Basic Computer Coding: RUBY

BIBLIOTEX
Digital Library

2nd Edition

BASIC COMPUTER CODING: RUBY

2nd Edition

BIBLIOTEX
Digital Library

www.bibliotex.com

BASIC COMPUTER CODING: RUBY

2ND EDITION

www.bibliotex.com

email: info@bibliotex.com

e-book Edition 2022

ISBN: 978-1-98467-605-4 (e-book)

In Collaboration with 3G E-Learning LLC. Originally Published in printed book format by 3G E-Learning LLC with ISBN 978-1-98465-898-2

EDITORIAL BOARD

Aleksandar Mratinković was born on May 5, 1988 in Arandjelovac, Serbia. He has graduated on Economic high school (2007), The College of Tourism in Belgrade (2013), and also has a master degree of Psychology (Faculty of Philosophy, University of Novi Sad). He has been engaged in different fields of psychology (Developmental Psychology, Clinical Psychology, Educational Psychology and Industrial Psychology) and has published several scientific works.

Dan Piestun (PhD) is currently a startup entrepreneur in Israel working on the interface of Agriculture and Biomedical Sciences and was formerly president-CEO of the National Institute of Agricultural Research (INIA) in Uruguay. Dan is a widely published scientist who has received many honours during his career including being a two-time recipient of the Amit Golda Meir Prize from the Hebrew University of Jerusalem, his areas of expertise includes stem cell molecular biology, plant and animal genetics and bioinformatics. Dan's passion for applied science and technological solutions did not stop him from pursuing a deep connection to the farmer, his family and nature. Among some of his interest and practices counts enjoying working as a beekeeper and onboard fishing.

Hazem Shawky Fouda has a PhD. in Agriculture Sciences, obtained his PhD. From the Faculty of Agriculture, Alexandria University in 2008, He is working in Cotton Arbitration & Testing General Organization (CATGO).

Felecia Killings is the Founder and CEO of LiyahAmore Publishing, a publishing company committed to providing technical and educational services and products to Christian Authors. She operates as the Senior Editor and Writer, the Senior Writing Coach, the Content Marketing Specialist, Editor-in-Chief to the company's quarterly magazine, the Executive and Host of an international virtual network, and the Executive Director of the company's online school for Authors. She is a former high-school English instructor and professional development professor. She possesses a Master of Arts degree in Education and a Bachelor's degree in English and African American studies.

Dr. Sandra El Hajj, Ph.D. in Health Sciences from Nova Southeastern University, Florida, USA is a health professional specialized in Preventive and Global Health. With her 12 years of education obtained from one of the most prominent universities in Beirut, in addition to two leading universities in the State of Florida (USA), Dr. Sandra made sure to incorporate interdisciplinary and multicultural approaches in her work. Her long years of studies helped her create her own miniature world of knowledge linking together the healthcare field with Medical Research, Statistics, Food Technology, Environmental & Occupational Health, Preventive Health and most noteworthy her precious last degree of Global Health. Till today, she is the first and only doctor specialized in Global Health in the Middle East area.

Fozia Parveen has a Dphil in Sustainable Water Engineering from the University of Oxford. Prior to this she has received MS in Environmental Sciences from National University of Science and Technology (NUST), Islamabad Pakistan and BS in Environmental Sciences from Fatima Jinnah Women University (FJWU), Rawalpindi.

Igor Krunic 2003-2007 in the School of Economics. After graduating in 2007, he went on to study at The College of Tourism, at the University of Belgrade where he got his bachelor degree in 2010. He was active as a third-year student representative in the student parliament.Then he went on the Faculty of science, at the University of Novi Sad where he successfully defended his master's thesis in 2013. The crown of his study was the work titled Opportunities for development of cultural tourism in Cacak". Later on, he became part of a multinational company where he got promoted to a deputy director of logistic. Nowadays he is a consultant and writer of academic subjects in the field of tourism.

Dr. Jovan Pehcevski obtained his PhD in Computer Science from RMIT University in Melbourne, Australia in 2007. His research interests include big data, business intelligence and predictive analytics, data and information science, information retrieval, XML, web services and service-oriented architectures, and relational and NoSQL database systems. He has published over 30 journal and conference papers and he also serves as a journal and conference reviewer. He is currently working as a Dean and Associate Professor at European University in Skopje, Macedonia.

