQuery

processing Web forms

Dynamic Web pages

Dynamic generated HTML

Java: Java is not a scripting language, it is an object Oriented procedural programming

The universe of scripting Languages:

The original UNIX world of traditional scripting using perl and TCL, which are now available on most platforms,

a) The microsoft world of Visual Basic and ActiveX Controls.

b) The world of VBA for scripting compound documents

c) The world of client-side and server-side web scripting.

What is Ruby and its use?

- Ruby is a server side scripting language similar to python and PERL.
- Ruby is mainly to build web applications and is useful for other programming projects.
- Ruby is widely used for building servers and data processing, web scraping and web crawling.
- The leading framework used to run Ruby is Ruby on Rails.
- Ruby is more organized than perl and python.
- metaprogramming is a feature in Ruby. This makes coding in Ruby even easier.

why Ruby on Rails?

a) It allows you to launch a faster web application.

b) Saves your money by using the Ruby on Rails framework

c) Ruby on Rail Framework makes our app faster and safer

d) Helps us with maintaining and avoiding problems with stuff migration.

e) we can easily update our app with the latest functionality.

f) It uses metaprogramming techniques to write programs.

```
hello_world.rb
 # learn to use varaibles
  hello_world = "Hello World!"
  puts hello_world            fav_num = 42
                              puts fav_num
PS D:\rubyseries> ruby hello_world.rb
         Hello world!
         42
```

```
fav_num = 42
puts   fav_num
```

Web Scraping is the process of using bots to extract content and data from a website. unlike Screen Scraping

web scraping : Extracting the data from ~~on or~~ one (or) more websites.

web crawling : Finding (or) discovering URL's (or) Links on the Web.

chat bot : Its key task is to answer user Questions

What is Scripting Language?

Scripting Language is a Computer language that does not require compilation and is instead interpreted one line at a time during runtime.

most popular Scripting Languages?

Bash
Ruby
Node.js
python
perl

jQuery is a lightweight, "write less, do more," JavaScript library. The purpose of jQuery is to make it much easier to use JavaScript on your website.

Node.js can generate dynamic page content.
Node.js can create, open, read write, delete and close files ~~us~~ on the server.
Node.js can collect form data.

pros & Cons

① Easy to learn
② Quick editing
③ Interactive
④ Functionality

① It allows users to read and code content.
② These languages do not compile and understand the file immediately.
③ Scripting languages may be Slower than compiled programmes.

what are programming languages?

programming languages are the instruments that we employ to write Computer instructions.

popular programming languages?

python	C#
Java	C++
JavaScript	php

programming languages prons & cons:

i) Compatible with a range of operating systems
ii) Good Compatibility with cloud services.
iii) Designed with quick compilation in mind.

a) Not every programming language is adaptable.
b) The Source code is widely accessible.
c) Anyone may read and reuse code written in languages like JavaScript.

Scripting Languages Vs Programming Languages:

Although all scripting languages are programming languages, scripting languages differ in that they directly interpret files rather than requiring Compilation.

Scripting Languages	programming Languages
① It is based on the interpreter	① It is based on the Compiler.
② It's used to put together existing parts	② It's for developing from the ground up.
③ Its job is to translate high-level commands into machine language	③ It convert the entire programme into machine code at once.
④ It is not necessary to compile the file before running it	④ It is necessary to first compile the file.

Scripting Languages	Programming Languages
> It lacks of data types, graphic design and user interface design, ~~and~~ as well as minimal support.	> Graphic design, data formats, and user interface design are all well supported.
> It is simple to write and utilize.	> It is challenging to use and write.
> It requires a host	> It does not require a host because of it is self-contained.
> It is checking only one line at a time.	> It is checking all lines at a time.
> Interpreter is slower than Compiler, because of it is checking/scanning each and every line.	> Compiler is faster > checking/scanning all lines at a time.

New Terminal

```
PS D:\ruby series> ruby ~~-v~~ -V
        ruby 2.7.1 p83 (2020-03-31 revision a0c7c2
                                              3c9c)
                                   (x64-mingw32)

PS D:\ruby series> ls                 hello_world.rb
                                      puts "Hello World"
     hello_world.rb

               > ruby hello_world.rb
                 Hello World!
```

programming Languages	Scripting Languages
① Compiler Based	① Interpreter Based.
② more Syntax and tightly coupled (single memory, Common memory)	② reduced Syntax & loosely coupled (distributed memory)

③ Convert all code to Binary & run
`#include <stdio.h>`
`main()`
`{`
`printf ("Hi");`
`}`
	③ run statement by statement
	`print ('Hi')`

④ for big & Complex programs faster for large code	④ Smaller code
⑤ extra memory	⑤ No extra memory
⑥ C, C++, Java, C#, etc	⑥ python, Javascript, perl, php, Ruby.

step1: ruby installer for windows

step2: download

step3: without DevKit ✓ with DevKit Ruby + DevKit 2.6.X (x64)

step4: RUN ✓

step5: I accept ✓

step6: import [---] Browse

step7: ☑

☑

☑

[install]

step8: Next step9: Finish

14th Feb, 2023
CSE (DS) - B section;
20VE1A6796, (91, 96 - (13th Feb))
CSE (DS)-A
02, 41, Le-2, Le-6

Introduction to Scripting Languages

1. All scripting languages are programming languages.
2. The scripting language is basically a language where instructions are written for a run time environment.
3. They do not require the compilation step and are rather interpreted. It brings new functions to applications and glue complex system together.
4. A scripting language is a programming language designed for integrating and communicating with other programming languages.

There are many scripting languages some of them are discussed below:

1. **bash:** It is a scripting language to work in the Linux interface. It is a lot easier to use bash to create scripts than other programming languages. It describes the tools to use and code in the command line and create useful reusable scripts and conserve documentation for other people to work with.
2. **Node js:** It is a framework to write network applications using **JavaScript**. Corporate users of Node.js include IBM, LinkedIn, Microsoft, Netflix, PayPal, Yahoo for real-time web applications.
3. **Ruby:** There are a lot of reasons to learn Ruby programming language. Ruby’s flexibility has allowed developers to create innovative software. It is a scripting language which is great for web development.
4. **Python:** It is easy, free and open source. It supports procedure-oriented programming and object-oriented programming. Python is an interpreted language with dynamic semantics and huge lines of code are scripted and is currently the most hyped language among developers.
5. **Perl:** A scripting language with innovative features to make it different and popular. Found on all windows and Linux servers. It helps in text manipulation tasks. High traffic websites that use Perl extensively include priceline.com, IMDB.

Advantages of scripting languages:

1. **Easy learning:** The user can learn to code in scripting languages quickly, not much knowledge of web technology is required.
2. **Fast editing:** It is highly efficient with the limited number of data structures and variables to use.
3. **Interactivity:** It helps in adding visualization interfaces and combinations in web pages. Modern web pages demand the use of scripting languages. To create enhanced web pages, fascinated visual description which includes background and foreground colors and so on.
4. **Functionality:** There are different libraries which are part of different scripting languages. They help in creating new applications in web browsers and are different from normal programming languages.

Application of Scripting Languages: Scripting languages are used in many areas:

1. Scripting languages are used in web applications. It is used in server side as well as client side. Server side scripting languages are: JavaScript, PHP, Perl etc. and client side scripting languages are: JavaScript, AJAX, jQuery etc.
2. Scripting languages are used in system administration. For example: Shell, Perl, Python scripts etc.
3. It is used in Games application and Multimedia.
4. It is used to create plugins and extensions for existing applications.

Difference Between Scripting and Programming Languages

Scripting Vs. Programming Languages: Know What is the Difference Between Scripting and Programming Languages

Every scripting language is basically a programming language. The only theoretical difference is that a scripting language does not include the compilation step- it is interpreted instead. For instance, one needs to first compile a C program before running it. On the other hand, one does not need to compile a scripting language such as PHP or JavaScript. There are various ways in which both languages vary. In this article, we will dive deeper into the difference between scripting and programming languages. But let us first take a brief look at both of them.

What is a Scripting Language?

1. Scripting languages help in automating various software apps, web pages in a browser, shell usage of an OS (operating system), etc. The scripting languages like VBScript, Perl, Javascript, etc., do not require compilation, and they have less access to any computer's native abilities. It is because these rather run on an original programming language's subset. An example here could be that the Javascript won't have the ability to access your file system.
2. Generally, a scripting language is interpreted. It doesn't primarily focus on building applications- but it can render behavior to an application that already exists. It basically helps in writing codes for targeting a software system. Thus, it can also automate a given operation on any software system. So basically, scripts act as a set of instructions that target any software system.
3. The scripting languages have eventually evolved and become more powerful. They now no longer create minute scripts for automating a software system's operations. One can also use scripting languages for building rich applications. These can customize, manipulate, and automate an existing system's facilities. The scripting languages come with a mechanism that exposes functionality to the program control.

What is a Programming Language?

- One needs to compile the programming languages to machine code so as to run them on the hardware of an underlying OS (operating system). A user needs to deploy a certain Integrated Development Environment (IDE) for using programming languages. A programmer needs to provide an instruction set for the computers for achieving certain goals. One can also implement certain algorithms by writing the programs.
- Out of all the programming languages present in the market, specific documentation dominates a majority of them. All the other languages comprise dominant implementation (treated as a reference). An example here is that the ISO standard associates with the C programming language. On the other hand, languages like Perl belong to the latter category.

- One can use a programming language for transforming data. It basically happens when creating those CPU instructions that jot down the input info into the output. An example here is using a set of conditions for solving an equation set. One can consider various programming languages such as C, C++, Scala, Java, etc., as general-purpose languages. These fall under the compiled programming languages. You must add some texts to write the score code, and then you can run them through a compiler. As a result, it would create various binary instructions.

Difference Between Scripting and Programming Languages

Parameters	**Scripting Language**	**Programming Language**
Language Type ✓	The scripting languages are interpreter-based languages.	The programming languages are compiler-based languages.
Use ✓	The scripting languages help in combining the existing components of an application.	The programming languages help in developing anything from scratch.
Running of Language	A user needs to run scripting languages inside an existing program. Thus, it's program-dependent.	Programming languages are program-independent.
Conversion	Scripting languages convert high-level instructions into machine language.	Programming languages help in converting the full program into the machine language (at once).
Compilation	You don't need to compile these languages.	These languages first need a compilation.
Design	These make the coding process simple and fast.	These provide full usage of the languages.
File Type	Scripting languages don't create any file types.	Programming languages create .exe files.
Complexity	These are very easy to use and easy to write.	These are pretty complex in terms of writing and usage.
Type of Coding	Scripting languages help write a small piece of an entire code.	Programming languages help write the full code concerning a program.

Developing Time	These take less time because they involve lesser code.	These take more time because a programmer must write the entire code.
Interpretation	We usually interpret a scripting language in another program.	The compile results of a programming language are stand-alone. No other program needs to interpret it.
Requirement of Host	Scripting languages require hosts for execution.	Programming languages are self-executable. They don't require any host.
Length of Codes	These involve very few and short coding lines.	These require numerous lines of coding for a single function.
Support	These provide limited support to data types, user interface design, and graphic design.	These provide rich support for graphic design, data types, and user interface design.
Maintenance	These involve very low maintenance.	These involve high maintenance.
Cost	It is easier and cheaper to maintain a scripting language.	Maintaining a programming language is comparatively more expensive.
Example	VB Script, Perl, Ruby, PHP, JavaScript, etc.	C, C++, COBOL, Basic, VB, C#, Pascal, Java, etc.

Ruby

1. Ruby is a computer programming language developed in 1995 by Yukihiro Matsumoto. He wanted to create a flexible, object-oriented language that programmers would enjoy using. They enjoyed it enough that Ruby became one of the most popular languages for developing web applications.
2. It’s a general use language that’s popular in the industry. Apple, GitHub, Twitter, Hulu, ZenDesk, and Urban Dictionary are websites developed with Ruby, demonstrating its versatility. Ruby is a general use language that's more popular in the industry than in science or academia.

Using Ruby to build applications

1. Ruby is mainly used to build web applications and is useful for other programming projects. It is widely used for building servers and data processing, web scraping, and crawling.
2. The leading framework used to run Ruby is Ruby on Rails, although that’s not the only one. Ruby on Rails was released in 2004 and made the language much easier to use. That’s one reason developers at many start-ups use Ruby to build their applications.

Features

1. Ruby is a general-purpose, object-oriented programming language that runs on Mac, Windows, Unix, and most operating systems.
2. It has a flexible approach to solving problems, which some programmers appreciate and some do not.

Advantages

1. Ruby’s syntax is similar to English, so many English speakers find it easy to learn and use. The program itself is free, and it’s open-source, with users sharing improvements and ideas for how to use it.
2. The Ruby community tends to focus on web development over other types of programming and has created a vast library of program elements.

Disadvantages

1. One of the disadvantages of Ruby’s user-friendly approach is that bugs can get hidden, making it more difficult to find and fix code problems, mainly because the documentation for Ruby isn’t as complete as it is for some other languages.
2. Also, Ruby and Ruby for Rails tend to take longer to boot and have a slower runtime than other programming platforms.

The Benefits and Applications of Using Ruby as a Programming Language

1. Ruby is one of the most reliable programming languages in town, which comes with its own Rails framework. The object-oriented programming language was first developed in 1995 and is among the top 10 programming languages, reviewed by multiple analysts.
2. Developers working on the Ruby platform can build high quality web applications, which boast clean architecture and implement all of CSS, HTML and JavaScript files. Ruby was further augmented with a proper framework in Ruby on Rails, which has become even more popular over time and is now extensively used by many.
3. Ruby is considered by a number of programmers and developers today and comes with a wide range of features. In this article, we take a look at some of the benefits and applications of Ruby as a programming language.

Easy Changes

1. Ruby is a simple programming language, which can simplify the changes in codes for developers. Developers would know that most projects require extensive changes, which are easy to manage on Ruby. Ruby can simplify the change process, which gives businesses a simple solution to their problems.
2. Ruby can also prove extremely useful for organizations looking to scale up their operations in the near future. The application can easily be grown over time and processes can be updated easily.

Extremely Secure

1. Ruby is ranked among some of the best programming languages and is trusted by developers for this very reason. Ruby puts a strong emphasis on securing solutions made on it.
2. The programming language stores and holds all objects based on reference rather than value to prevent any data from being hijacked or overwritten.

3. The secure environment on Ruby allows organizations to secure all forms of sensitive information and make sure that the information isn’t accessible to external threat actors.

Fun to Code

1. Programming and coding on Ruby can be simple and fun for all developers. Developers who have worked on Ruby will be able to vouch for its interactive UI and how it is a lot easier to understand than other programming languages.
2. Ruby’s simple syntax makes it perfect for newbies who are still getting a hack of how the process works. Ruby can simplify difficult programming concepts and give programmers a chance to work on them and provide simple solutions. The simple syntax also ensures that programmers can create solutions easily without spending too much on them.

Faster Processing

1. It is a lot easier for developers to configure and develop solutions on Ruby. The programming language comes with multi-threading or native thread support, which can allow the solution to operate multiple programs at one time without slowing down your system at any time. The fast web application makes Ruby extremely suitable for projects with a quick ETA.
2. Faster operations are also possible because of the portability of the language. The language is extremely portable, which ensures that it can be run on almost all operating systems.
3. Ruby isn’t dependent on any external factors, which makes it easy for developers to use it across operating systems. Ruby isn’t just quick but can also be used extensively for cross-platform programming and development.

Open Source and Flexible

1. Businesses today like using Ruby because it is extremely flexible to use and comes with an open-source library. The flexibility offered by Ruby helps give developers the option to add multiple objects and methods to the solution.
2. ROR developers can add objects to all existing classes without disrupting stability in any way. This can help developers make flexible APIs.
3. Additionally, Ruby is also open source in nature, which gives developers the feasibility to share their codes with other programmers.
4. The open-source network can help make programming easier for new beginners. All users can gain access to helpful codes and can use them in their solutions.

Consistent in Nature

Perhaps the biggest benefit of using Ruby is that it is generally consistent in nature. The syntax for Ruby is generally consistent and allows you to build skills and solutions without learning a lot of new things. You can create programs in the language without going through a significant learning curve.

Application of Ruby as a Programming Language

1. Ruby can work well for dynamic websites and long-term solutions. Ruby is the perfect solution for programming a general-purpose application.
2. Ruby comes with all the necessary adjustments required to run sprawling apps such as Shopify and GitHub.
3. Ruby is also preferred by Bloomberg, Airbnb, Apple, Groupon, Hulu and Dribble, among other big names. The following categories can benefit the most from the provisions on Ruby.

E-Commerce Sites

1. Ruby can help facilitate the development of e-commerce sites and solutions. The programming language provides solutions that can meet business requirements.
2. Ruby can help assist in uploading product images, setting price algorithms, updating image processing and a number of other processes.

Content Sites

1. All content sites that upload audio content, reading material or visual content can benefit from Ruby.
2. Ruby comes with a fast upload procedure, and the coding on it is relatively simple. The simple operations make coding easier for all involved.

Social Networking Sites

1. While Ruby isn’t the default programming language for large web apps with millions of users interacting at one time, the plug-ins within Ruby can help manage the operations well.
2. Ruby is an ideal solution for your development project as it helps improve upload times and can simplify the programming process. With the information in this article, you can make a concrete decision on whether to use it for your next project or not.

MIT License: Massachusetts Institute of Technology in the late 1980's.

Ruby on Rails Introduction

Ruby on Rails or also known as rails is a server-side web application development framework that is written in the Ruby programming language, and it is developed by David Heinemeier Hansson under the MIT License. It supports MVC(model-view-controller) architecture that provides a default structure for database, web pages, and web services, it also uses web standards like JSON or XML for transfer data and HTML, CSS, and JavaScript for the user interface. It emphasizes the use of other well-known software engineering pattern and paradigms like:

- **Don't Repeat Yourself (DRY):** It is a principle of software development to reducing the repetition of information or codes.
- **Convention Over Configuration (CoC):** It provides many opinions for the best way to do many things in a web application.