Dr. Tanjina Nur finished her PhD in Civil and Environmental Engineering in 2014 from University of Technology Sydney (UTS). Now she is working as Post-Doctoral Researcher in the Centre for Technology in Water and Wastewater (CTWW) and published about eight International journal papers with 80 citations. Her research interest is wastewater treatment technology using adsorption process.

Stephen obtained his PhD from the University of North Carolina at Charlotte in 2013 where his graduate research focused on cancer immunology and the tumor microenvironment. He received postdoctoral training in regenerative and translational medicine, specifically gastrointestinal tissue engineering, at the Wake Forest Institute of Regenerative Medicine. Currently, Stephen is an instructor for anatomy and physiology and biology at Forsyth Technical Community College.

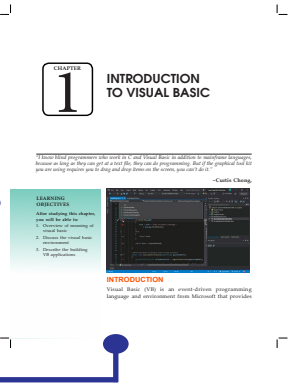
Michelle holds a Masters of Business Administration from the University of Phoenix, with a concentration in Human Resources Management. She is a professional author and has had numerous articles published in the Henry County Times and has written and revised several employee handbooks for various YMCA organizations throughout the United States.

HOW TO USE THE BOOK

This book has been divided into many chapters. Chapter gives the motivation for this book and the use of templates. The text is presented in the simplest language. Each paragraph has been arranged under a suitable heading for easy retention of concept. Keywords are the words that academics use to reveal the internal structure of an author's reasoning. Review questions at the end of each chapter ask students to review or explain the concepts. References provides the reader an additional source through which he/she can obtain more information regarding the topic.

LEARNING OBJECTIVES

See what you are going to cover and what you should already know at the start of each chapter


CHAPTER 1

INTRODUCTION TO VISUAL BASIC

"I know blind programmers who work in C and Visual Basic in addition to mainframe languages, because as long as they can get at a text file, they can do programming. But if the graphical tool kit you are using requires you to drag and drop items on the screen, you can't do it."

–Curtis Chong,

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Overview of meaning of visual basic
2. Discuss the visual basic programming
3. Describe the building VB applications

INTRODUCTION

Visual Basic (VB) is an event-driven programming language and environment from Microsoft that provides

enhancements, including the striking ability of creating web based applications. The extended support for Visual Basic 6.0 was ceased in the month of March in 2008. The basic parts of development environment of Visual Basic 6, however, still run in all the 32-bit Microsoft windows, including Windows 8.1.

After the cessation of mainstream and extended support for Visual Basic 6.0 caused a number of programs to show concern. The community members then created a lobby of users and a petition was signed by them. The basic aim of this petition was to ensure that the product remains alive. However, the petition did not attain its aim effectively.

1.1.2 The Importance of Visual Basic Programming Language

Visual Basic is regarded as the third generation event-drivon programming language. It was released in 1987. Being the first visual development tool from Microsoft, it is considered as one of the most powerful programming languages. As compared to other computer programming languages, such as, C, C++, it is easy to learn and understand, provided that one has determination and dedication to do so.

Visual basic programming language allows programmers to create software interface and codes in an easy to use graphical environment. VB is the combination of different components that are used on forms having specific attributes and actions with the help of those components. On the one hand it allows programmers to develop seldom based applications rapidly; on the other hand, it helps greatly in accessing data bases, using ADO while letting the programmers use ActiveX controls and various objects. While it is intended more to develop applications, it is also useful for games development for particular or limited purposes, unlike C++ that is more suitable for developing games.

As compared to other languages, Visual basic may be slower though, yet it is flexible and it can be rightly said that things that are difficult in other languages are comparatively easier in visual basic programming language. It may also be said that, since it is one of the most popular programming languages, lots of related books and material and other resources are available and can be accessed for developing programming skills at visual basic programming language conveniently.