Ruby on Rails was first released in July 2004 but until February 2005 did not share the commit rights. In August 2006, it would ship Ruby on Rails with Mac OS X v10.5 "Leopard". Ruby on Rail's latest version(Rail 5.0.1) released on December 21, 2016. Action cable, Turbolinks 5, and API mode Introduced in this version.

JSON : Java Script object Notation

Why Ruby on Rails?

- It allows you to launch a faster web application.
- Saves your money by using the Ruby on Rails framework.
- Helps us with maintaining and avoiding problems with stuff migration.
- Ruby on Rail Framework makes our app faster and safer.
- We can easily update our app with the latest functionality.
- It uses Metaprogramming techniques to write programs.

Where to use Ruby on Rails?

You can use Ruby on Rails application in various area of web development like in a long term project which needs large transformation, or in the project that has heavy traffic, or to develop a short prototype or MVPs, or in a project that requires wide range of complex functions, etc. (which means putting in a little more effort)

Feature of Ruby on Rails

As we know that most of the languages like Java, HTML, CSS, etc. do not cover the front end and back end. They either only for the back end or for the front end but Ruby on Rails is used for both front end back end, it is like a complete package to develop a web application. Some important features of Ruby on Rails are:

1. Model-view-controller Architecture: Ruby on Rails used MVC architecture, and it contains three components, i.e., model, view, and controller. Here, the model is used to maintain the relationship between object and database, the view is templates that are used to build the data users for web applications, and the controller is used to merge model and view together. MVC is generally used for developing user interfaces that divide the data into three interconnected components so that it can separate the internal representation of the information from the way it presents to and get from the user.

MVC Architecture: It is an architectural pattern that separates an application into 3 main logical components: The model, The View, and the controller. Each of these components are built to handle specific ↑ development aspects of an application.

metaprogramming: It is the ability to generate code on the fly (or) is it the ability to injects methods and attributes into existing objects at runtime (like python, Ruby and Groovy)

2. Active Records: The active record framework is introduced in Ruby on Rails. It is a powerful library that allows the developer to design the database interactive queries.
3. Built-in Testing: Ruby on Rails provides its own set of tests that will run on your code. It will save time and effort.
4. Programming Language: This syntax of Ruby on Rails is simple because the syntax of the Ruby programming language is close to English, so it is always easier to structure your thinking and writing it into code.
5. MetaProgramming: Ruby on rails uses the metaprogramming technique to write programs.
6. Convention over configuration: In Ruby on Rails, a programmer can only specify the unconventional aspects of the application.
7. Scaffolding: Ruby on rails provides a scaffolding feature in which the developer is allowed to define how the application database works. After defining the work of the application database the framework automatically generates the required code according to the given definition. This technique creates interfaces automatically.

Advantages of Ruby on Rails

- **Tooling:** Rails provides tooling that helps us to deliver more features in less time.
- **Libraries:** There's a 3rd party module(gem) for just about anything we can think of.
- **Code Quality:** Ruby code quality significantly higher than PHP or NodeJS equivalents.
- **Test Automation:** The Ruby community is big into and test automation and testing.
- **Large Community:** Ruby is large in the community.
- **Productivity:** Ruby is incredibly fast from another language. Its productivity is high.

Disadvantages of Ruby on Rails

- **Runtime Speed:** The run time speed of Ruby on Rails is slow as compare to Node.Js and Golang.
- **Lack of Flexibility:** As we know that Ruby on Rails is ideal for standard web applications due to its hard dependency between components and models. But when it comes to adding unique functionality and customization in apps it is challenging.
- **Boot Speed:** The boot speed is also a drawback of ROR. Due to the dependence upon the number of gem dependencies and files, it takes some time to start which can obstruct the developer performance.
- **Documentation:** To find good documentation is hard for the less popular gems and for libraries that make heavy use of mixins.
- **Multithreading:** Ruby on Rails supports multithreading, but some IO libraries do not support multithreading because they keep hold of the global interpreter

metaprogramming refers to a variety of ways a program has knowledge of itself (or) can manipulate itself.

Difference between Ruby and Ruby on Rails

1. Ruby :

Ruby is an object-oriented scripting language launched in 1995 and is known as a general-purpose programming language. It was programmed in C programming language. Ruby is a secured programming language and its syntax is similar to Perl and Python. It was developed on the principle of user interface design and it is mainly used to develop desktop applications. While developing applications mainly C++, Java, VB.net are used.

Some of the top **companies which are using Ruby** are Github, Twitter, Airbnb, SCRIBD, Slideshare, Fiverr, etc.

Pros of Ruby :

1. Good memory Management & Garbage Collection.
2. Good Dependency Management.
3. Instant Gratification.

Cons of Ruby :

1. Syntax Complexity and error arise.
2. Supports multiple programming paradigms
3. Shared Mutable State.

2. Ruby on Rails :

Ruby on Rails is a web app development framework based on MVC system and it is known as a framework for data base driven web app. It was programmed in Ruby programming language. It is considered as more secure than Ruby language and its syntax is similar to Phoenix in Elixir, Python. It was developed on the principle of DRY (Don't Repeat Yourself) and COC (Convention Over Configuration) and it is mainly used to develop web applications. While developing applications mainly HTML, CSS, JavaScript and XML are used.

Some of the top **companies which are using Ruby on Rails** are Bloomberg, Crunchbase, zendesk, PIXLR, etc.

Pros of Ruby on Rails :

1. Secure Tool
2. Versatile
3. Cost-Effective

Cons of Ruby on Rails :

1. Runtime Speed and Performance.
2. Absence of Flexibility.
3. High Expense in the Development.

Difference between Ruby and Ruby on Rails :

S.NO.	RUBY	RUBY ON RAILS
01.	Ruby is an object-oriented scripting language launched in 1995.	Ruby on Rails is a web app development framework based on MVC system.
02.	It is known as a general-purpose programming language.	Where as it is known as a framework for data base driven web app.
03.	It was programmed in C programming language.	It was programmed in Ruby programming language.
04.	It is considered as a secure programming language.	While it is considered as more secure than Ruby language.
05.	It is not a framework.	While it is a web development framework.
06.	Ruby is commonly used in static website development.	Ruby on Rails is not generally recommended when creating static website.
07.	Ruby programming language is considered as taking inspiration from Perl and Smalltalk.	Ruby on Rails is considered as taking inspiration from Django, Python's Laravel, and PHP, respectively.
08.	Ruby programming language is used to develop desktop applications.	While it is used to develop web applications.
09.	It was developed on the principle of user interface design.	It was developed on the principle of DRY and COC.
10.	Its syntax is similar to Perl and Python.	Its syntax is similar to Phoenix in Elixir, Python.
11.	While developing applications mainly C++, Java, VB.net are used.	While developing applications mainly HTML, CSS, JavaScript and XML are used.
12.	Some of the top companies which are using Ruby are Github, Twitter, airbnb, SCRIBD, slideshare, fiverr etc.	Some of the top companies which are using Ruby on Rails are Bloomberg, crunchbase, zendesk, PIXLR etc.

CHAPTER 2

The Structure and Execution of Ruby Programs

This chapter explains the structure of Ruby programs. It starts with the lexical structure, covering tokens and the characters that comprise them. Next, it covers the syntactic structure of a Ruby program, explaining how expressions, control structures, methods, classes, and so on are written as a series of tokens. Finally, the chapter describes files of Ruby code, explaining how Ruby programs can be split across multiple files and how the Ruby interpreter executes a file of Ruby code.

2.1 Lexical Structure

The Ruby interpreter parses a program as a sequence of *tokens*. Tokens include comments, literals, punctuation, identifiers, and keywords. This section introduces these types of tokens and also includes important information about the characters that comprise the tokens and the whitespace that separates the tokens.

2.1.1 Comments

Comments in Ruby begin with a `#` character and continue to the end of the line. The Ruby interpreter ignores the `#` character and any text that follows it (but does not ignore the newline character, which is meaningful whitespace and may serve as a statement terminator). If a `#` character appears within a string or regular expression literal (see Chapter 3), then it is simply part of the string or regular expression and does not introduce a comment:

```
# This entire line is a comment
x = "#This is a string"              # And this is a comment
y = /#This is a regular expression/   # Here's another comment
```

Multiline comments are usually written simply by beginning each line with a separate `#` character:

```
#
# This class represents a Complex number
# Despite its name, it is not complex at all.
#
```

Note that Ruby has no equivalent of the C-style `/*...*/` comment. There is no way to embed a comment in the middle of a line of code.

2.1.1.1 Embedded documents

Ruby supports another style of multiline comment known as an *embedded document*. These start on a line that begins `=begin` and continue until (and include) a line that begins `=end`. Any text that appears after `=begin` or `=end` is part of the comment and is also ignored, but that extra text must be separated from the `=begin` and `=end` by at least one space.

Embedded documents are a convenient way to comment out long blocks of code without prefixing each line with a `#` character:

```
=begin Someone needs to fix the broken code below!
    Any code here is commented out
=end
```

Note that embedded documents only work if the = signs are the first characters of each line:

```
# =begin This used to begin a comment. Now it is itself commented out!
    The code that goes here is no longer commented out
# =end
```

As their name implies, embedded documents can be used to include long blocks of documentation within a program, or to embed source code of another language (such as HTML or SQL) within a Ruby program. Embedded documents are usually intended to be used by some kind of postprocessing tool that is run over the Ruby source code, and it is typical to follow `=begin` with an identifier that indicates which tool the comment is intended for.

2.1.1.2 Documentation comments

Ruby programs can include embedded API documentation as specially formatted comments that precede method, class, and module definitions. You can browse this documentation using the *ri* tool described earlier in §1.2.4. The *rdoc* tool extracts documentation comments from Ruby source and formats them as HTML or prepares them for display by *ri*. Documentation of the *rdoc* tool is beyond the scope of this book; see the file *lib/rdoc/README* in the Ruby source code for details.

Documentation comments must come immediately before the module, class, or method whose API they document. They are usually written as multiline comments where each line begins with `#`, but they can also be written as embedded documents that start `=begin rdoc`. (The *rdoc* tool will not process these comments if you leave out the "`rdoc`".)

The following example comment demonstrates the most important formatting elements of the markup grammar used in Ruby's documentation comments; a detailed description of the grammar is available in the *README* file mentioned previously:

```
#
# Rdoc comments use a simple markup grammar like those used in wikis.
#
# Separate paragraphs with a blank line.
#
# = Headings
#
# Headings begin with an equals sign
#
# == Sub-Headings
# The line above produces a subheading.
# === Sub-Sub-Heading
# And so on.
#
# = Examples
```

```
#
#   Indented lines are displayed verbatim in code font.
#     Be careful not to indent your headings and lists, though.
#
# = Lists and Fonts
#
# List items begin with * or -. Indicate fonts with punctuation or HTML:
# * _italic_ or <i>multi-word italic</i>
# * *bold* or <b>multi-word bold</b>
# * +code+ or <tt>multi-word code</tt>
#
# 1. Numbered lists begin with numbers.
# 99. Any number will do; they don't have to be sequential.
# 1. There is no way to do nested lists.
#
# The terms of a description list are bracketed:
# [item 1]  This is a description of item 1
# [item 2]  This is a description of item 2
#
```

2.1.2 Literals

Literals are values that appear directly in Ruby source code. They include numbers, strings of text, and regular expressions. (Other literals, such as array and hash values, are not individual tokens but are more complex expressions.) Ruby number and string literal syntax is actually quite complicated, and is covered in detail in Chapter 3. For now, an example suffices to illustrate what Ruby literals look like:

```
1                     # An integer literal
1.0                   # A floating-point literal
'one'                 # A string literal
"two"                 # Another string literal
/three/               # A regular expression literal
```

2.1.3 Punctuation

Ruby uses punctuation characters for a number of purposes. Most Ruby operators are written using punctuation characters, such as + for addition, * for multiplication, and || for the Boolean OR operation. See §4.6 for a complete list of Ruby operators. Punctuation characters also serve to delimit string, regular expression, array, and hash literals, and to group and separate expressions, method arguments, and array indexes. We'll see miscellaneous other uses of punctuation scattered throughout Ruby syntax.

2.1.4 Identifiers

An *identifier* is simply a name. Ruby uses identifiers to name variables, methods, classes, and so forth. Ruby identifiers consist of letters, numbers, and underscore characters, but they may not begin with a number. Identifiers may not include whitespace or nonprinting characters, and they may not include punctuation characters except as described here.

Identifiers that begin with a capital letter A–Z are constants, and the Ruby interpreter will issue a warning (but not an error) if you alter the value of such an identifier. Class and module names must begin with initial capital letters. The following are identifiers:

```
i
x2
old_value
_internal    # Identifiers may begin with underscores
PI           # Constant
```

By convention, multiword identifiers that are not constants are written with underscores `like_this`, whereas multiword constants are written `LikeThis` or `LIKE_THIS`.

2.1.4.1 Case sensitivity

Ruby is a case-sensitive language. Lowercase letters and uppercase letters are distinct. The keyword `end`, for example, is completely different from the keyword `END`.

2.1.4.2 Unicode characters in identifiers

Ruby's rules for forming identifiers are defined in terms of ASCII characters that are not allowed. In general, all characters outside of the ASCII character set are valid in identifiers, including characters that appear to be punctuation. In a UTF-8 encoded file, for example, the following Ruby code is valid:

```
def ×(x,y)  # The name of this method is the Unicode multiplication sign
  x*y       # The body of this method multiplies its arguments
end
```

Similarly, a Japanese programmer writing a program encoded in SJIS or EUC can include Kanji characters in her identifiers. See §2.4.1 for more about writing Ruby programs using encodings other than ASCII.

The special rules about forming identifiers are based on ASCII characters and are not enforced for characters outside of that set. An identifier may not begin with an ASCII digit, for example, but it may begin with a digit from a non-Latin alphabet. Similarly, an identifier must begin with an ASCII capital letter in order to be considered a constant. The identifier `Å`, for example, is not a constant.

Two identifiers are the same only if they are represented by the same sequence of bytes. Some character sets, such as Unicode, have more than one codepoint that represents the same character. No Unicode normalization is performed in Ruby, and two distinct codepoints are treated as distinct characters, even if they have the same meaning or are represented by the same font glyph.

2.1.4.3 Punctuation in identifiers

Punctuation characters may appear at the start and end of Ruby identifiers. They have the following meanings:

`$`	Global variables are prefixed with a dollar sign. Following Perl's example, Ruby defines a number of global variables that include other punctuation characters, such as `$_` and `$-K`. See Chapter 10 for a list of these special globals.
`@`	Instance variables are prefixed with a single at sign, and class variables are prefixed with two at signs. Instance variables and class variables are explained in Chapter 7.
`?`	As a helpful convention, methods that return Boolean values often have names that end with a question mark.
`!`	Method names may end with an exclamation point to indicate that they should be used cautiously. This naming convention is often to distinguish mutator methods that alter the object on which they are invoked from variants that return a modified copy of the original object.
`=`	Methods whose names end with an equals sign can be invoked by placing the method name, without the equals sign, on the left side of an assignment operator. (You can read more about this in §4.5.3 and §7.1.5.)

Here are some example identifiers that contain leading or trailing punctuation characters:

```
$files          # A global variable
@data           # An instance variable
@@counter       # A class variable
empty?          # A Boolean-valued method or predicate
sort!           # An in-place alternative to the regular sort method
timeout=        # A method invoked by assignment
```

A number of Ruby's operators are implemented as methods, so that classes can redefine them for their own purposes. It is therefore possible to use certain operators as method names as well. In this context, the punctuation character or characters of the operator are treated as identifiers rather than operators. See §4.6 for more about Ruby's operators.

2.1.5 Keywords

The following keywords have special meaning in Ruby and are treated specially by the Ruby parser:

```
__LINE__      case        ensure      not         then
__ENCODING__  class       false       or          true
__FILE__      def         for         redo        undef
BEGIN         defined?    if          rescue      unless
END           do          in          retry       until
alias         else        module      return      when
and           elsif       next        self        while
begin         end         nil         super       yield
break
```

In addition to those keywords, there are three keyword-like tokens that are treated specially by the Ruby parser when they appear at the beginning of a line:

```
=begin    =end      __END__
```

As we've seen, `=begin` and `=end` at the beginning of a line delimit multiline comments. And the token `__END__` marks the end of the program (and the beginning of a data section) if it appears on a line by itself with no leading or trailing whitespace.

In most languages, these words would be called "reserved words" and they would be never allowed as identifiers. The Ruby parser is flexible and does not complain if you prefix these keywords with `@`, `@@`, or `$` prefixes and use them as instance, class, or global variable names. Also, you can use these keywords as method names, with the caveat that the method must always be explicitly invoked through an object. Note, however, that using these keywords in identifiers will result in confusing code. The best practice is to treat these keywords as reserved.

Many important features of the Ruby language are actually implemented as methods of the `Kernel`, `Module`, `Class`, and `Object` classes. It is good practice, therefore, to treat the following identifiers as reserved words as well:

```
# These are methods that appear to be statements or keywords
at_exit        catch          private        require
attr           include        proc           throw
attr_accessor  lambda         protected
attr_reader    load           public
attr_writer    loop           raise

# These are commonly used global functions
Array          chomp!         gsub!          select
Float          chop           iterator?      sleep
Integer        chop!          load           split
String         eval           open           sprintf
URI            exec           p              srand
abort          exit           print          sub
autoload       exit!          printf         sub!
autoload?      fail           putc           syscall
binding        fork           puts           system
block_given?   format         rand           test
callcc         getc           readline       trap
caller         gets           readlines      warn
chomp          gsub           scan

# These are commonly used object methods
allocate       freeze         kind_of?       superclass
clone          frozen?        method         taint
display        hash           methods        tainted?
dup            id             new            to_a
enum_for       inherited      nil?           to_enum
eql?           inspect        object_id      to_s
equal?         instance_of?   respond_to?    untaint
extend         is_a?          send
```

2.1.6 Whitespace

Spaces, tabs, and newlines are not tokens themselves but are used to separate tokens that would otherwise merge into a single token. Aside from this basic token-separating function, most whitespace is ignored by the Ruby interpreter and is simply used to format programs so that they are easy to read and understand. Not all whitespace is ignored, however. Some is required, and some whitespace is actually forbidden. Ruby's grammar is expressive but complex, and there are a few cases in which inserting or removing whitespace can change the meaning of a program. Although these cases do not often arise, it is important to know about them.

2.1.6.1 Newlines as statement terminators

The most common form of whitespace dependency has to do with newlines as statement terminators. In languages like C and Java, every statement must be terminated with a semicolon. You can use semicolons to terminate statements in Ruby, too, but this is only required if you put more than one statement on the same line. Convention dictates that semicolons be omitted elsewhere.

Without explicit semicolons, the Ruby interpreter must figure out on its own where statements end. If the Ruby code on a line is a syntactically complete statement, Ruby uses the newline as the statement terminator. If the statement is not complete, then Ruby continues parsing the statement on the next line. (In Ruby 1.9, there is one exception, which is described later in this section.)

This is no problem if all your statements fit on a single line. When they don't, however, you must take care that you break the line in such a way that the Ruby interpreter cannot interpret the first line as a statement of its own. This is where the whitespace dependency lies: your program may behave differently depending on where you insert a newline. For example, the following code adds `x` and `y` and assigns the sum to `total`:

```
total = x +     # Incomplete expression, parsing continues
  y
```

But this code assigns `x` to `total`, and then evaluates `y`, doing nothing with it:

```
total = x  # This is a complete expression
  + y      # A useless but complete expression
```

As another example, consider the `return` and `break` statements. These statements may optionally be followed by an expression that provides a return value. A newline between the keyword and the expression will terminate the statement before the expression.

You can safely insert a newline without fear of prematurely terminating your statement after an operator or after a period or comma in a method invocation, array literal, or hash literal.

You can also escape a line break with a backslash, which prevents Ruby from automatically terminating the statement:

```
var total = first_long_variable_name + second_long_variable_name \
  + third_long_variable_name # Note no statement terminator above
```

In Ruby 1.9, the statement terminator rules change slightly. If the first nonspace character on a line is a period, then the line is considered a continuation line, and the newline before it is not a statement terminator. Lines that start with periods are useful for the long method chains sometimes used with “fluent APIs,” in which each method invocation returns an object on which additional invocations can be made. For example:

```
animals = Array.new
  .push("dog")   # Does not work in Ruby 1.8
  .push("cow")
  .push("cat")
  .sort
```

2.1.6.2 Spaces and method invocations

Ruby’s grammar allows the parentheses around method invocations to be omitted in certain circumstances. This allows Ruby methods to be used as if they were statements, which is an important part of Ruby’s elegance. Unfortunately, however, it opens up a pernicious whitespace dependency. Consider the following two lines, which differ only by a single space:

```
f(3+2)+1
f (3+2)+1
```

The first line passes the value `5` to the function `f` and then adds `1` to the result. Since the second line has a space after the function name, Ruby assumes that the parentheses around the method call have been omitted. The parentheses that appear after the space are used to group a subexpression, but the entire expression `(3+2)+1` is used as the method argument. If warnings are enabled (with `-w`), Ruby issues a warning whenever it sees ambiguous code like this.

The solution to this whitespace dependency is straightforward:

- Never put a space between a method name and the opening parenthesis.
- If the first argument to a method begins with an open parenthesis, always use parentheses in the method invocation. For example, write `f((3+2)+1)`.
- Always run the Ruby interpreter with the `-w` option so it will warn you if you forget either of the rules above!

2.2 Syntactic Structure

So far, we’ve discussed the tokens of a Ruby program and the characters that make them up. Now we move on to briefly describe how those lexical tokens combine into the larger syntactic structures of a Ruby program. This section describes the syntax of Ruby programs, from the simplest expressions to the largest modules. This section is, in effect, a roadmap to the chapters that follow.

The basic unit of syntax in Ruby is the *expression*. The Ruby interpreter *evaluates* expressions, producing values. The simplest expressions are *primary expressions*, which represent values directly. Number and string literals, described earlier in this chapter, are primary expressions. Other primary expressions include certain keywords such as `true`, `false`, `nil`, and `self`. Variable references are also primary expressions; they evaluate to the value of the variable.

More complex values can be written as compound expressions:

```
[1,2,3]                # An Array literal
{1=>"one", 2=>"two"}   # A Hash literal
1..3                   # A Range literal
```

Operators are used to perform computations on values, and compound expressions are built by combining simpler subexpressions with operators:

```
1         # A primary expression
x         # Another primary expression
x = 1     # An assignment expression
x = x + 1 # An expression with two operators
```

Chapter 4 covers operators and expressions, including variables and assignment expressions.

Expressions can be combined with Ruby's keywords to create *statements*, such as the `if` statement for conditionally executing code and the `while` statement for repeatedly executing code:

```
if x < 10 then   # If this expression is true
  x = x + 1      # Then execute this statement
end              # Marks the end of the conditional

while x < 10 do  # While this expression is true...
  print x        # Execute this statement
  x = x + 1      # Then execute this statement
end              # Marks the end of the loop
```

In Ruby, these statements are technically expressions, but there is still a useful distinction between expressions that affect the control flow of a program and those that do not. Chapter 5 explains Ruby's control structures.

In all but the most trivial programs, we usually need to group expressions and statements into parameterized units so that they can be executed repeatedly and operate on varying inputs. You may know these parameterized units as functions, procedures, or subroutines. Since Ruby is an object-oriented language, they are called *methods*. Methods, along with related structures called *procs* and *lambdas*, are the topic of Chapter 6.

Finally, groups of methods that are designed to interoperate can be combined into *classes*, and groups of related classes and methods that are independent of those classes can be organized into *modules*. Classes and modules are the topic of Chapter 7.

2.2.1 Block Structure in Ruby

Ruby programs have a block structure. Module, class, and method definitions, and most of Ruby's statements, include blocks of nested code. These blocks are delimited by keywords or punctuation and, by convention, are indented two spaces relative to the delimiters. There are two kinds of blocks in Ruby programs. One kind is formally called a "block." These blocks are the chunks of code associated with or passed to iterator methods:

```
3.times { print "Ruby! " }
```

In this code, the curly braces and the code inside them are the block associated with the iterator method invocation `3.times`. Formal blocks of this kind may be delimited with curly braces, or they may be delimited with the keywords `do` and `end`:

```
1.upto(10) do |x|
  print x
end
```

`do` and `end` delimiters are usually used when the block is written on more than one line. Note the two-space indentation of the code within the block. Blocks are covered in §5.4.

To avoid ambiguity with these true blocks, we can call the other kind of block a *body* (in practice, however, the term "block" is often used for both). A body is just the list of statements that comprise the body of a class definition, a method definition, a `while` loop, or whatever. Bodies are never delimited with curly braces in Ruby—keywords usually serve as the delimiters instead. The specific syntax for statement bodies, method bodies, and class and module bodies are documented in Chapters 5, 6, and 7.

Bodies and blocks can be nested within each other, and Ruby programs typically have several levels of nested code, made readable by their relative indentation. Here is a schematic example:

```
module Stats                          # A module
  class Dataset                       # A class in the module
    def initialize(filename)          # A method in the class
      IO.foreach(filename) do |line|  # A block in the method
        if line[0,1] == "#"           # An if statement in the block
          next                        # A simple statement in the if
        end                           # End the if body
      end                             # End the block
    end                               # End the method body
  end                                 # End the class body
end                                   # End the module body
```

2.3 File Structure

There are only a few rules about how a file of Ruby code must be structured. These rules are related to the deployment of Ruby programs and are not directly relevant to the language itself.

First, if a Ruby program contains a "shebang" comment, to tell the (Unix-like) operating system how to execute it, that comment must appear on the first line.

Second, if a Ruby program contains a "coding" comment (as described in §2.4.1), that comment must appear on the first line or on the second line if the first line is a shebang.

Third, if a file contains a line that consists of the single token `__END__` with no whitespace before or after, then the Ruby interpreter stops processing the file at that point. The remainder of the file may contain arbitrary data that the program can read using the `IO` stream object `DATA`. (See Chapter 10 and §9.7 for more about this global constant.)

Ruby programs are not required to fit in a single file. Many programs load additional Ruby code from external libraries, for example. Programs use `require` to load code from another file. `require` searches for specified modules of code against a search path, and prevents any given module from being loaded more than once. See §7.6 for details.

The following code illustrates each of these points of Ruby file structure:

```
#!/usr/bin/ruby -w          shebang comment
# -*- coding: utf-8 -*-     coding comment
require 'socket'            load networking library

  ...                       program code goes here

__END__                     mark end of code
  ...                       program data goes here
```

2.4 Program Encoding

At the lowest level, a Ruby program is simply a sequence of characters. Ruby's lexical rules are defined using characters of the ASCII character set. Comments begin with the `#` character (ASCII code 35), for example, and allowed whitespace characters are horizontal tab (ASCII 9), newline (10), vertical tab (11), form feed (12), carriage return (13), and space (32). All Ruby keywords are written using ASCII characters, and all operators and other punctuation are drawn from the ASCII character set.

By default, the Ruby interpreter assumes that Ruby source code is encoded in ASCII. This is not required, however; the interpreter can also process files that use other encodings, as long as those encodings can represent the full set of ASCII characters. In order for the Ruby interpreter to be able to interpret the bytes of a source file as characters, it must know what encoding to use. Ruby files can identify their own encodings or you can tell the interpreter how they are encoded. Doing so is explained shortly.

The Ruby interpreter is actually quite flexible about the characters that appear in a Ruby program. Certain ASCII characters have specific meanings, and certain ASCII characters are not allowed in identifiers, but beyond that, a Ruby program may contain any characters allowed by the encoding. We explained earlier that identifiers may contain characters outside of the ASCII character set. The same is true for comments and string and regular expression literals: they may contain any characters other than the delimiter character that marks the end of the comment or literal. In ASCII-encoded files, strings may include arbitrary bytes, including those that represent nonprinting control characters. (Using raw bytes like this is not recommended, however; Ruby string literals support escape sequences so that arbitrary characters can be included by numeric code instead.) If the file is written using the UTF-8 encoding, then comments, strings, and regular expressions may include arbitrary Unicode characters. If the file is encoded using the Japanese SJIS or EUC encodings, then strings may include Kanji characters.

2.4.1 Specifying Program Encoding

By default, the Ruby interpreter assumes that programs are encoded in ASCII. In Ruby 1.8, you can specify a different encoding with the `-K` command-line option. To run a Ruby program that includes Unicode characters encoded in UTF-8, invoke the interpreter with the `-Ku` option. Programs that include Japanese characters in EUC-JP or SJIS encodings can be run with the `-Ke` and `-Ks` options.

Ruby 1.9 also supports the `-K` option, but it is no longer the preferred way to specify the encoding of a program file. Rather than have the user of a script specify the encoding when they invoke Ruby, the author of the script can specify the encoding of the script by placing a special "coding comment" at the start of the file.[*] For example:

```
# coding: utf-8
```

The comment must be written entirely in ASCII, and must include the string `coding` followed by a colon or equals sign and the name of the desired encoding (which cannot include spaces or punctuation other than hyphen and underscore). Whitespace is allowed on either side of the colon or equals sign, and the string `coding` may have any prefix, such as `en` to spell `encoding`. The entire comment, including `coding` and the encoding name, is case-insensitive and can be written with upper- or lowercase letters.

Encoding comments are usually written so that they also inform a text editor of the file encoding. Emacs users might write:

```
# -*- coding: utf-8 -*-
```

And vi users can write:

```
# vi: set fileencoding=utf-8 :
```

An encoding comment like this one is usually only valid on the first line of the file. It may appear on the second line, however, if the first line is a shebang comment (which makes a script executable on Unix-like operating systems):

```
#!/usr/bin/ruby -w
# coding: utf-8
```

[*] Ruby follows Python's conventions in this; see *http://www.python.org/dev/peps/pep-0263/*.

Encoding names are not case-sensitive and may be written in uppercase, lowercase, or a mix. Ruby 1.9 supports at least the following source encodings: ASCII-8BIT (also known as BINARY), US-ASCII (7-bit ASCII), the European encodings ISO-8859-1 through ISO-8859-15, the Unicode encoding UTF-8, and the Japanese encodings SHIFT_JIS (also known as SJIS) and EUC-JP. Your build or distribution of Ruby may support additional encodings as well.

As a special case, UTF-8-encoded files identify their encoding if the first three bytes of the file are 0xEF 0xBB 0xBF. These bytes are known as the BOM or "Byte Order Mark" and are optional in UTF-8-encoded files. (Certain Windows programs add these bytes when saving Unicode files.)

In Ruby 1.9, the language keyword `__ENCODING__` (there are two underscores at the beginning and at the end) evaluates to the source encoding of the currently executing code. The resulting value is an `Encoding` object. (See §3.2.6.2 for more on the `Encoding` class.)

2.4.2 Source Encoding and Default External Encoding

In Ruby 1.9, it is important to understand the difference between the *source encoding* of a Ruby file and the *default external encoding* of a Ruby process. The source encoding is what we described earlier: it tells the Ruby interpreter how to read characters in a script. Source encodings are typically set with coding comments. A Ruby program may consist of more than one file, and different files may have different source encodings. The source encoding of a file affects the encoding of the string literals in that file. For more about the encoding of strings, see §3.2.6.

The default external encoding is something different: this is the encoding that Ruby uses by default when reading from files and streams. The default external encoding is global to the Ruby process and does not change from file to file. Normally, the default external encoding is set based on the locale that your computer is configured to. But you can also explicitly specify the default external encoding with command-line options, as we'll describe shortly. The default external encoding does not affect the encoding of string literals, but it is quite important for I/O, as we'll see in §9.7.2.

We described the `-K` interpreter option earlier as a way to set the source encoding. In fact, what this option really does is set the default external encoding of the process and then uses that encoding as the default source encoding.

In Ruby 1.9, the `-K` option exists for compatibility with Ruby 1.8 but is not the preferred way to set the default external encoding. Two new options, `-E` and `--encoding`, allow you to specify an encoding by its full name rather than by a one-character abbreviation. For example:

```
ruby -E utf-8           # Encoding name follows -E
ruby -Eutf-8            # The space is optional
ruby --encoding utf-8   # Encoding following --encoding with a space
ruby --encoding=utf-8   # Or use an equals sign with --encoding
```

See §10.1 for complete details.

You can query the default external encoding with `Encoding.default_external`. This class method returns an `Encoding` object. Use `Encoding.locale_charmap` to obtain the name (as a string) of the character encoding derived from the locale. This method is always based on the locale setting and ignores command-line options that override the default external encoding.

2.5 Program Execution

Ruby is a scripting language. This means that Ruby programs are simply lists, or scripts, of statements to be executed. By default, these statements are executed sequentially, in the order they appear. Ruby's control structures (described in Chapter 5) alter this default execution order and allow statements to be executed conditionally or repeatedly, for example.

Programmers who are used to traditional static compiled languages like C or Java may find this slightly confusing. There is no special `main` method in Ruby from which execution begins. The Ruby interpreter is given a script of statements to execute, and it begins executing at the first line and continues to the last line.

(Actually, that last statement is not quite true. The Ruby interpreter first scans the file for `BEGIN` statements, and executes the code in their bodies. Then it goes back to line 1 and starts executing sequentially. See §5.7 for more on `BEGIN`.)

Another difference between Ruby and compiled languages has to do with module, class, and method definitions. In compiled languages, these are syntactic structures that are processed by the compiler. In Ruby, they are statements like any other. When the Ruby interpreter encounters a class definition, it executes it, causing a new class to come into existence. Similarly, when the Ruby interpreter encounters a method definition, it executes it, causing a new method to be defined. Later in the program, the interpreter will probably encounter and execute a method invocation expression for the method, and this invocation will cause the statements in the method body to be executed.

The Ruby interpreter is invoked from the command line and given a script to execute. Very simple one-line scripts are sometimes written directly on the command line. More commonly, however, the name of the file containing the script is specified. The Ruby interpreter reads the file and executes the script. It first executes any `BEGIN` blocks. Then it starts at the first line of the file and continues until one of the following happens:

- It executes a statement that causes the Ruby program to terminate.
- It reaches the end of the file.
- It reads a line that marks the logical end of the file with the token `__END__`.

Before it quits, the Ruby interpreter typically (unless the `exit!` method was called) executes the bodies of any `END` statements it has encountered and any other "shutdown hook" code registered with the `at_exit` function.

Chad Fowler *is a leading figure in the Ruby community. He's on the board of Ruby Central, Inc. He's one of the organizers of RubyConf. And he's one of the writers of RubyGems. All this makes him uniquely qualified to write this chapter.*

Chapter 17

Package Management with RubyGems

RubyGems is a standardized packaging and installation framework for libraries and applications, making it easy to locate, install, upgrade, and uninstall Ruby packages. It provides users and developers with four main facilities.

1. A standardized package format,
2. A central repository for hosting packages in this format,
3. Installation and management of multiple, simultaneously installed versions of the same library,
4. End-user tools for querying, installing, uninstalling, and otherwise manipulating these packages.

Before RubyGems came along, installing a new library involved searching the Web, downloading a package, and attempting to install it—only to find that its dependencies haven't been met. If the library you want is packaged using RubyGems, however, you can now simply ask RubyGems to install it (and all its dependencies). Everything is done for you.

In the RubyGems world, developers bundle their applications and libraries into single files called *gems*. These files conform to a standardized format, and the RubyGems system provides a command-line tool, appropriately named *gem*, for manipulating these gem files.

In this chapter, we'll see how to

1. Install RubyGems on your computer.
2. Use RubyGems to install other applications and libraries.
3. Write your own gems.