One of the most important things to be considered with regard to programming in Visual basic is that the structure of VB is designed in a way that allows programmers to create executable code – Exe files. It enables programmers to develop programs that can be used as front end to databases. Besides, it's with the help of visual basic tools, one can change the abstract ideas into programs or into the whole software while it allows revising and modifying the programs fittingly.

ABOUT THIS CHAPTER

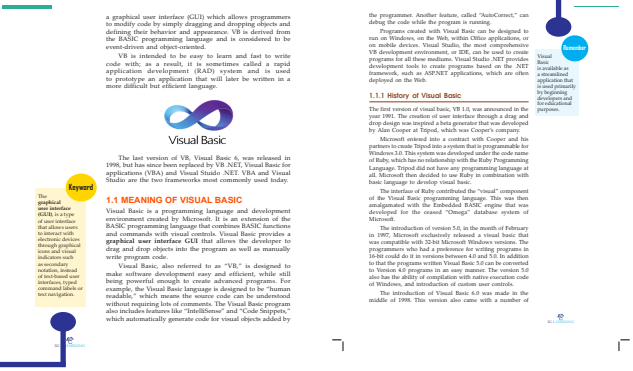
An introduction is a beginning of section which states the purpose and goals of the topics which are discussed in the chapter. It also starts the topics in brief.

REMEMBER

This revitalizes a must read information of the topic.

KEYWORDS

This section contains some important definitions that are discussed in the chapter. A keyword is an index entry that identifies a specific record or document. It also gives the extra information to the reader and an easy way to remember the word definition.


Keyword

The graphical user interface (GUI) is a type of user interface that allows users to interact with electronic devices through graphical icons and visual indicators such as secondary notation, as opposed to of text-based user interfaces, typed command labels or text navigation.

a graphical user interface (GUI) which allows programmers to modify code by simply dragging and dropping objects and defining their behavior and appearance. VB is derived from the BASIC programming language and is considered to be event-driven and object-oriented.

VB is intended to be easy to learn and fast to write code with; as a result, it is sometimes called a rapid application development (RAD) system and is used to prototype an application that will later be written in a more difficult but efficient language.

Visual Basic

The last version of VB, Visual Basic 6, was released in 1998, but has since been replaced by VB .NET, Visual Basic for applications (VBA) and Visual Studio .NET. VBA and Visual Studio are the two frameworks most commonly used today.

1.1 MEANING OF VISUAL BASIC

Visual Basic is a programming language and development environment created by Microsoft. It is an extension of the BASIC programming language that combines BASIC functions and commands with visual controls. Visual Basic provides a **graphical user interface GUI** that allows the developer to drag and drop objects into the program as well as manually write program code.

Visual Basic, also referred to as "VB," is designed to make software development easy and efficient, while still being powerful enough to create advanced programs. For example, the Visual Basic language is designed to be "human readable," which means the source code can be understood without requiring lots of comments. The Visual Basic program also includes features like "IntelliSense" and "Code Snippets," which automatically generate code for visual objects added by

the programmer. Another feature, called "AutoCorrect," can debug the code while the program is running.

Programs created with Visual Basic can be designed to run on Windows, on the Web, within Office applications, or on mobile devices. Visual Studio, the most comprehensive VB development environment, or IDE, can be used to create programs for all these mediums. Visual Studio .NET provides development tools to create programs based on the .NET framework, such as ASP.NET applications, which are often deployed on the Web.

Remember

Visual Basic is available as a standalone application that is used primarily by beginning developers and for educational purposes.

1.1.1 History of Visual Basic

The first version of visual basic, VB 1.0, was announced in the year 1991. The creation of user interface through a drag and drop design was inspired a beta generator that was developed by Alan Cooper at Tripod, which was Cooper's company.

Microsoft entered into a contract with Cooper and his partners to create Tripod into a system that is programmable for Windows 3.0. This system was developed under the code name of Ruby, which has no relationship with the Ruby Programming Language. Tripod did not have any programming language at all. Microsoft then decided to use Ruby in combination with basic language to develop visual basic.