Installing RubyGems

To use RubyGems, you'll first need to download and install the RubyGems system from the project's home page at http://rubygems.rubyforge.org. After downloading and unpacking the distribution, you can install it using the included installation script.

```
% cd rubygems-0.7.0
% ruby install.rb
```

Depending on your operating system, you may need suitable privileges to write files into Ruby's site_ruby/ and bin/ directories.

The best way to test that RubyGems was installed successfully also happens to be the most important command you'll learn.

```
% gem help
RubyGems is a sophisticated package manager for Ruby.  This is
a basic help message containing pointers to more information.
  Usage:
    gem -h/--help
    gem -v/--version
    gem command [arguments...] [options...]
  Examples:
    gem install rake
    gem list --local
    gem build package.gemspec
    gem help install
  Further help:
    gem help commands            list all 'gem' commands
    gem help examples            show some examples of usage
    gem help <COMMAND>           show help on COMMAND
                                  (e.g. 'gem help install')
  Further information:
    http://rubygems.rubyforge.org
```

Because RubyGems' help is quite comprehensive, we won't go into detail about each of the available RubyGems commands and options in this chapter.

Installing Application Gems

Let's start by using RubyGems to install an application that is written in Ruby. Jim Weirich's Rake (http://rake.rubyforge.org) holds the distinction of being the first application that was available as a gem. Not only that, but it's generally a great tool to have around, as it is a build tool similar to Make and Ant. In fact, you can even use Rake to build gems!

Locating and installing Rake with RubyGems is simple.

```
% gem install -r rake
Attempting remote installation of 'Rake'
Successfully installed rake, version 0.4.3
% rake --version
rake, version 0.4.3
```

RubyGems downloads the Rake package and installs it. Because Rake is an application, RubyGems downloads both the Rake libraries and the command-line program `rake`.

You control the gem program using subcommands, each of which has its own options and help screen. In this example, we used the `install` subcommand with the `-r` option, which tells it to operate remotely. (Many RubyGems operations can be performed either locally or remotely. For example, you can use the `query` command either to display all the gems that are available remotely for installation or to display a list of gems you already have installed. For this reason, subcommands accept the options `-r` and `-l`, specifying whether an operation is meant to be carried out remotely or locally.)

If for some reason—perhaps because of a potential compatibility issue—you wanted an older version of Rake, you could use RubyGems' version requirement operators to specify criteria by which a version would be selected.

```
% gem install -r rake -v "< 0.4.3"
Attempting remote installation of 'rake'
Successfully installed rake, version 0.4.2
% rake --version
rake, version 0.4.2
```

Table 17.1 on the next page lists the version requirement operators. The `-v` argument in our previous example asks for the highest version lower than 0.4.3.

There's a subtlety when it comes to installing different versions of the same application with RubyGems. Even though RubyGems keeps separate versions of the application's library files, it does not version the actual command you use to run the application. As a result, each install of an application effectively overwrites the previous one.

During installation, you can also add the `-t` option to the RubyGems `install` command, causing RubyGems to run the gem's test suite (if one has been created). If the tests fail, the installer will prompt you to either keep or discard the gem. This is a good way to gain a little more confidence that the gem you've just downloaded works on your system the way the author intended.

```
% gem install SomePoorlyTestedProgram -t
Attempting local installation of 'SomePoorlyTestedProgram-1.0.1'
Successfully installed SomePoorlyTestedProgram, version 1.0.1
23 tests, 22 assertions, 0 failures, 1 errors...keep Gem? [Y/n] n
Successfully uninstalled SomePoorlyTestedProgram version 1.0.1
```

Had we chosen the default and kept the gem installed, we could have inspected the gem to try to determine the cause of the failing test.

Table 17.1. Version operators

Both the `require_gem` method and the `add_dependency` attribute in a `Gem::Specification` accept an argument that specifies a version dependency. RubyGems version dependencies are of the form `operator major.minor.patch_level`. Listed below is a table of all the possible version operators.

Operator	Description
=	Exact version match. Major, minor, and patch level must be identical.
!=	Any version that is not the one specified.
>	Any version that is greater (even at the patch level) than the one specified.
<	Any version that is less than the one specified.
>=	Any version greater than or equal to the specified version.
<=	Any version less than or equal to the specified version.
~>	"Boxed" version operator. Version must be greater than or equal to the specified version *and* less than the specified version after having its minor version number increased by one. This is to avoid API incompatibilities between minor version releases.

Installing and Using Gem Libraries

Using RubyGems to install a complete application was a good way to get your feet wet and to start to learn your way around the `gem` command. However, in most cases, you'll use RubyGems to install Ruby libraries for use in your own programs. Since RubyGems enables you to install and manage multiple versions of the same library, you'll also need to do some new, RubyGems-specific things when you require those libraries in your code.

Perhaps you've been asked by your mother to create a program to help her maintain and publish a diary. You have decided that you would like to publish the diary in HTML format, but you are worried that your mother may not understand all of the ins and outs of HTML markup. For this reason, you've opted to use one of the many excellent templating packages available for Ruby. After some research, you've decided on Michael Granger's BlueCloth, based on its reputation for being very simple to use.

You first need to find and install the BlueCloth gem.

```
% gem query -rn Blue
*** REMOTE GEMS ***
BlueCloth (0.0.4, 0.0.3, 0.0.2)
   BlueCloth is a Ruby implementation of Markdown, a text-to-HTML
   conversion tool for web writers. Markdown allows you to write using
   an easy-to-read, easy-to-write plain text format, then convert it
   to structurally valid XHTML (or HTML).
```

This invocation of the query command uses the -n option to search the central gem repository for any gem whose name matches the regular expression /Blue/. The results show that three available versions of BlueCloth exist (0.0.4, 0.0.3, and 0.0.2). Because you want to install the most recent one, you don't have to state an explicit version on the install command; the latest is downloaded by default.

```
% gem install -r BlueCloth
Attempting remote installation of 'BlueCloth'
Successfully installed BlueCloth, version 0.0.4
```

Generating API Documentation

Being that this is your first time using BlueCloth, you're not exactly sure how to use it. You need some API documentation to get started. Fortunately, with the addition of the --rdoc option to the install command, RubyGems will generate RDoc documentation for the gem it is installing. For more information on RDoc, see Chapter 16 on page 187.

```
% gem install -r BlueCloth --rdoc
Attempting remote installation of 'BlueCloth'
Successfully installed BlueCloth, version 0.0.4
Installing RDoc documentation for BlueCloth-0.0.4...
WARNING: Generating RDoc on .gem that may not have RDoc.
        bluecloth.rb: cc..............................
Generating HTML...
```

Having generated all this useful HTML documentation, how can you view it? You have at least two options. The hard way (though it really isn't that hard) is to open RubyGems' documentation directory and browse the documentation directly. As with most things in RubyGems, the documentation for each gem is stored in a central, protected, RubyGems-specific place. This will vary by system and by where you may explicitly choose to install your gems. The most reliable way to find the documents is to ask the gem command where your RubyGems main directory is located. For example:

```
% gem environment gemdir
/usr/local/lib/ruby/gems/1.8
```

RubyGems stores generated documentation in the doc/ subdirectory of this directory, in this case /usr/local/lib/ruby/gems/1.8/doc. You can open the file index.html and view the documentation. If you find yourself using this path often, you can create a shortcut. Here's one way to do that on Mac OS X boxes.

```
% gemdoc=`gem environment gemdir`/doc
% ls $gemdoc
BlueCloth-0.0.4
% open $gemdoc/BlueCloth-0.0.4/rdoc/index.html
```

To save time, you could declare `$gemdoc` in your login shell's profile or rc file.

The second (and easier) way to view gems' RDoc documentation is to use RubyGems' included `gem_server` utility. To start `gem_server`, simply type

```
% gem_server
[2004-07-18 11:28:51] INFO  WEBrick 1.3.1
[2004-07-18 11:28:51] INFO  ruby 1.8.2 (2004-06-29) [i386-mswin32]
[2004-07-18 11:28:51] INFO  WEBrick::HTTPServer#start: port=8808
```

`gem_server` starts a Web server running on whatever computer you run it on. By default, it will start on port 8808 and will serve gems and their documentation from the default RubyGems installation directory. Both the port and the gem directory are overridable via command-line options, using the `-p` and `-d` options, respectively.

Once you've started the `gem_server` program, if you are running it on your local computer, you can access the documentation for your installed gems by pointing your Web browser to `http://localhost:8808`. There, you will see a list of the gems you have installed with their descriptions and links to their RDoc documentation.

Let's Code!

Now you've got BlueCloth installed and you know how to use it, you're ready to write some code. Having used RubyGems to download the library, we can now also use it to load the library components into our application. Prior to RubyGems, we'd say something like

```
require 'bluecloth'
```

With RubyGems, though, we can take advantage of its packaging and versioning support. To do this, we use `require_gem` in place of `require`.

```
require 'rubygems'
require_gem 'BlueCloth', ">= 0.0.4"
doc = BlueCloth::new <<MARKUP
 This is some sample [text][1].  Just learning to use [BlueCloth][1].
 Just a simple test.

 [1]: http://ruby-lang.org
MARKUP

puts doc.to_html
```

produces:

```
<p>This is some sample <a href="http://ruby-lang.org">text</a>.  Just
 learning to use <a href="http://ruby-lang.org">BlueCloth</a>.
 Just a simple test.</p>
```

The first two lines are the RubyGems-specific code. The first line loads the RubyGems core libraries that we'll need in order to work with installed gems.

```
require 'rubygems'
```

The second line is where most of the magic happens.

```
require_gem 'BlueCloth', '>= 0.0.4'
```

This line adds the BlueCloth gem to Ruby's `$LOAD_PATH` and uses `require` to load any libraries that the gem's creator specified to be autoloaded. Let's say that again a slightly different way.

Each gem is considered to be a bundle of resources. It may contain one library file or one hundred. In an old-fashioned, non-RubyGems library, all these files would be copied into some shared location in the Ruby library tree, a location that was in Ruby's predefined load path.

RubyGems doesn't work this way. Instead, it keeps each version of each gem in its own self-contained directory tree. The gems are not injected into the standard Ruby library directories. As a result, RubyGems needs to do some fancy footwork so that you can get to these files. It does this by adding the gem's directory tree to Ruby's load path. From inside a running program, the effect is the same: `require` just works. From the outside, though, RubyGems gives you far better control over what's loaded into your Ruby programs.

In the case of BlueCloth, the templating code is distributed as one file, `bluecloth.rb`; that's the file that `require_gem` will load. `require_gem` has an optional second argument, which specifies a version requirement. In this example, you've specified that BlueCloth version 0.0.4 or greater be installed to use this code. If you had required version 0.0.5 or greater, this program would fail, because the version you've just installed is too low to meet the requirement of the program.

```
require 'rubygems'
require_gem 'BlueCloth', '>= 0.0.5'
```

produces:

```
/usr/local/lib/ruby/site_ruby/rubygems.rb:30:
          in `require_gem': (LoadError)
RubyGem version error: BlueCloth(0.0.4 not >= 0.0.5)
from prog.rb:2
```

As we said earlier, the version requirement argument is optional, and this example is obviously contrived. But, it's easy to imagine how this feature can be useful as different projects begin to depend on multiple, potentially incompatible, versions of the same library.

Dependent on RubyGems?

Astute readers (that's all of you) will have noticed that the code we've created so far is dependent on the RubyGems package being installed. In the long term, that'll be a fairly safe bet (we're guessing that RubyGems will make its way into the Ruby core distribution). For now, though, RubyGems is not part of the standard Ruby distribution,

The Code Behind the Curtain

So just what does happen behind the scenes when you call the magic `require_gem` method?

First, the gems library modifies your `$LOAD_PATH`, including any directories you have added to the gemspec's `require_paths`. Second, it calls Ruby's `require` method on any files specified in the gemspec's `autorequires` attribute (described on page 212). It's this `$LOAD_PATH`-modifying behavior that enables RubyGems to manage multiple installed versions of the same library.

so users of your software may not have RubyGems installed on their computers. If we distribute code that has `require 'rubygems'` in it, that code will fail.

You can use at least two techniques to get around this issue. First, you can wrap the RubyGems-specific code in a block and use Ruby's exception handling to rescue the resultant `LoadError` should RubyGems not be found during the `require`.

```
begin
  require 'rubygems'
  require_gem 'BlueCloth', ">= 0.0.4"
rescue LoadError
  require 'bluecloth'
end
```

This code first tries to require in the RubyGems library. If this fails, the `rescue` stanza is invoked, and your program will try to load BlueCloth using a conventional `require`. This latter require will fail if BlueCloth isn't installed, which is the same behavior users see now if they're not using RubyGems.

As of RubyGems 0.8.0, requiring `rubygems.rb` will install an overloaded version of Ruby's `require` method. Having loaded the RubyGems framework, you could say

```
require 'bluecloth'
```

Although this looks like conventional code, behind the scenes RubyGems will load `bluecloth.rb` from the first match it finds in its list of currently installed gems.

The overloaded `require` method *almost* allows you to free your applications from any RubyGems-specific code. The one exception is that the RubyGems library must be loaded before any calls to require gem-installed libraries.

To avoid RubyGems dependencies, the Ruby interpreter can be called with the -r switch

```
ruby -rubygems myprogram.rb
```

This will cause the interpreter to load the RubyGems framework, thereby installing RubyGems' overloaded version of the `require` method. To globally cause RubyGems to load with each invocation of the Ruby interpreter on a given system, you can set the `RUBYOPT` environment variable

```
% export RUBYOPT=rubygems
```

You can then run the ruby interpreter without explicitly loading the RubyGems framework, and gem-installed libraries will be available to the applications that need them.

The biggest disadvantage of using the overloaded `require` method is that you lose the ability to manage multiple installed versions of the same library. If you need a specific version of a library, it's better to use the `LoadError` method described previously.

Creating Your Own Gems

By now, you've seen how easy RubyGems makes things for the users of an application or library and are probably ready to make a gem of your own. If you're creating code to be shared with the open-source community, RubyGems are an ideal way for end-users to discover, install, and uninstall your code. They also provide a powerful way to manage internal, company projects, or even personal projects, since they make upgrades and rollbacks so simple. Ultimately, the availability of more gems makes the Ruby community stronger. These gems have to come from somewhere; we're going to show you how they can start coming from you.

Let's say you've finally gotten your mother's online diary application, MomLog, finished, and you have decided to release it under an open-source license. After all, other programmers have mothers, too. Naturally, you want to release MomLog as a gem (moms love it when you give them gems).

Package Layout

The first task in creating a gem is organizing your code into a directory structure that makes sense. The same rules that you would use in creating a typical tar or zip archive apply in package organization. Some general conventions follow.

- Put all of your Ruby source files under a subdirectory called `lib/`. Later, we'll show you how to ensure that this directory will be added to Ruby's `$LOAD_PATH` when users load this gem.
- If it's appropriate for your project, include a file under `lib/yourproject.rb` that performs the necessary `require` commands to load the bulk of the project's functionality. Before RubyGems' autorequire feature, this made things easier for others to use a library. Even with RubyGems, it makes it easier for others to explore your code if you give them an obvious starting point.

- Always include a README file including a project summary, author contact information, and pointers for getting started. Use RDoc format for this file so you can add it to the documentation that will be generated during gem installation. Remember to include a copyright and license in the README file, as many commercial users won't use a package unless the license terms are clear.
- Tests should go in a directory called test/. Many developers use a library's unit tests as a usage guide. It's nice to put them somewhere predictable, making them easy for others to find.
- Any executable scripts should go in a subdirectory called bin/.
- Source code for Ruby extensions should go in ext/.
- If you've got a great deal of documentation to include with your gem, it's good to keep it in its own subdirectory called docs/. If your README file is in the top level of your package, be sure to refer readers to this location.

This directory layout is illustrated in Figure 17.1 on page 220.

The Gem Specification

Now that you've got your files laid out as you want them, it's time to get to the heart of gem creation: the gem specification, or *gemspec*. A gemspec is a collection of metadata in Ruby or YAML (see page 737) that provides key information about your gem. The gemspec is used as input to the gem-building process. You can use several different mechanisms to create a gem, but they're all conceptually the same. Here's your first, basic MomLog gem.

```ruby
require 'rubygems'
SPEC = Gem::Specification.new do |s|
  s.name     = "MomLog"
  s.version  = "1.0.0"
  s.author   = "Jo Programmer"
  s.email    = "jo@joshost.com"
  s.homepage = "http://www.joshost.com/MomLog"
  s.platform = Gem::Platform::RUBY
  s.summary  = "An online Diary for families"
  candidates = Dir.glob("{bin,docs,lib,test}/**/*")
  s.files    = candidates.delete_if do |item|
                 item.include?("CVS") || item.include?("rdoc")
               end
  s.require_path     = "lib"
  s.autorequire      = "momlog"
  s.test_file        = "test/ts_momlog.rb"
  s.has_rdoc         = true
  s.extra_rdoc_files = ["README"]
  s.add_dependency("BlueCloth", ">= 0.0.4")
end
```

Let's quickly walk through this example. A gem's metadata is held in an object of class `Gem::Specification`. The gemspec can be expressed in either YAML or Ruby code. Here we'll show the Ruby version, as it's generally easier to construct and more flexible in use. The first five attributes in the specification give basic information such as the gem's name, the version, and the author's name, e-mail, and home page.

In this example, the next attribute is the platform on which this gem can run. In this case, the gem is a pure Ruby library with no operating system–specific requirements, so we've set the platform to `RUBY`. If this gem were written for Windows only, for example, the platform would be listed as `WIN32`. For now, this field is only informational, but in the future it will be used by the gem system for intelligent selection of precompiled native extension gems.

The gem's summary is the short description that will appear when you run a `gem query` (as in our previous BlueCloth example).

The `files` attribute is an array of pathnames to files that will be included when the gem is built. In this example, we've used `Dir.glob` to generate the list and filtered out CVS and RDoc files.

Runtime Magic

The next two attributes, `require_path` and `autorequire`, let you specify the directories that will be added to the `$LOAD_PATH` when `require_gem` loads the gem, as well as any files that will automatically be loaded using `require`. In this example, `lib` refers to a relative path under the MomLog gem directory, and the `autorequire` will cause `lib/momlog.rb` to be required when `require_gem "MomLog"` is called. RubyGems also provides `require_paths`, a plural version of `require_path`. This takes an array, allowing you to specify a number of directories to include in `$LOAD_PATH`.

Adding Tests and Documentation

The `test_file` attribute holds the relative pathname to a single Ruby file included in the gem that should be loaded as a `Test::Unit` test suite. (You can use the plural form, `test_files`, to reference an array of files containing tests.) For details on how to create a test suite, see Chapter 12 on page 143 on unit testing.

Finishing up this example, we have two attributes controlling the production of local documentation of the gem. The `has_rdoc` attribute specifies that you have added RDoc comments to your code. It's possible to run RDoc on totally uncommented code, providing a browsable view of its interfaces, but obviously this is a lot less valuable than running RDoc on well-commented code. `has_rdoc` is a way for you to tell the world, "Yes. It's worth generating the documentation for this gem."

RDoc has the convenience of being very readable in both source and rendered form, making it an excellent choice for an included README file with a package. The rdoc command normally runs only on source code files. The extra_rdoc_files attribute takes an array of paths to non-source files in your gem that you would like to be included in the generation of RDoc documentation.

Adding Dependencies

For your gem to work properly, users are going to need to have BlueCloth installed.

We saw earlier how to set a load-time version dependency for a library. Now we need to tell our gemspec about that dependency, so the installer will ensure that it is present while installing MomLog. We do that with the addition of a single method call to our Gem::Specification object.

```
s.add_dependency("BlueCloth", ">= 0.0.4")
```

The arguments to our add_dependency method are identical to those of require_gem, which we explained earlier.

After generating this gem, attempting to install it on a clean system would look something like this.

```
% gem install pkg/MomLog-1.0.0.gem
Attempting local installation of 'pkg/MomLog-1.0.0.gem'
/usr/local/lib/ruby/site_ruby/1.8/rubygems.rb:50:in `require_gem':
    (LoadError)
Could not find RubyGem BlueCloth (>= 0.0.4)
```

Because you are performing a local installation from a file, RubyGems won't attempt to resolve the dependency for you. Instead, it fails noisily, telling you that it needs BlueCloth to complete the installation. You could then install BlueCloth as we did before, and things would go smoothly the next time you attempted to install the MomLog gem.

If you had uploaded MomLog to the central RubyGems repository and then tried to install it on a clean system, you would be prompted to automatically install BlueCloth as part of the MomLog installation.

```
% gem install -r MomLog
Attempting remote installation of 'MomLog'
Install required dependency BlueCloth? [Yn]   y
Successfully installed MomLog, version 1.0.0
```

Now you've got both BlueCloth and MomLog installed, and your mother can start happily publishing her diary. Had you chosen not to install BlueCloth, the installation would have failed as it did during the local installation attempt.

As you add more features to MomLog, you may find yourself pulling in additional external gems to support those features. The add_dependency method can be called multiple times in a single gemspec, supporting as many dependencies as you need it to support.

Ruby Extension Gems

So far, all of the examples we've looked at have been pure Ruby code. However, many Ruby libraries are created as native extensions (see Chapter 21 on page 261). You have two ways to package and distribute this kind of library as a gem. You can distribute the gem in source format and have the installer compile the code at installation time. Alternatively, you can precompile the extensions and distribute one gem for each separate platform you want to support.

For source gems, RubyGems provides an additional `Gem::Specification` attribute called `extensions`. This attribute is an array of paths to Ruby files that will generate Makefiles. The most typical way to create one of these programs is to use Ruby's `mkmf` library (see Chapter 21 on page 261 and the appendix about `mkmf` on page 755). These files are conventionally named `extconf.rb`, though any name will do.

Your mom has a computerized recipe database that is near and dear to her heart. She has been storing her recipes in it for years, and you would like to give her the ability to publish these recipes on the Web for her friends and family. You discover that the recipe program, MenuBuilder, has a fairly nice native API and decide to write a Ruby extension to wrap it. Since the extension may be useful to others who aren't necessarily using MomLog, you decide to package it as a separate gem and add it as an additional dependency for MomLog.

Here's the gemspec.

```
require 'rubygems'

spec = Gem::Specification.new do |s|
  s.name = "MenuBuilder"
  s.version = "1.0.0"
  s.author = "Jo Programmer"
  s.email = "jo@joshost.com"
  s.homepage = "http://www.joshost.com/projects/MenuBuilder"
  s.platform = Gem::Platform::RUBY
  s.summary = "A Ruby wrapper for the MenuBuilder recipe database."
  s.files = ["ext/main.c", "ext/extconf.rb"]
  s.require_path = "."
  s.autorequire = "MenuBuilder"
  s.extensions = ["ext/extconf.rb"]
end

if $0 == __FILE__
  Gem::manage_gems
  Gem::Builder.new(spec).build
end
```

Note that you have to include source files in the specification's `files` list so they'll be included in the gem package for distribution.

When a source gem is installed, RubyGems runs each of its `extensions` programs and then executes the resultant Makefile.

```
% gem install MenuBuilder-1.0.0.gem
Attempting local installation of 'MenuBuilder-1.0.0.gem'
ruby extconf.rb inst MenuBuilder-1.0.0.gem
creating Makefile
make
gcc -fPIC -g -O2  -I. -I/usr/local/lib/ruby/1.8/i686-linux \
     -I/usr/local/lib/ruby/1.8/i686-linux -I.   -c main.c
gcc -shared  -L"/usr/local/lib" -o MenuBuilder.so main.o  \
     -ldl -lcrypt -lm   -lc
make install
install -c -p -m 0755 MenuBuilder.so \
     /usr/local/lib/ruby/gems/1.8/gems/MenuBuilder-1.0.0/.
Successfully installed MenuBuilder, version 1.0.0
```

RubyGems does not have the capability to detect system library dependencies that source gems may have. Should your source gems depend on a system library that is not installed, the gem installation will fail, and any error output from the make command will be displayed.

Distributing source gems obviously requires that the consumer of the gem have a working set of development tools. At a minimum, they'll need some kind of make program and a compiler. Particularly for Windows users, these tools may not be present. You can get around this limitation by distributing precompiled gems.

Creation of precompiled gems is simple—add the compiled shared object files (DLLs on Windows) to the gemspec's files list, and make sure these files are in one of the gem's require_path attributes. As with pure Ruby gems, the require_gem command will modify Ruby's $LOAD_PATH, and the shared object will be accessible via require.

Since these gems will be platform specific, you can also use the platform attribute (remember this from the first gemspec example?) to specify the target platform for the gem. The Gem::Specification class defines constants for Windows, Intel Linux, Macintosh, and pure Ruby. For platforms not included in this list, you can use the value of the RUBY_PLATFORM variable. This attribute is purely informational for now, but it's a good habit to acquire. Future RubyGems releases will use the platform attribute to intelligently select precompiled gems for the platform on which the installer is running.

Building the Gem File

The MomLog gemspec we just created is runnable as a Ruby program. Invoking it will create a gem file, MomLog-0.5.0.gem.

```
% ruby momlog.gemspec
Attempting to build gem spec 'momlog.gemspec'
Successfully built RubyGem
Name: MomLog
Version: 0.5.0
File: MomLog-0.5.0.gem
```

Alternatively, you can use the `gem build` command to generate the gem file.

```
% gem build momlog.gemspec
Attempting to build gem spec 'momlog.gemspec'
Successfully built RubyGem
Name: MomLog
Version: 0.5.0
File: MomLog-0.5.0.gem
```

Now that you've got a gem file, you can distribute it like any other package. You can put it on an FTP server or a Web site for download or e-mail it to your friends. Once your friends have got this file on their local computers (downloading from your FTP server if necessary), they can install the gem (assuming they have RubyGems installed too) by calling

```
% gem install MomLog-0.5.0.gem
Attempting local installation of 'MomLog-0.5.0.gem'
Successfully installed MomLog, version 0.5.0
```

If you would like to release your gem to the Ruby community, the easiest way is to use RubyForge (`http://rubyforge.org`). RubyForge is an open-source project management Web site. It also hosts the central gem repository. Any gem files released using RubyForge's file release feature will be automatically picked up and added to the central gem repository several times each day. The advantage to potential users of your software is that it will be available via RubyGems' remote query and installation operations, making installation even easier.

Building with Rake

Last but certainly not least, we can use Rake to build gems (remember Rake, the pure-Ruby build tool we mentioned back on page 204). Rake uses a command file called a `Rakefile` to control the build. This defines (in Ruby syntax!) a set of *rules* and *tasks*. The intersection of make's rule-driven concepts and Ruby's power make for a build and release automator's dream environment. And, what release of a Ruby project would be complete without the generation of a gem?

For details on how to use Rake, see `http://rake.rubyforge.org`. Its documents are comprehensive and always up-to-date. Here, we'll focus on just enough Rake to build a gem. From the Rake documentation:

Tasks are the main unit of work in a Rakefile. Tasks have a name (usually given as a symbol or a string), a list of prerequisites (more symbols or strings), and a list of actions (given as a block).

Normally, you can use Rake's built-in `task` method to define your own named tasks in your Rakefile. For special cases, it makes sense to provide helper code to automate some of the repetitive work you would have to do otherwise. Gem creation is one of these special cases. Rake comes with a special *TaskLib*, called GemPackageTask, that helps integrate gem creation into the rest of your automated build and release process.

To use GemPackageTask in your Rakefile, create the gemspec exactly as we did previously, but this time place it into your Rakefile. We then feed this specification to GemPackageTask.

```
require 'rubygems'
Gem::manage_gems
require 'rake/gempackagetask'
spec = Gem::Specification.new do |s|
  s.name     = "MomLog"
  s.version  = "0.5.0"
  s.author   = "Jo Programmer"
  s.email    = "jo@joshost.com"
  s.homepage = "http://www.joshost.com/MomLog"
  s.platform = Gem::Platform::RUBY
  s.summary  = "An online Diary for families"
  s.files = FileList["{bin,tests,lib,docs}/**/*"].exclude("rdoc").to_a
  s.require_path     = "lib"
  s.autorequire      = "momlog"
  s.test_file        = "tests/ts_momlog.rb"
  s.has_rdoc         = true
  s.extra_rdoc_files = ["README"]
  s.add_dependency("BlueCloth", ">= 0.0.4")
  s.add_dependency("MenuBuilder", ">= 1.0.0")
end
Rake::GemPackageTask.new(spec) do |pkg|
    pkg.need_tar = true
end
```

Note that you'll have to require the `rubygems` package into your Rakefile. You'll also notice that we've used Rake's `FileList` class instead of `Dir.glob` to build the list of files. `FileList` is smarter than `Dir.glob` for this purpose in that it automatically ignores commonly unused files (such as the CVS directory that the CVS version control tool leaves lying around).

Internally, the GemPackageTask generates a Rake target with the identifier

```
package_directory/gemname-gemversion.gem
```

In our case, this identifier will be `pkg/MomLog-0.5.0.gem`. You can invoke this task from the same directory where you've put the Rakefile.

```
% rake pkg/MomLog-0.5.0.gem
(in /home/chad/download/gembook/code/MomLog)
  Successfully built RubyGem
  Name: MomLog
  Version: 0.5.0
  File: MomLog-0.5.0.gem
```

Now that you've got a task, you can use it like any other Rake task, adding dependencies to it or adding it to the dependency list of another task, such as deployment or release packaging.

Maintaining Your Gem (and One Last Look at MomLog)

You've released MomLog, and it's attracting new, adoring users every week. You have taken great care to package it cleanly and are using Rake to build your gem.

Your gem being "in the wild" with your contact information attached to it, you know that it's only a matter of time before you start receiving feature requests (and fan mail!) from your users. But, your first request comes via a phone call from none other than dear old Mom. She has just gotten back from a vacation in Florida and asks you how she can include her vacation pictures in her diary. You don't think an explanation of command-line FTP would be time well spent, and being the ever-devoted son or daughter, you spend your evening coding a nice photo album module for MomLog.

Since you have added functionality to the application (as opposed to just fixing a bug), you decide to increase MomLog's version number from 1.0.0 to 1.1.0. You also add a set of tests for the new functionality and a document about how to set up the photo upload functionality.

Figure 17.1 on the following page shows the complete directory structure of your final MomLog 1.1.0 package. The final gem specification (extracted from the Rakefile) looks like this.

```
spec = Gem::Specification.new do |s|
  s.name     = "MomLog"
  s.version  = "1.1.0"
  s.author   = "Jo Programmer"
  s.email    = "jo@joshost.com"
  s.homepage = "http://www.joshost.com/MomLog"
  s.platform = Gem::Platform::RUBY
  s.summary  = "An online diary, recipe publisher, " +
                "and photo album for families."
  s.files = FileList["{bin,tests,lib,docs}/**/*"].exclude("rdoc").to_a
  s.require_path     = "lib"
  s.autorequire      = "momlog"
  s.test_file        = "tests/ts_momlog.rb"
  s.has_rdoc         = true
  s.extra_rdoc_files = ["README", "docs/DatabaseConfiguration.rdoc",
                "docs/Installing.rdoc", "docs/PhotoAlbumSetup.rdoc"]
  s.add_dependency("BlueCloth", ">= 0.0.4")
  s.add_dependency("MenuBuilder", ">= 1.0.0")
end
```

Figure 17.1. MomLog package structure

```
momlog/
 ├── README
 ├── Rakefile
 ├── bin/
 │    └── momlog_server
 ├── docs/
 │    ├── Installing.rdoc
 │    ├── DatabaseConfiguration.rdoc
 │    └── PhotoAlbumSetup.rdoc
 ├── lib/
 │    ├── momlog.rb
 │    └── momlog/
 │         ├── diary.rb
 │         ├── recipes.rb
 │         ├── db.rb
 │         ├── upload.rb
 │         ├── photo_album.rb
 │         └── rss.rb
 └── tests/
      ├── ts_momlog.rb
      ├── tc_recipe.rb
      ├── tc_photo_album.rb
      ├── tc_upload.rb
      ├── tc_diary.rb
      └── tc_rss.rb
```

You run Rake over your Rakefile, generating the updated MomLog gem, and you're ready to release the new version. You log into your RubyForge account, and upload your gem to the "Files" section of your project. While you wait for RubyGems' automated process to release the gem into the central gem repository, you type a release announcement to post to your RubyForge project.

Within about an hour, you log in to your mother's Web server to install the new software for her. RubyGems makes things easy, but we have to take special care of Mom.

```
% gem query -rn MomLog

*** REMOTE GEMS ***

MomLog (1.1.0, 1.0.0)
    An online diary, recipe publisher, and photo album for families.
```

Great! The query indicates that there are two versions of MomLog available now. You type the `install` command without specifying a version argument, because you know that the default is to install the most recent version.

```
% gem install -r MomLog
Attempting remote installation of 'MomLog'
Successfully installed MomLog, version 1.1.0
```

You haven't changed any of the dependencies for MomLog, so your existing BlueCloth and MenuBuilder installations meet the requirements for MomLog 1.1.0.

Now that Mom's happy, it's time to go try some of her recently posted recipes.

This documentation displayed by *ri* is extracted from specially formatted comments in Ruby source code. See §2.1.1.2 for details.

1.2.5 Ruby Package Management with gem

Ruby's package management system is known as RubyGems, and packages or modules distributed using RubyGems are called "gems." RubyGems makes it easy to install Ruby software and can automatically manage complex dependencies between packages.

The frontend script for RubyGems is *gem*, and it's distributed with Ruby 1.9 just as *irb* and *ri* are. In Ruby 1.8, you must install it separately—see *http://rubygems.org*. Once the *gem* program is installed, you might use it like this:

```
# gem install rails
Successfully installed activesupport-1.4.4
Successfully installed activerecord-1.15.5
Successfully installed actionpack-1.13.5
Successfully installed actionmailer-1.3.5
Successfully installed actionwebservice-1.2.5
Successfully installed rails-1.2.5
6 gems installed
Installing ri documentation for activesupport-1.4.4...
Installing ri documentation for activerecord-1.15.5...
...etc...
```

As you can see, the `gem install` command installs the most recent version of the gem you request and also installs any gems that the requested gem requires. *gem* has other useful subcommands as well. Some examples:

```
gem list               # List installed gems
gem enviroment         # Display RubyGems configuration information
gem update rails       # Update a named gem
gem update             # Update all installed gems
gem update --system    # Update RubyGems itself
gem uninstall rails    # Remove an installed gem
```

In Ruby 1.8, the gems you install cannot be automatically loaded by Ruby's `require` method. (See §7.6 for more about loading modules of Ruby code with the `require` method.) If you're writing a program that will be using modules installed as gems, you must first require the `rubygems` module. Some Ruby 1.8 distributions are preconfigured with the `RubyGems` library, but you may need to download and install this manually. Loading this `rubygems` module alters the `require` method itself so that it searches the set of installed gems before it searches the standard library. You can also automatically enable RubyGems support by running Ruby with the `-rubygems` command-line option. And if you add `-rubygems` to the `RUBYOPT` environment variable, then the `RubyGems` library will be loaded on every invocation of Ruby.

The `rubygems` module is part of the standard library in Ruby 1.9, but it is no longer required to load gems. Ruby 1.9 knows how to find installed gems on its own, and you do not have to put `require 'rubygems'` in your programs that use gems.