The interface of Ruby contributed the "visual" component of the Visual Basic programming language. This was then amalgamated with the Embedded BASIC engine that was developed for the ceased "Omega" database system of Microsoft.

The introduction of version 5.0, in the month of February in 1997, Microsoft exclusively released a visual basic that was compatible with 32-bit Microsoft Windows versions. The programmers who had a preference for writing programs in 16-bit could do it in versions between 4.0 and 5.0. In addition to that the programs written Visual Basic 5.0 can be converted to Version 4.0 programs in an easy manner. The version 5.0 also has the ability of compilation with native execution code of Windows, and introduction of custom user controls.

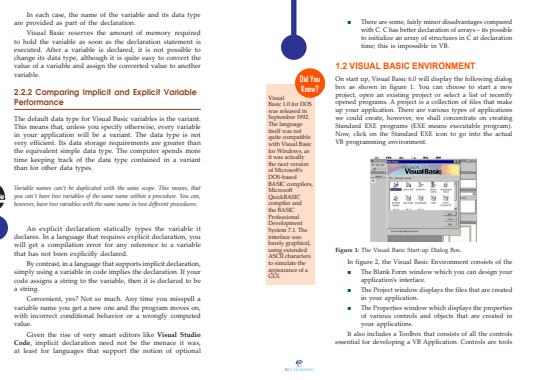
The introduction of Visual Basic 6.0 was made in the middle of 1998. This version also came with a number of

DID YOU KNOW?

This section equip readers the interesting facts and figures of the topic.

EXAMPLE

The book cabinets' examples to illustrate specific ideas in each chapter.



In each case, the name of the variable and its data type are provided as part of the declaration.

Visual Basic reserves the amount of memory required to hold the variable as soon as the declaration statement is executed. After a variable is declared, it is not possible to change its data type, although it is quite easy to convert the value of a variable and assign the converted value to another variable.

2.2.2 Comparing Implicit and Explicit Variable Performance

The default data type for Visual Basic variables is the variant. This means that, unless you specify otherwise, every variable in your application will be a variant. The data type is not very efficient. Its data storage requirements are greater than the equivalent simple data type. The computer spends more time keeping track of the data type contained in a variant than for other data types.

Variable names can't be duplicated with the same scope. This means, that you can't have two variables of the same name within a procedure. You can, however, have two variables with the same name in two different procedures.

An explicit declaration statically types the variable it declares. In a language that requires explicit declaration, you will get a compilation error for any reference to a variable that has not been explicitly declared.

By contrast, in a language that supports implicit declaration, simply using a variable in code implies the declaration. If your code assigns a string to the variable, then it is declared to be a string.

Convenient, yes? Not so much. Any time you misspell a variable name you get a new one and the program moves on, with incorrect conditional behavior or a wrongly computed value.

Given the rise of very smart editors like **Visual Studio Code**, implicit declaration need not be the menace it was, at least for languages that support the notion of optional

- There are some, fairly minor disadvantages compared with C. C has better declaration of arrays – its possible to initialize an array of structures in C at declaration time; this is impossible in VB.

1.2 VISUAL BASIC ENVIRONMENT

Did You Know?

Visual Basic 1.0 for DOS was released in September 1992. The language itself was not quite compatible with Visual Basic for Windows, as it was actually the next version of Microsoft's DOS-based BASIC compilers, Microsoft QuickBASIC compiler and the BASIC Professional Development System 7.1. The interface was barely graphical, using extended ASCII characters to simulate the appearance of a GUI.

On start up, Visual Basic 6.0 will display the following dialog box as shown in figure 1. You can choose to start a new project, open an existing project or select a list of recently opened programs. A project is a collection of files that make up your application. There are various types of applications we could create, however, we shall concentrate on creating Standard EXE programs (EXE means executable program). Now, click on the Standard EXE icon to go into the actual VB programming environment.

Figure 1: The Visual Basic Start-up Dialog Box.

In figure 2, the Visual Basic Environment consists of the

- The Blank Form window which you can design your application's interface.
- The Project window displays the files that are created in your application.
- The Properties window which displays the properties of various controls and objects that are created in your applications.