When you load a gem with `require` (in either 1.8 or 1.9), it loads the most recent installed version of the gem you specify. If you have more specific version requirements, you can use the `gem` method before calling `require`. This finds a version of the gem matching the version constraints you specify and "activates" it, so that a subsequent `require` will load that version:

```
require 'rubygems'               # Not necessary in Ruby 1.9
gem 'RedCloth', '> 2.0', '< 4.0' # Activate RedCloth version 2.x or 3.x
require 'RedCloth'               # And now load it
```

You'll find more about `require` and gems in §7.6.1. Complete coverage of RubyGems, the *gem* program, and the `rubygems` module are beyond the scope of this book. The `gem` command is self-documenting—start by running `gem help`. For details on the `gem` method, try `ri gem`. And for complete details, see the documentation at *http://rubygems.org*.

1.2.6 More Ruby Tutorials

This chapter began with a tutorial introduction to the Ruby language. You can try out the code snippets of that tutorial using *irb*. If you want more tutorials before diving into the language more formally, there are two good ones available by following links on the *http://www.ruby-lang.org* home page. One *irb*-based tutorial is called "Ruby in Twenty Minutes."[*] Another tutorial, called "Try Ruby!", is interesting because it works in your web browser and does not require you to have Ruby or *irb* installed on your system.[†]

1.2.7 Ruby Resources

The Ruby web site (*http://www.ruby-lang.org*) is the place to find links to other Ruby resources, such as online documentation, libraries, mailing lists, blogs, IRC channels, user groups, and conferences. Try the "Documentation," "Libraries," and "Community" links on the home page.

1.3 About This Book

As its title implies, this book covers the Ruby programming language and aspires to do so comprehensively and accessibly. This edition of the book covers language versions 1.8 and 1.9. Ruby blurs the distinction between language and platform, and so our coverage of the language includes a detailed overview of the core Ruby API. But this book is not an API reference and does not cover the core classes comprehensively. Also,

[*] At the time of this writing, the direct URL for this tutorial is *http://www.ruby-lang.org/en/documentation/quickstart/*.

[†] If you can't find the "Try Ruby!" link on the Ruby home page, try this URL: *http://tryruby.hobix.com*.

Chapter 18

Ruby and the Web

Ruby is no stranger to the Internet. Not only can you write your own SMTP server, FTP daemon, or Web server in Ruby, but you can also use Ruby for more usual tasks such as CGI programming or as a replacement for PHP.

Many options are available for using Ruby to implement Web applications, and a single chapter can't do them all justice. Instead, we'll try to touch some of the highlights and point you toward libraries and resources that can help.

Let's start with some simple stuff: running Ruby programs as Common Gateway Interface (CGI) programs.

Writing CGI Scripts

You can use Ruby to write CGI scripts quite easily. To have a Ruby script generate HTML output, all you need is something like

```
#!/usr/bin/ruby
print "Content-type: text/html\r\n\r\n"
print "<html><body>Hello World! It's #{Time.now}</body></html>\r\n"
```

Put this script in a CGI directory, mark it as executable, and you'll be able to access it via your browser. (If your Web server doesn't automatically add headers, you'll need to add the response header yourself.)

```
#!/usr/bin/ruby
print "HTTP/1.0 200 OK\r\n"
print "Content-type: text/html\r\n\r\n"
print "<html><body>Hello World! It's #{Time.now}</body></html>\r\n"
```

However, that's hacking around at a pretty low level. You'd need to write your own request parsing, session management, cookie manipulation, output escaping, and so on. Fortunately, options are available to make this easier.

Using cgi.rb

Class `CGI` provides support for writing CGI scripts. With it, you can manipulate forms, cookies, and the environment; maintain stateful sessions; and so on. It's a fairly large class, but we'll take a quick look at its capabilities here.

Quoting

When dealing with URLs and HTML code, you must be careful to quote certain characters. For instance, a slash character (`/`) has special meaning in a URL, so it must be "escaped" if it's not part of the pathname. That is, any `/` in the query portion of the URL will be translated to the string `%2F` and must be translated back to a `/` for you to use it. Space and ampersand are also special characters. To handle this, `CGI` provides the routines `CGI.escape` and `CGI.unescape`.

```
require 'cgi'
puts CGI.escape("Nicholas Payton/Trumpet & Flugel Horn")
```

produces:

```
Nicholas+Payton%2FTrumpet+%26+Flugel+Horn
```

More frequently, you may want to escape HTML special characters.

```
require 'cgi'
puts CGI.escapeHTML("a < 100 && b > 200")
```

produces:

```
a &lt; 100 &amp;&amp; b &gt; 200
```

To get really fancy, you can decide to escape only certain HTML elements within a string.

```
require 'cgi'
puts CGI.escapeElement('<hr><a href="/mp3">Click Here</a><br>','A')
```

produces:

```
<hr>&lt;a href=&quot;/mp3&quot;&gt;Click Here&lt;/a&gt;<br>
```

Here only the `A` element is escaped; other elements are left alone. Each of these methods has an "un-" version to restore the original string.

```
require 'cgi'
puts CGI.unescapeHTML("a &lt; 100 &amp;&amp; b &gt; 200")
```

produces:

```
a < 100 && b > 200
```

Query Parameters

HTTP requests from the browser to your application may contain parameters, either passed as part of the URL or passed as data embedded in the body of the request.

Processing of these parameters is complicated by the fact that a value with a given name may be returned multiple times in the same request. For example, say we're writing a survey to find out why folks like Ruby. The HTML for our form looks like this.

```
<html>
  <head><title>Test Form</title></head>
  <body>
    I like Ruby because:
    <form target="cgi-bin/survey.rb">
      <input type="checkbox" name="reason" value="flexible" />
         It's flexible<br />
      <input type="checkbox" name="reason" value="transparent" />
         It's transparent<br />
      <input type="checkbox" name="reason" value="perlish" />
         It's like Perl<br />
      <input type="checkbox" name="reason" value="fun" />
         It's fun
      <p>
        Your name: <input type="text" name="name">
      </p>
      <input type="submit"/>
    </form>
  </body>
</html>
```

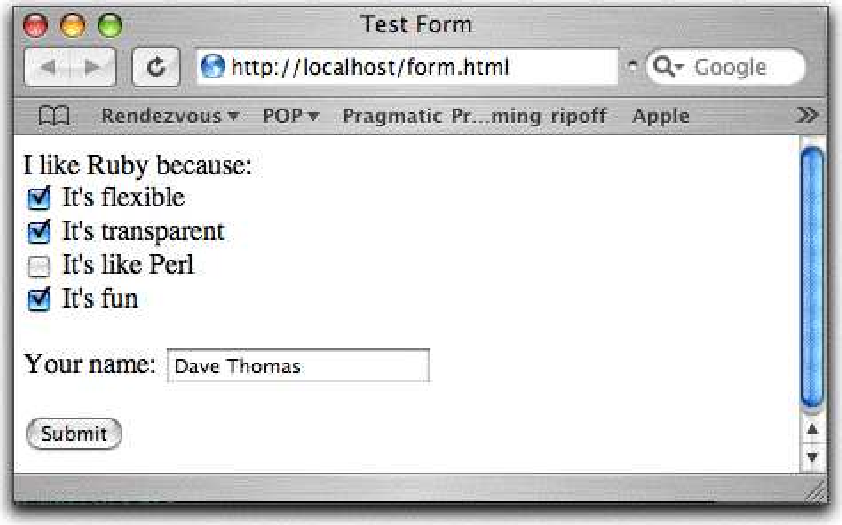
When someone fills in this form, they might check multiple reasons for liking Ruby (as shown in Figure 18.1 on the next page). In this case, the form data corresponding to the name `reason` will have three values, corresponding to the three checked boxes.

Class `CGI` gives you access to form data in a couple of ways. First, we can just treat the CGI object as a hash, indexing it with field names and getting back field values.

```
require 'cgi'
cgi = CGI.new
cgi['name']     →   "Dave Thomas"
cgi['reason']   →   "flexible"
```

1.8 However, this doesn't work well with the `reason` field: we see only one of the three values. We can ask to see them all by using the `CGI#params` method. The value returned by `params` acts like a hash containing the request parameters. You can both read and write this hash (the latter allows you to modify the data associated with a request). Note that each of the values in the hash is actually an array.

Figure 18.1. Sample CGI Form


Test Form
http://localhost/form.html
Google
Rendezvous ▾ POP ▾ Pragmatic Pr...ming ripoff Apple
Submit

I like Ruby because:
- [✓] It's flexible
- [✓] It's transparent
- [ ] It's like Perl
- [✓] It's fun

Your name: Dave Thomas

```
require 'cgi'
cgi = CGI.new
cgi.params                 →   {"name"=>["Dave Thomas"],
                               "reason"=>["flexible", "transparent",
                               "fun"]}
cgi.params['name']         →   ["Dave Thomas"]
cgi.params['reason']       →   ["flexible", "transparent", "fun"]
cgi.params['name'] = [ cgi['name'].upcase ]
cgi.params                 →   {"name"=>["DAVE THOMAS"],
                               "reason"=>["flexible", "transparent",
                               "fun"]}
```

You can determine if a particular parameter is present in a request using CGI#has_key?.

```
require 'cgi'
cgi = CGI.new
cgi.has_key?('name')       →   true
cgi.has_key?('age')        →   false
```

Generating HTML

CGI contains a huge number of methods that can be used to create HTML—one method per element. To enable these methods, you must create a CGI object by calling CGI.new, passing in the required level of HTML. In these examples, we'll use html3.

To make element nesting easier, these methods take their content as code blocks. The code blocks should return a String, which will be used as the content for the element. For this example, we've added some gratuitous newlines to make the output fit on the page.

```
require 'cgi'
cgi = CGI.new("html3")  # add HTML generation methods
cgi.out {
  cgi.html {
    cgi.head { "\n"+cgi.title{"This Is a Test"} } +
    cgi.body { "\n"+
      cgi.form {"\n"+
        cgi.hr +
        cgi.h1 { "A Form: " } + "\n"+
        cgi.textarea("get_text") +"\n"+
        cgi.br +
        cgi.submit
      }
    }
  }
}
```

produces:

```
Content-Type: text/html
Content-Length: 302

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><HTML><HEAD>
<TITLE>This Is a Test</TITLE></HEAD><BODY>
<FORM METHOD="post" ENCTYPE="application/x-www-form-urlencoded">
<HR><H1>A Form: </H1>
<TEXTAREA NAME="get_text" ROWS="10" COLS="70"></TEXTAREA>
<BR><INPUT TYPE="submit"></FORM></BODY></HTML>
```

This code will produce an HTML form titled "This Is a Test," followed by a horizontal rule, a level-one header, a text input area, and finally a submit button. When the submit comes back, you'll have a CGI parameter named get_text containing the text the user entered.

Although quite interesting, this method of generating HTML is fairly laborious and probably isn't used much in practice. Most people seem to write the HTML directly, use a templating system, or use an application framework, such as Iowa. Unfortunately, we don't have space here to discuss Iowa—have a look at the online documentation at http://enigo.com/projects/iowa, or look at Chapter 6 of *The Ruby Developer's Guide* [FJN02]—but we can look at templating.

Templating Systems

Templating systems let you separate the presentation and logic of your application. It seems that just about everyone who writes a Web application using Ruby at some point also writes a templating system: the RubyGarden wiki lists quite a few,[1] and even this list isn't complete. For now, let's just look at three: RDoc templates, Amrita, and erb/eruby.

RDoc Templates

The RDoc documentation system (described in Chapter 16 on page 187) includes a very simple templating system that it uses to generate all its XML and HTML output. Because RDoc is distributed as part of standard Ruby, the templating system is available wherever Ruby 1.8.2 or later is installed. However, the templating system does not use conventional HTML or XML markup (as it is intended to be used to generate output in many different formats), so files marked up with RDoc templates may not be easy to edit using conventional HTML editing tools.

```
require 'rdoc/template'
HTML = %{Hello, %name%.
<p>
The reasons you gave were:
<ul>
START:reasons
    <li>%reason_name% (%rank%)
END:reasons
</ul>
}
data = {
  'name' => 'Dave Thomas',
  'reasons' => [
    { 'reason_name' => 'flexible',    'rank' => '87' },
    { 'reason_name' => 'transparent', 'rank' => '76' },
    { 'reason_name' => 'fun',         'rank' => '94' },
  ]
}

t = TemplatePage.new(HTML)
t.write_html_on(STDOUT, data)
```

produces:

```
Hello, Dave Thomas.
<p>
The reasons you gave were:
<ul>
    <li>flexible (87)
    <li>transparent (76)
    <li>fun (94)
</ul>
```

1. http://www.rubygarden.org/ruby?HtmlTemplates

The constructor is passed a string containing the template to be used. The method write_html_on is then passed a hash containing names and values. If the template contains the sequence %xxxx%, the hash is consulted, and the value corresponding to the name xxx is substituted in. If the template contains START:yyy, the hash value corresponding to yyy is assumed to be an array of hashes. The template lines between START:yyy and END:yyy are repeated for each element in that array. The templates also support conditions: lines between IF:zzz and ENDIF:zzz are included in the output only if the hash has a key zzz.

Amrita

Amrita[2] is a library that generates HTML documents from a template that is itself valid HTML. This makes Amrita easy to use with existing HTML editors. It also means that Amrita templates display correctly as freestanding HTML pages.

Amrita uses the id tags in HTML elements to determine the values to be substituted. If the value corresponding to a given name is nil or false, the HTML element won't be included in the resulting output. If the value is an array, it iterates the corresponding HTML element.

```
require 'amrita/template'
include Amrita
HTML = %{<p id="greeting" />
<p>The reasons you gave were:</p>
<ul>
    <li id="reasons"><span id="reason_name"></span>,
                     <span id="rank"></span>
</ul>
}
data = {
  :greeting => 'Hello, Dave Thomas',
  :reasons  => [
    { :reason_name => 'flexible',    :rank => '87' },
    { :reason_name => 'transparent', :rank => '76' },
    { :reason_name => 'fun',         :rank => '94' },
  ]
}
t = TemplateText.new(HTML)
t.prettyprint = true
t.expand(STDOUT, data)
```

produces:

```
<p>Hello, Dave Thomas</p>
<p>The reasons you gave were:</p>
```

2. http://www.brain-tokyo.jp/research/amrita/rdocs/

```
<ul>
  <li>flexible, 87 </li>
  <li>transparent, 76 </li>
  <li>fun, 94 </li>
</ul>
```

erb and eruby

So far we've looked at using Ruby to create HTML output, but we can turn the problem inside out; we can actually embed Ruby in an HTML document.

A number of packages allow you to embed Ruby statements in some other sort of a document, especially in an HTML page. Generically, this is known as "eRuby." Specifically, several different implementations of eRuby exist, including `eruby` and `erb`. `eruby`, written by Shugo Maeda, is available for download from the Ruby Application Archive. `erb`, its little cousin, is written in pure Ruby and is included with the standard distribution. We'll look at `erb` here.

Embedding Ruby in HTML is a very powerful concept—it basically gives us the equivalent of a tool such as ASP, JSP, or PHP, but with the full power of Ruby.

Using erb

`erb` is normally used as a filter. Text within the input file is passed through untouched, with the following exceptions

Expression	**Description**
<% *ruby code* %>	Execute the Ruby code between the delimiters.
<%= *ruby expression* %>	Evaluate the Ruby expression, and replace the sequence with the expression's value.
<%# *ruby code* %>	The Ruby code between the delimiters is ignored (useful for testing).
`%` *`line of ruby code`*	A line that starts with a percent is assumed to contain just Ruby code.

You invoke `erb` as

```
erb [ options ] [ document ]
```

If the *document* is omitted, `eruby` will read from standard input. The command-line options for `erb` are shown in Table 18.1 on the next page.

Let's look at some simple examples. We'll run the `erb` executable on the following input.

```
% a = 99
<%= a %> bottles of beer...
```

Table 18.1. Command-line options for `erb`

Option	Description
`-d`	Sets $DEBUG to `true`.
`-K`*kcode*	Specifies an alternate encoding system (see page 169).
`-n`	Display resulting Ruby script (with line numbers).
`-r` *library*	Loads the named *library*.
`-P`	Doesn't do `erb` processing on lines starting `%`.
`-S` *level*	Sets the *safe level*.
`-T` *mode*	Sets the *trim mode*.
`-v`	Enables verbose mode.
`-x`	Displays resulting Ruby script.

The line starting with the percent sign simply executes the given Ruby statement. The next line contains the sequence `<%= a %>`, which substitutes in the value of `a`.

```
erb f1.erb
```

produces:

```
99 bottles of beer...
```

`erb` works by rewriting its input as a Ruby script and then executing that script. You can see the Ruby that `erb` generates using the `-n` or `-x` option.

```
erb -x f1.erb
```

produces:

```
_erbout = '';  a = 99
_erbout.concat(( a ).to_s); _erbout.concat " bottles of beer...\n"
_erbout
```

Notice how `erb` builds a string, `_erbout`, containing both the static strings from the template and the results of executing expressions (in this case the value of `a`).

Of course, you can embed Ruby within a more complex document type, such as HTML. Figure 18.2 on page 232 shows a couple of loops in an HTML document.

Installing eruby in Apache

If you want to use `erb`-like page generation for a Web site that gets a reasonable amount of traffic, you'll probably want to switch across to using `eruby`, which has better performance. You can then configure the Apache Web server to automatically parse Ruby-embedded documents using eRuby, much in the same way that PHP does. You create Ruby-embedded files with an `.rhtml` suffix and configure the Web server to run the `eruby` executable on these documents to produce the desired HTML output.

To use eruby with the Apache Web server, you need to perform the following steps.

1. Copy the eruby binary to the cgi-bin directory.
2. Add the following two lines to httpd.conf.

   ```
   AddType application/x-httpd-eruby .rhtml
   Action application/x-httpd-eruby /cgi-bin/eruby
   ```

3. If desired, you can also add or replace the DirectoryIndex directive such that it includes index.rhtml. This lets you use Ruby to create directory listings for directories that do not contain an index.html. For instance, the following directive would cause the embedded Ruby script index.rhtml to be searched for and served if neither index.html nor index.shtml existed in a directory.

   ```
   DirectoryIndex index.html index.shtml index.rhtml
   ```

   Of course, you could also simply use a sitewide Ruby script as well.

   ```
   DirectoryIndex index.html index.shtml /cgi-bin/index.rb
   ```

Cookies

Cookies are a way of letting Web applications store their state on the user's machine. Frowned upon by some, cookies are still a convenient (if unreliable) way of remembering session information.

The Ruby CGI class handles the loading and saving of cookies for you. You can access the cookies associated with the current request using the CGI#cookies method, and you can set cookies back into the browser by setting the cookies parameter of CGI#out to reference either a single cookie or an array of cookies.

```
#!/usr/bin/ruby
COOKIE_NAME = 'chocolate chip'
require 'cgi'
cgi = CGI.new
values = cgi.cookies[COOKIE_NAME]
if values.empty?
  msg = "It looks as if you haven't visited recently"
else
  msg = "You last visited #{values[0]}"
end
cookie = CGI::Cookie.new(COOKIE_NAME, Time.now.to_s)
cookie.expires = Time.now + 30*24*3600 # 30 days
cgi.out("cookie" => cookie ) { msg }
```

Figure 18.2. Erb processing a file with loops

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
<head>
<title>eruby example</title>
</head>
<body>
<h1>Enumeration</h1>
<ul>
%5.times do |i|
  <li>number <%=i%></li>
%end
</ul>
<h1>"Environment variables starting with "T"</h1>
<table>
%ENV.keys.grep(/^T/).each do |key|
  <tr><td><%=key%></td><td><%=ENV[key]%></td></tr>
%end
</table>
</body>
</html>
```

produces:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
<head>
<title>eruby example</title>
</head>
<body>
<h1>Enumeration</h1>
<ul>
  <li>number 0</li>
  <li>number 1</li>
  <li>number 2</li>
  <li>number 3</li>
  <li>number 4</li>
</ul>
<h1>"Environment variables starting with "T"</h1>
<table>
  <tr><td>TERM_PROGRAM</td><td>Apple_Terminal</td></tr>
  <tr><td>TERM</td><td>xterm-color</td></tr>
  <tr><td>TERM_PROGRAM_VERSION</td><td>133</td></tr>
  <tr><td>TYPE</td><td>SCREEN</td></tr>
</table>
</body>
</html>
```

Sessions

Cookies by themselves still need a bit of work to be useful. We really want *session:* information that persists between requests from a particular Web browser. Sessions are handled by class CGI::Session, which uses cookies but provides a higher-level abstraction.

As with cookies, sessions emulate a hashlike behavior, letting you associate values with keys. Unlike cookies, sessions store the majority of their data on the server, using the browser-resident cookie simply as a way of uniquely identifying the server-side data. Sessions also give you a choice of storage techniques for this data: it can be held in regular files, in a PStore (see the description on page 698), in memory, or even in your own customized store.

Sessions should be closed after use, as this ensures that their data is written out to the store. When you've permanently finished with a session, you should delete it.

```
require 'cgi'
require 'cgi/session'
cgi = CGI.new("html3")
sess = CGI::Session.new(cgi,
                        "session_key" => "rubyweb",
                        "prefix" => "web-session."
                        )
if sess['lastaccess']
  msg = "You were last here #{sess['lastaccess']}."
else
  msg = "Looks like you haven't been here for a while"
end
count = (sess["accesscount"] || 0).to_i
count += 1
msg << "<p>Number of visits: #{count}"
sess["accesscount"] = count
sess["lastaccess"]  = Time.now.to_s
sess.close
cgi.out {
  cgi.html {
    cgi.body {
      msg
    }
  }
}
```

The code in the previous example used the default storage mechanism for sessions: persistent data was stored in files in your default temporary directory (see Dir.tmpdir). 1.8 The filenames will all start web-session. and will end with a hashed version of the session number. See ri CGI::Session for more information.

Improving Performance

You can use Ruby to write CGI programs for the Web, but, as with most CGI programs, the default configuration has to start a new copy of Ruby with every cgi-bin page access. That's expensive in terms of machine utilization and can be painfully slow for Web surfers. The Apache Web server solves this problem by supporting loadable *modules*.

Typically, these modules are dynamically loaded and become part of the running Web server process—you have no need to spawn another interpreter over and over again to service requests; the Web server *is* the interpreter.

And so we come to `mod_ruby` (available from the archives), an Apache module that links a full Ruby interpreter into the Apache Web server itself. The `README` file included with `mod_ruby` provides details on how to compile and install it.

Once installed and configured, you can run Ruby scripts pretty much as you could without `mod_ruby`, except that now they will come up much faster. You can also take advantage of the extra facilities that `mod_ruby` provides (such as tight integration into Apache's request handling).

You have some things to watch, however. Because the interpreter remains in memory between requests, it may end up handling requests from multiple applications. It's possible for libraries in these applications to clash (particularly if different libraries contain classes with the same name). You also cannot assume that the same interpreter will handle the series of requests from one browser's session—Apache will allocate handler processes using its internal algorithms.

Some of these issues are resolved using the FastCGI protocol. This is an interesting hack, available to all CGI-style programs, not just Ruby. It uses a very small proxy program, typically running as an Apache module. When requests are received, this proxy then forwards them to a particular long-running process that acts like a normal CGI script. The results are fed back to the proxy, and then back to the browser. FastCGI has the same advantages as running `mod_ruby`, as the interpreter is always running in the background. It also gives you more control over how requests are allocated to interpreters. You'll find more information at `http://www.fastcgi.com`.

Choice of Web Servers

1.8 So far, we've been running Ruby scripts under the control of the Apache Web server. However, Ruby 1.8 and later comes bundled with WEBrick, a flexible, pure-Ruby HTTP server toolkit. Basically, it's an extensible plug in–based framework that lets you write servers to handle HTTP requests and responses. Here's a basic HTTP server that serves documents and directory indexes.

```
#!/usr/bin/ruby
require 'webrick'
include WEBrick

s = HTTPServer.new(
  :Port         => 2000,
  :DocumentRoot => File.join(Dir.pwd, "/html")
)

trap("INT") { s.shutdown }

s.start
```

The HTTPServer constructor creates a new Web server on port 2000. The code sets the document root to be the html/ subdirectory of the current directory. It then uses Kernel.trap to arrange to shut down tidily on interrupts before starting the server running. If you point your browser at http://localhost:2000, you should see a listing of your html subdirectory.

WEBrick can do far more than serve static content. You can use it just like a Java servlet container. The following code mounts a simple servlet at the location /hello. As requests arrive, the do_GET method is invoked. It uses the response object to display the user agent information and parameters from the request.

```
#!/usr/bin/ruby

require 'webrick'
include WEBrick

s = HTTPServer.new( :Port => 2000 )

class HelloServlet < HTTPServlet::AbstractServlet
  def do_GET(req, res)
    res['Content-Type'] = "text/html"
    res.body = %{
      <html><body>
        Hello. You're calling from a #{req['User-Agent']}
       <p>
        I see parameters: #{req.query.keys.join(', ')}
      </body></html>
    }
  end
end

s.mount("/hello", HelloServlet)

trap("INT"){ s.shutdown }

s.start
```

More information on WEBrick is available from http:///www.webrick.org. There you'll find links to a set of useful servlets, including one that lets you write SOAP servers in Ruby.

SOAP and Web Services

1.8 Speaking of SOAP, Ruby now comes with an implementation of SOAP.[3] This lets you write both servers and clients using Web services. By their nature, these applications can operate both locally and remotely across a network. SOAP applications are also unaware of the implementation language of their network peers, so SOAP is a convenient way of interconnecting Ruby applications with those written in languages such as Java, Visual Basic, or C++.

SOAP is basically a marshaling mechanism which uses XML to send data between two nodes in a network. It is typically used to implement remote procedure calls, RPCs, between distributed processes. A SOAP server publishes one or more interfaces. These interfaces are defined in terms of data types and methods that use those types. SOAP clients then create local proxies that SOAP connects to interfaces on the server. A call to a method on the proxy is then passed to the corresponding interface on the server. Return values generated by the method on the server are passed back to the client via the proxy.

Let's start with a trivial SOAP service. We'll write an object that does interest calculations. Initially, it offers a single method, `compound`, that determines compound interest given a principal, an interest rate, the number of times interest is compounded per year, and the number of years. For management purposes, we'll also keep track of how many times this method was called and make that count available via an accessor. Note that this class is just regular Ruby code—it doesn't know that it's running in a SOAP environment.

```
class InterestCalculator
  attr_reader :call_count
  def initialize
    @call_count = 0
  end
  def compound(principal, rate, freq, years)
    @call_count += 1
    principal*(1.0 + rate/freq)**(freq*years)
  end
end
```

Now we'll make an object of this class available via a SOAP server. This will enable client applications to call the object's methods over the network. We're using the standalone server here, which is convenient when testing, as we can run it from the command line. You can also run Ruby SOAP servers as CGI scripts or under `mod_ruby`.

3. SOAP once stood for Simple Object Access Protocol. When folks could no longer stand the irony, the acronym was dropped, and now SOAP is just a name.

```
require 'soap/rpc/standaloneServer'
require 'interestcalc'
NS = 'http://pragprog.com/InterestCalc'
class Server2 < SOAP::RPC::StandaloneServer
  def on_init
    calc = InterestCalculator.new
    add_method(calc, 'compound', 'principal', 'rate', 'freq', 'years')
    add_method(calc, 'call_count')
  end
end
svr = Server2.new('Calc', NS, '0.0.0.0', 12321)
trap('INT') { svr.shutdown }
svr.start
```

This code defines a class which implements a standalone SOAP server. When it is initialized, the class creates a `InterestCalculator` object (an instance of the class we just wrote). It then uses `add_method` to add the two methods implemented by this class, `compound` and `call_count`. Finally, the code creates and runs an instance of this server class. The parameters to the constructor are the name of the application, the default namespace, the address of the interface to use, and the port.

We then need to write some client code to access this server. The client creates a local proxy for the `InterestCalculator` service on the server, adds the methods it wants to use, and then calls them.

```
require 'soap/rpc/driver'
proxy = SOAP::RPC::Driver.new("http://localhost:12321",
                              "http://pragprog.com/InterestCalc")
proxy.add_method('compound', 'principal', 'rate', 'freq', 'years')
proxy.add_method('call_count')
puts "Call count: #{proxy.call_count}"
puts "5 years, compound annually: #{proxy.compound(100, 0.06, 1, 5)}"
puts "5 years, compound monthly:  #{proxy.compound(100, 0.06, 12, 5)}"
puts "Call count: #{proxy.call_count}"
```

To test this, we can run the server in one console window (the output here has been reformated slightly to fit the width of this page).

```
% ruby server.rb
I, [2004-07-26T10:55:51.629451 #12327]  INFO
        -- Calc: Start of Calc.
I, [2004-07-26T10:55:51.633755 #12327]  INFO
         -- Calc: WEBrick 1.3.1
I, [2004-07-26T10:55:51.635146 #12327]  INFO
         -- Calc: ruby 1.8.2 (2004-07-26) [powerpc-darwin]
I, [2004-07-26T10:55:51.639347 #12327]  INFO
         -- Calc: WEBrick::HTTPServer#start: pid=12327 port=12321
```

We then run the client in another window.

```
% ruby client.rb
Call count: 0
5 years, compound annually: 133.82255776
5 years, compound monthly:  134.885015254931
Call count: 2
```

Looking good! Flush with success, we call all our friends over and run it again.

```
% ruby client.rb
Call count: 2
5 years, compound annually: 133.82255776
5 years, compound monthly:  134.885015254931
Call count: 4
```

Notice how the call count now starts at two the second time we run the client. The server creates a single `InterestCalculator` object to service incoming requests, and this object is reused for each request.

SOAP and Google

Obviously the real benefit of SOAP is the way it lets you interoperate with other services on the Web. As an example, let's write some Ruby code to send queries to Google's Web API.

Before sending queries to Google, you need a developer key. This is available from Google—go to http://www.google.com/apis and follow the instructions in step 2, *Create a Google Account*. After you fill in your e-mail address and supply a password, Google will send you a developer key. In the following examples, we'll assume that you've stored this key in the file `.google_key` in your home directory.

Let's start at the most basic level. Looking at the documentation for the Google API method `doGoogleSearch`, we discover it has ten (!) parameters.

`key`	The developer key
`q`	The query string
`start`	The index of the first required result
`maxResults`	The maximum number of results to return per query
`filter`	If enabled, compresses results so that similar pages and pages from the same domain are only shown once
`restrict`	Restricts the search to a subset of the Google Web index
`safeSearch`	If enabled, removes possible adult content from the results
`lr`	Restricts the search to documents in a given set of languages
`ie`	Ignored (was input encoding)
`oe`	Ignored (was output encoding)

We can use the `add_method` call to construct a SOAP proxy for the `doGoogleSearch` method. The following example does just that, printing out the first entry returned if you search Google for the term *pragmatic*.

```
require 'soap/rpc/driver'
require 'cgi'
endpoint = 'http://api.google.com/search/beta2'
namespace = 'urn:GoogleSearch'
soap = SOAP::RPC::Driver.new(endpoint, namespace)
soap.add_method('doGoogleSearch', 'key', 'q', 'start',
                                 'maxResults', 'filter', 'restrict',
                                 'safeSearch', 'lr', 'ie', 'oe')
query = 'pragmatic'
key = File.read(File.join(ENV['HOME'], ".google_key")).chomp
result = soap.doGoogleSearch(key, query, 0, 1, false, '',
                             false, '', '', '')
printf "Estimated number of results is %d.\n",
       result.estimatedTotalResultsCount
printf "Your query took %6f seconds.\n", result.searchTime
first = result.resultElements[0]
puts first.title
puts first.uRL
puts CGI.unescapeHTML(first.snippet)
```

Run this, and you'll see something such as the following (notice how the query term has been highlighted by Google).

```
Estimated number of results is 550000.
Your query took 0.123762 seconds.
The <b>Pragmatic</b> Programmers, LLC
http://www.pragmaticprogrammer.com/
Home of Andrew Hunt and David Thomas's best-selling book 'The
<b>Pragmatic</b> Programmer'<br> and The '<b>Pragmatic</b> Starter Kit
(tm)' series. <b>...</b> The <b>Pragmatic</b> Bookshelf TM. <b>...</b>
```

However, SOAP allows for the dynamic discovery of the interface of objects on the server. This is done using WSDL, the Web Services Description Language. A WSDL file is an XML document that describes the types, methods, and access mechanisms for a Web services interface. SOAP clients can read WSDL files to create the interfaces to a server automatically.

The Web page http://api.google.com/GoogleSearch.wsdl contains the WSDL describing the Google interface. We can alter our search application to read this WSDL, which removes the need to add the doGoogleSearch method explicitly.

```
require 'soap/wsdlDriver'
require 'cgi'
WSDL_URL = "http://api.google.com/GoogleSearch.wsdl"
soap = SOAP::WSDLDriverFactory.new(WSDL_URL).create_rpc_driver
query = 'pragmatic'
key = File.read(File.join(ENV['HOME'], ".google_key")).chomp
result = soap.doGoogleSearch(key, query, 0, 1, false,
                             '', false, '', '', '')
```

```
printf "Estimated number of results is %d.\n",
       result.estimatedTotalResultsCount
printf "Your query took %6f seconds.\n", result.searchTime
first = result.resultElements[0]
puts first.title
puts first.uRL
puts CGI.unescapeHTML(first.snippet)
```

Finally, we can take this a step further using Ian Macdonald's Google library (available in the RAA). It encapsulates the Web services API behind a nice interface (nice if for no other reason than it eliminates the need for all those extra parameters). The library also has methods to construct the date ranges and other restrictions on a Google query and provides interfaces to the Google cache and the spell-checking facility. The following code is our "pragmatic" search using Ian's library.

```
require 'google'
require 'cgi'
key = File.read(File.join(ENV['HOME'], ".google_key")).chomp
google = Google::Search.new(key)
result = google.search('pragmatic')
printf "Estimated number of results is %d.\n",
       result.estimatedTotalResultsCount
printf "Your query took %6f seconds.\n", result.searchTime
first = result.resultElements[0]
puts first.title
puts first.url
puts CGI.unescapeHTML(first.snippet)
```

More Information

Ruby Web programming is a big topic. To dig deeper, you may want to look at Chapter 9 in *The Ruby Way* [Ful01], where you'll find many examples of network and Web programming, and Chapter 6 of *The Ruby Developer's Guide* [FJN02], where you'll find some good examples of structuring CGI applications, along with some example Iowa code.

If SOAP strikes you being complex, you may want to look at using XML-RPC, which is described briefly on page 736.

A number of other Ruby Web development frameworks are available on the 'net. This is a dynamic area: new contenders appear constantly, and it is hard for a printed book to be definitive. However, two frameworks that are currently attracting mindshare in the Ruby community are

- Rails (`http://www.rubyonrails.org`), and
- CGIKit (`http://www.spice-of-life.net/cgikit/index_en.html`).

Chapter 19

Ruby Tk

The Ruby Application Archive contains several extensions that provide Ruby with a graphical user interface (GUI), including extensions for Fox, GTK, and others.

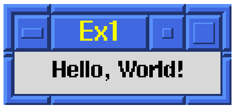
The Tk extension is bundled in the main distribution and works on both Unix and Windows systems. To use it, you need to have Tk installed on your system. Tk is a large system, and entire books have been written about it, so we won't waste time or resources by delving too deeply into Tk itself but instead concentrate on how to access Tk features from Ruby. You'll need one of these reference books in order to use Tk with Ruby effectively. The binding we use is closest to the Perl binding, so you probably want to get a copy of *Learning Perl/Tk* [Wal99] or *Perl/Tk Pocket Reference* [Lid98].

Tk works along a composition model—that is, you start by creating a container (such as a `TkFrame` or `TkRoot`) and then create the widgets (another name for GUI components) that populate it, such as buttons or labels. When you are ready to start the GUI, you invoke `Tk.mainloop`. The Tk engine then takes control of the program, displaying widgets and calling your code in response to GUI events.