It also includes a Toolbox that consists of all the controls essential for developing a VB Application. Controls are tools

ROLE MODEL

A biography of someone who has/had acquired remarkable success in their respective field as Role Models are important because they give us the ability to imagine our future selves.

CASE STUDY

This reveals what students need to create and provide an opportunity for the development of key skills such as communication, group working and problem solving.

ROLE MODEL

ALAN COOPER: FATHER OF VISUAL BASIC

Born in San Francisco in 1952 and raised in Marin County, California, Alan Cooper has always taken the path less traveled. A rebellious teenager, he dropped out of high school, but eventually made his way to the College of Marin to pursue his interest in architecture. After an exploratory course in programming, it became clear that his future was in architecture—software architecture. After getting his associate degree and a COBOL programming job, he saw an advertisement for one of the first personal computers and .conceived an idea for a new business venture

In 1976, Cooper founded Structured Systems Group (SSG), a company Fire in the Valley authors Paul Freiberger and Michael Swaine said created "the first serious business software for microcomputers." In four years, Cooper wrote and shipped a dozen application programs. SSG became the archetype for many software startups in the early days of the personal computer revolution.

During the 1980s, after leaving SSG, Cooper invented, wrote, and sold three major software packages to prominent publishers. One of those was the visual programming front-end code named "Ruby," for what became Visual BASIC. Bill Gates purchased it from Cooper in 1988, noting that it would have significant impact across Microsoft's entire product line. Visual BASIC was deemed both a commercial and critical success, earning Cooper the moniker "Father of Visual BASIC." Visual BASIC has influenced integrated development languages ever since.

In 1990 Cooper became fascinated with the challenge of making software products that were easy to use and understand. He and his wife, Susan, founded Cooper Interaction Design (now "Cooper") to assist in what Cooper calls "interaction design." In the design field, Cooper's software development background was unique and, over the next few years, he invented many of the tools and techniques now standard in the user experience industry, including personas and scenarios,

CASE STUDY

FUJITSU FACILITATES SMOOTH MIGRATION TO VB.NET AT AN POST

Fujitsu has an excellent technical team, which works closely with our staff. We have had a good working relationship for many years and Fujitsu has an in-depth knowledge of our mission-critical application gained from several years' development and support work.

Challenge

A Post, one of Ireland's largest companies, is a major commercial organization providing a wide range of postal, communication, retail and financial services. With 9,600 employees throughout its national network of retail, processing and delivery points, the business also provides services to government departments, the National Treasury Management Agency and its own National Lottery Company. A decade ago, A Post implemented a new nationwide time and attendance system to calculate and record staff salary and wages functions. The Staff Remuneration and Administration Management System (STREAMS) is a bespoke, mission-critical application developed by Fujitsu as a reliable, scalable client server system using Microsoft technologies. The STREAMS front-end system gathers information and feeds the data to the company's HR, payroll and financial departments. It primarily creates more efficient processes for A Post to capture data for the weekly payroll run whilst simultaneously minimizing the number of payroll queries by employees. Following deployment, STREAMS improved cost center reporting, significantly lowered the time to record pay details and enhanced the processing of casual staff pay. During this period, Fujitsu provided quality support and maintenance services and application enhancements to increase functionality, ensuring the long-term reliability of STREAMS. For instance, as employee numbers steadily increased to exceed original expectations, Fujitsu boosted system performance by upgrading the infrastructure and optimizing the software. STREAMS originally employed Visual Basic (VB), a third-generation event-driven programming language and integrated development environment (IDE) from Microsoft. IDE provides programmers with comprehensive facilities for software development and comprises a source code editor, a compiler and/or an interpreter, build automation tools and a debugger. However, Microsoft no longer supports VB version 6.0, the edition employed by A Post. Syl Byrne, IT Manager Remuneration Services, A Post, explains: "To ensure that our business-critical application is future-proof, we needed to move to a platform that Microsoft will support for the foreseeable future." A Post therefore decided to migrate STREAMS to the VB.NET platform, an object-orientated programming language. This strategy would protect its investment for the next 10 years by creating a secure, scalable

KNOWLEDGE CHECK

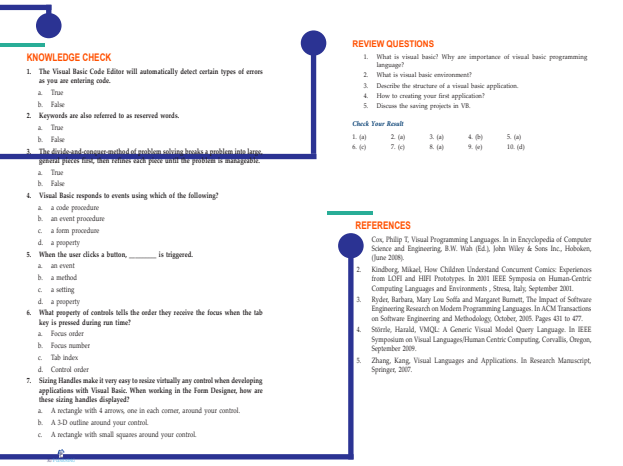
This is given to the students for progress check at the end of each chapter.

REVIEW QUESTIONS

This section is to analyze the knowledge and ability of the reader.

REFERENCES

References refer those books which discuss the topics given in the chapters in almost same manner.



KNOWLEDGE CHECK

1. **The Visual Basic Code Editor will automatically detect certain types of errors as you are entering code.**
   a. True
   b. False
2. **Keywords are also referred to as reserved words.**
   a. True
   b. False
3. **The divide-and-conquer method of problem solving breaks a problem into large general pieces first, then refines each piece until the problem is manageable.**
   a. True
   b. False
4. **Visual Basic responds to events using which of the following?**
   a. a code procedure
   b. an event procedure
   c. a form procedure
   d. a property
5. **When the user clicks a button, _____ is triggered.**
   a. an event
   b. a method
   c. a setting
   d. a property
6. **What property of controls tells the order they receive the focus when the tab key is pressed during run time?**
   a. Focus order
   b. Focus number
   c. Tab index
   d. Control order
7. **Sizing Handles make it very easy to resize virtually any control when developing applications with Visual Basic. When working in the Form Designer, how are these sizing handles displayed?**
   a. A rectangle with 4 arrows, one in each corner, around your control.
   b. A 3-D outline around your control.
   c. A rectangle with small squares around your control.

REVIEW QUESTIONS

1. What is visual basic? Why are importance of visual basic programming language?
2. What is visual basic environment?
3. Describe the structure of a visual basic application.
4. How to creating your first application?
5. Discuss the saving projects in VB.

Check Your Result

1. (a)	2. (a)	3. (a)	4. (b)	5. (a)
6. (c)	7. (c)	8. (a)	9. (e)	10. (d)

REFERENCES

1. Cox, Philip T, Visual Programming Languages. In in Encyclopedia of Computer Science and Engineering, B.W. Wah (Ed.), John Wiley & Sons Inc., Hoboken, (June 2008).
2. Kindborg, Mikael, How Children Understand Concurrent Comics: Experiences from LOFI and HIFI Prototypes. In 2001 IEEE Symposia on Human-Centric Computing Languages and Environments , Stresa, Italy, September 2001.
3. Ryder, Barbara, Mary Lou Soffa and Margaret Burnett, The Impact of Software Engineering Research on Modern Programming Languages. In ACM Transactions on Software Engineering and Methodology, October, 2005. Pages 431 to 477.
4. Störrle, Harald, VMQL: A Generic Visual Model Query Language. In IEEE Symposium on Visual Languages/Human Centric Computing, Corvallis, Oregon, September 2009.
5. Zhang, Kang, Visual Languages and Applications. In Research Manuscript, Springer, 2007.

TABLE OF CONTENTS

PREFACE

Ruby is a dynamic, reflective, object-oriented, general-purpose programming language. Ruby is a pure Object-Oriented language developed by Yukihiro Matsumoto. One of the goals of Ruby is to allow the simple and fast creation of web applications. The language itself satisfies this goal. Because of this, there is much less tedious work with this language than many other programming languages. More specifically, Ruby is a scripting language designed for front- and back-end web development, as well as other similar applications. It's a robust, dynamically typed, object-oriented language, with high-level syntax that makes programming with it feel almost like coding in English. In fact, some people feel that they can practically understand Ruby code before even learning how to program. In the world of computer programming, there is an infinite amount of information to learn. This book covers certain topics that are beneficial to the beginner.