Simple Tk Application

A simple Tk application in Ruby may look something like this.

```
require 'tk'
root = TkRoot.new { title "Ex1" }
TkLabel.new(root) do
  text  'Hello, World!'
  pack('padx' => 15, 'pady' => 15, 'side' => 'left')
end
Tk.mainloop
```



Let's look at the code a little more closely. After loading the `tk` extension module, we create a root-level frame using `TkRoot.new`. We then make a `TkLabel` widget as a child of the root frame, setting several options for the label. Finally, we pack the root frame and enter the main GUI event loop.

It's a good habit to specify the root explicitly, but you could leave it out—along with the extra options—and boil this down to a three-liner.

```
require 'tk'
TkLabel.new { text 'Hello, World!'; pack }
Tk.mainloop
```

That's all there is to it! Armed with one of the Perl/Tk books we reference at the start of this chapter, you can now produce all the sophisticated GUIs you need. But then again, if you'd like to stick around for some more details, here they come.

Widgets

Creating widgets is easy. Take the name of the widget as given in the Tk documentation and add a Tk to the front of it. For instance, the widgets Label, Button, and Entry become the classes TkLabel, TkButton, and TkEntry. You create an instance of a widget using new, just as you would any other object. If you don't specify a parent for a given widget, it will default to the root-level frame. We usually want to specify the parent of a given widget, along with many other options—color, size, and so on. We also need to be able to get information back from our widgets while our program is running by setting up *callbacks* (routines invoked when certain events happen) and sharing data.

Setting Widget Options

If you look at a Tk reference manual (the one written for Perl/Tk, for example), you'll notice that options for widgets are usually listed with a hyphen—as a command-line option would be. In Perl/Tk, options are passed to a widget in a Hash. You can do that in Ruby as well, but you can also pass options using a code block; the name of the option is used as a method name within the block and arguments to the option appear as arguments to the method call. Widgets take a parent as the first argument, followed by an optional hash of options or the code block of options. Thus, the following two forms are equivalent.

```
TkLabel.new(parent_widget) do
  text    'Hello, World!'
  pack('padx'  => 5,
       'pady'  => 5,
       'side'  => 'left')
end
# or
TkLabel.new(parent_widget, 'text' => 'Hello, World!').pack(...)
```

One small caution when using the code block form: the scope of variables is not what you think it is. The block is actually evaluated in the context of the widget's object, not the caller's. This means that the caller's instance variables will not be available in the block, but local variables from the enclosing scope and globals will be (not that you use global variables, of course.) We'll show option passing using both methods in the examples that follow.

Distances (as in the `padx` and `pady` options in these examples) are assumed to be in pixels but may be specified in different units using one of the suffixes `c` (centimeter), `i` (inch), `m` (millimeter), or `p` (point). `"12p"`, for example, is twelve points.

Getting Widget Data

We can get information back from widgets by using callbacks and by binding variables.

Callbacks are very easy to set up. The `command` option (shown in the `TkButton` call in the example that follows) takes a `Proc` object, which will be called when the callback fires. Here we pass the proc in as a block associated with the method call, but we could also have used `Kernel.lambda` to generate an explicit `Proc` object.

```
require 'tk'
TkButton.new do
  text "EXIT"
  command { exit }
  pack('side'=>'left', 'padx'=>10, 'pady'=>10)
end
Tk.mainloop
```

We can also bind a Ruby variable to a Tk widget's value using a `TkVariable` proxy. This arranges things so that whenever the widget's value changes, the Ruby variable will automatically be updated, and whenever the variable is changed, the widget will reflect the new value.

We show this in the following example. Notice how the `TkCheckButton` is set up; the documentation says that the `variable` option takes a *var reference* as an argument. For this, we create a Tk variable reference using `TkVariable.new`. Accessing `mycheck.value` will return the string "0" or "1" depending on whether the checkbox is checked. You can use the same mechanism for anything that supports a var reference, such as radio buttons and text fields.

```
require 'tk'
packing = { 'padx'=>5, 'pady'=>5, 'side' => 'left' }
checked = TkVariable.new
def checked.status
  value == "1" ? "Yes" : "No"
end
```

```
status = TkLabel.new do
  text checked.status
  pack(packing)
end
TkCheckButton.new  do
  variable checked
  pack(packing)
end
TkButton.new do
  text "Show status"
  command { status.text(checked.status) }
  pack(packing)
end
Tk.mainloop
```

Setting/Getting Options Dynamically

In addition to setting a widget's options when it's created, you can reconfigure a widget while it's running. Every widget supports the `configure` method, which takes a `Hash` or a code block in the same manner as new. We can modify the first example to change the label text in response to a button click.

```
require 'tk'
root = TkRoot.new { title "Ex3" }
top = TkFrame.new(root) { relief 'raised'; border 5 }
lbl = TkLabel.new(top) do
  justify 'center'
  text    'Hello, World!'
  pack('padx'=>5, 'pady'=>5, 'side' => 'top')
end
TkButton.new(top) do
  text "Ok"
  command { exit }
  pack('side'=>'left', 'padx'=>10, 'pady'=>10)
end
TkButton.new(top) do
  text "Cancel"
  command { lbl.configure('text'=>"Goodbye, Cruel World!") }
  pack('side'=>'right', 'padx'=>10, 'pady'=>10)
end
top.pack('fill'=>'both', 'side' =>'top')
Tk.mainloop
```

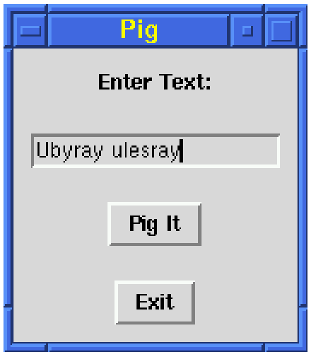
Now when the `Cancel` button is clicked, the text in the label will change immediately from "`Hello, World!`" to "`Goodbye, Cruel World!`"

You can also query widgets for particular option values using `cget`.

```
require 'tk'
b = TkButton.new do
  text     "OK"
  justify  "left"
  border   5
end
b.cget('text')      →   "OK"
b.cget('justify')   →   "left"
b.cget('border')    →   5
```

Sample Application

Here's a slightly longer example, showing a genuine application—a pig latin generator. Type in the phrase such as **Ruby rules**, and the `Pig It` button will instantly translate it into pig latin.



```
require 'tk'
class PigBox
  def pig(word)
    leading_cap = word =~ /^[A-Z]/
    word.downcase!
    res = case word
      when /^[aeiouy]/
        word+"way"
      when /^([^aeiouy]+)(.*)/
        $2+$1+"ay"
      else
        word
    end
    leading_cap ? res.capitalize : res
  end
  def show_pig
    @text.value = @text.value.split.collect{|w| pig(w)}.join(" ")
  end
  def initialize
    ph = { 'padx' => 10, 'pady' => 10 }      # common options
    root = TkRoot.new { title "Pig" }
    top = TkFrame.new(root) { background "white" }
    TkLabel.new(top) {text 'Enter Text:' ; pack(ph) }
    @text = TkVariable.new
    TkEntry.new(top, 'textvariable' =>  @text).pack(ph)
    pig_b = TkButton.new(top) { text 'Pig It'; pack ph}
    pig_b.command { show_pig }
    exit_b = TkButton.new(top) {text 'Exit'; pack ph}
    exit_b.command { exit }
    top.pack('fill'=>'both', 'side' =>'top')
  end
end
PigBox.new
Tk.mainloop
```

Geometry Management

In the example code in this chapter, you'll see references to the widget method `pack`. That's a very important call, as it turns out—leave it off and you'll never see the widget. `pack` is a command that tells the geometry manager to place the widget according to constraints that we specify. Geometry managers recognize three commands.

Command	Placement Specification
`pack`	Flexible, constraint-based placement
`place`	Absolute position
`grid`	Tabular (row/column) position

As `pack` is the most commonly used command, we'll use it in our examples.

Binding Events

Our widgets are exposed to the real world; they get clicked, the mouse moves over them, the user tabs into them; all these things, and more, generate *events* that we can capture. You can create a *binding* from an event on a particular widget to a block of code, using the widget's `bind` method.

For instance, suppose we've created a button widget that displays an image. We'd like the image to change when the user's mouse is over the button.

```
require 'tk'

image1 = TkPhotoImage.new { file "img1.gif" }
image2 = TkPhotoImage.new { file "img2.gif" }

b = TkButton.new(@root) do
  image    image1
  command  { exit }
  pack
end

b.bind("Enter") { b.configure('image'=>image2) }
b.bind("Leave") { b.configure('image'=>image1) }

Tk.mainloop
```

First, we create two GIF image objects from files on disk, using `TkPhotoImage`. Next we create a button (very cleverly named "b"), which displays the image `image1`. We then bind the `Enter` event so that it dynamically changes the image displayed by the button to `image2` when the mouse is over the button, and the `Leave` event to revert back to `image1` when the mouse leaves the button.

This example shows the simple events `Enter` and `Leave`. But the named event given as an argument to `bind` can be composed of several substrings, separated with dashes, in the order *modifier-modifier-type-detail*. Modifiers are listed in the Tk reference and include `Button1`, `Control`, `Alt`, `Shift`, and so on. *Type* is the name of the event (taken from the X11 naming conventions) and includes events such as `ButtonPress`, `KeyPress`, and `Expose`. *Detail* is either a number from 1 to 5 for buttons or a keysym for keyboard input. For instance, a binding that will trigger on mouse release of button 1 while the control key is pressed could be specified as

```
Control-Button1-ButtonRelease
```

or

```
Control-ButtonRelease-1
```

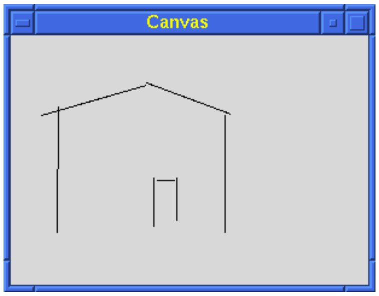
The event itself can contain certain fields such as the time of the event and the x and y positions. `bind` can pass these items to the callback, using *event field codes*. These are used like `printf` specifications. For instance, to get the x and y coordinates on a mouse move, you'd specify the call to `bind` with three parameters. The second parameter is the `Proc` for the callback, and the third parameter is the event field string.

```
canvas.bind("Motion", lambda {|x, y| do_motion (x, y)}, "%x %y")
```

Canvas

Tk provides a `Canvas` widget with which you can draw and produce PostScript output. Figure 19.1 on the next page shows a simple bit of code (adapted from the distribution) that will draw straight lines. Clicking and holding button 1 will start a line, which will be "rubber-banded" as you move the mouse around. When you release button 1, the line will be drawn in that position.

A few mouse clicks, and you've got an instant masterpiece.



As they say, "We couldn't find the artist, so we had to hang the picture. . . ."

Figure 19.1. Drawing on a Tk Canvas

```
require 'tk'

class Draw
  def do_press(x, y)
    @start_x = x
    @start_y = y
    @current_line = TkcLine.new(@canvas, x, y, x, y)
  end

  def do_motion(x, y)
    if @current_line
      @current_line.coords @start_x, @start_y, x, y
    end
  end

  def do_release(x, y)
    if @current_line
      @current_line.coords @start_x, @start_y, x, y
      @current_line.fill 'black'
      @current_line = nil
    end
  end

  def initialize(parent)
    @canvas = TkCanvas.new(parent)
    @canvas.pack
    @start_x = @start_y = 0
    @canvas.bind("1", lambda {|e| do_press(e.x, e.y)})
    @canvas.bind("B1-Motion",
                 lambda {|x, y| do_motion(x, y)}, "%x %y")
    @canvas.bind("ButtonRelease-1",
                 lambda {|x, y| do_release(x, y)},
                 "%x %y")
  end
end

root = TkRoot.new { title 'Canvas' }
Draw.new(root)
Tk.mainloop
```

Scrolling

Unless you plan on drawing very small pictures, the previous example may not be all that useful. TkCanvas, TkListbox, and TkText can be set up to use scrollbars, so you can work on a smaller subset of the "big picture."

Communication between a scrollbar and a widget is bidirectional. Moving the scrollbar means that the widget's view has to change; but when the widget's view is changed by some other means, the scrollbar has to change as well to reflect the new position.

Since we haven't done much with lists yet, our scrolling example will use a scrolling list of text. In the following code fragment, we'll start by creating a plain old TkListbox and an associated TkScrollbar. The scrollbar's callback (set with command) will call the list widget's yview method, which will change the value of the visible portion of the list in the y direction.

After that callback is set up, we make the inverse association: when the list feels the need to scroll, we'll set the appropriate range in the scrollbar using TkScrollbar#set. We'll use this same fragment in a fully functional program in the next section.

```
list_w = TkListbox.new(frame) do
  selectmode 'single'
  pack 'side' => 'left'
end
list_w.bind("ButtonRelease-1") do
  busy do
    filename = list_w.get(*list_w.curselection)
    tmp_img = TkPhotoImage.new { file filename }
    scale   = tmp_img.height / 100
    scale   = 1 if scale < 1
    image_w.copy(tmp_img, 'subsample' => [scale, scale])
    image_w.pack
  end
end
scroll_bar = TkScrollbar.new(frame) do
  command {|*args| list_w.yview *args }
  pack    'side' => 'left', 'fill' => 'y'
end
list_w.yscrollcommand  {|first,last| scroll_bar.set(first,last) }
```

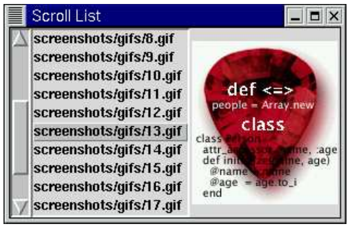
Just One More Thing

We could go on about Tk for another few hundred pages, but that's another book. The following program is our final Tk example—a simple GIF image viewer. You can select a GIF filename from the scrolling list, and a thumb nail version of the image will be displayed. We'll point out just a *few* more things.

Have you ever used an application that creates a “busy cursor” and then forgets to reset it to normal? A neat trick in Ruby will prevent this from happening. Remember how `File.new` uses a block to ensure that the file is closed after it is used? We can do a similar thing with the method `busy`, as shown in the next example.

This program also demonstrates some simple `TkListbox` manipulations—adding elements to the list, setting up a callback on a mouse button release,[1] and retrieving the current selection.

So far, we’ve used `TkPhotoImage` to display images directly, but you can also zoom, subsample, and show portions of images as well. Here we use the subsample feature to scale down the image for viewing.


Scroll List
screenshots/gifs/8.gif
screenshots/gifs/9.gif
screenshots/gifs/10.gif
screenshots/gifs/11.gif
screenshots/gifs/12.gif
screenshots/gifs/13.gif
screenshots/gifs/14.gif
screenshots/gifs/15.gif
screenshots/gifs/16.gif
screenshots/gifs/17.gif

```
require 'tk'
class GifViewer
  def initialize(filelist)
    setup_viewer(filelist)
  end
  def run
    Tk.mainloop
  end
  def setup_viewer(filelist)
    @root = TkRoot.new {title 'Scroll List'}
    frame = TkFrame.new(@root)
    image_w = TkPhotoImage.new
    TkLabel.new(frame) do
      image image_w
      pack 'side'=>'right'
    end
    list_w = TkListbox.new(frame) do
      selectmode 'single'
      pack 'side' => 'left'
    end
    list_w.bind("ButtonRelease-1") do
      busy do
        filename = list_w.get(*list_w.curselection)
        tmp_img = TkPhotoImage.new { file filename }
        scale   = tmp_img.height / 100
        scale   = 1 if scale < 1
        image_w.copy(tmp_img, 'subsample' => [scale, scale])
        image_w.pack
      end
    end
```

1. You probably want the button release, not the press, as the widget gets selected on the button press.

```
      filelist.each do |name|
        list_w.insert('end', name) # Insert each file name into the list
      end
      scroll_bar = TkScrollbar.new(frame) do
        command {|*args| list_w.yview *args }
        pack    'side' => 'left', 'fill' => 'y'
      end
      list_w.yscrollcommand  {|first,last| scroll_bar.set(first,last) }
      frame.pack
    end
    # Run a block with a 'wait' cursor
    def busy
      @root.cursor "watch" # Set a watch cursor
      yield
    ensure
      @root.cursor "" # Back to original
  end
  end
  viewer = GifViewer.new(Dir["screenshots/gifs/*.gif"])
  viewer.run
```

Translating from Perl/Tk Documentation

That's it, you're on your own now. For the most part, you can easily translate the documentation given for Perl/Tk to Ruby. There are a few exceptions; some methods are not implemented, and some extra functionality is undocumented. Until a Ruby/Tk book comes out, your best bet is to ask on the newsgroup or read the source code.

But in general, it's pretty easy to see what's happening. Remember that options may be given as a hash, or in code block style, and the scope of the code block is within the `TkWidget` being used, not your class instance.

Object Creation

In the Perl/Tk mapping, parents are responsible for creating their child widgets. In Ruby, the parent is passed as the first parameter to the widget's constructor.

```
Perl/Tk:  $widget = $parent->Widget( [ option => value ] )
Ruby:     widget = TkWidget.new(parent, option-hash)
          widget = TkWidget.new(parent) { code block }
```

You may not need to save the returned value of the newly created widget, but it's there if you do. Don't forget to pack a widget (or use one of the other geometry calls), or it won't be displayed.

Options

```
Perl/Tk:  -background => color
Ruby:     'background' => color
          { background color }
```

Remember that the code block scope is different.

Variable References

```
Perl/Tk:  -textvariable => \$variable
          -textvariable => varRef
Ruby:     ref = TkVariable.new
          'textvariable' => ref
          { textvariable ref }
```

Use `TkVariable` to attach a Ruby variable to a widget's value. You can then use the `value` accessors in `TkVariable` (`TkVariable#value` and `TkVariable#value=`) to affect the contents of the widget directly.