Organization of the Book

This edition is systematically divided into nine chapters. This book is your guide to rapid, real-world software development with this unique and elegant language. The book will excite students on the capabilities of computer programming and inspire them to delve deeper into the computer science discipline. It will give you plenty of practice to commit basic Ruby syntax to long-term memory so you can focus on solving real-world problems and building real-world applications.

Chapter 1 presents an introduction to Ruby Basics. It also explains the strings, arrays, hashes. You will take a look at the general structure of a Rails application and the important commands used in the terminal.

Chapter 2 shows you how to work with Strings, Objects, and Variables. There are two kinds of objects: built-in objects and custom objects. Built-in objects are predefined objects that all programmers can use. They are

available with the core of the Ruby language or from various libraries. Custom objects are created by application programmers for their application domains.

Chapter 3 begins with the conditional statement. It also explains sets and files used in Python. Comparison operators and assignment operators is also given. Additionally, it also explains the logical operators and ternary operator.

Chapter 4 presents an exploration on working with loops in Ruby used to execute the same block of code a specified number of times. In this chapter, the loop statements supported by Ruby are also discussed.

Chapter 5 is aimed to the ruby regular expressions that can be matched against a string. Regular expression literals may include an optional modifier to control various aspects of matching.

Chapter 6 will take you through all the major functionalities related to Object Oriented Ruby. A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and methods within a class are called members of the class.

Chapter 7 discusses about the debugging process in software development whereby program analysts comb through computer code looking for "bugs" — the source of errors, flaws or security holes in the internal program instructions.

Chapter 8 focuses on reflection and metaprogramming. A Ruby program can dynamically set named variables, invoke named methods, and even define new classes and new methods.

Chapter 9 sheds light on methods, PRCs, lambdas, and closure Methods are a fundamental part of Ruby's syntax, but they are not values that Ruby programs can operate on. That is, Ruby's methods are not objects in the way that strings, numbers, and arrays are. It is possible, however, to obtain a Method object that represents a given method, and we can invoke methods indirectly through Method objects.

CHAPTER

1

RUBY BASICS

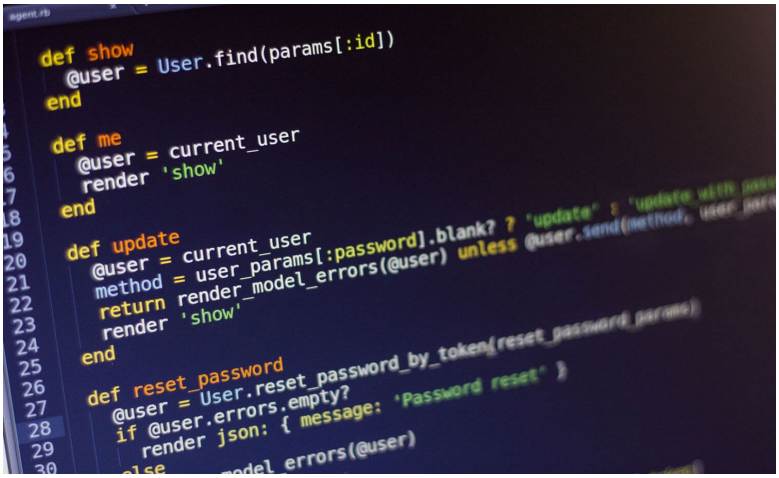
"Ruby is simple in appearance, but is very complex inside, just like our human body"

–Matz

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Focus on concept of ruby basics
2. Explain the rails basics
3. Describe the history of ruby



INTRODUCTION

Ruby is one of the fastest growing languages. Ruby is an interpreted, high-level, general-purpose programming language. It was designed and developed in the mid-1990s

by Yukihiro "Matz" Matsumoto in Japan. Websites like GitHub, Scribd, and Shopify are created with the help of Ruby. So if you have decided to become a developer or programmer, and are looking for a suitable platform, then Ruby is a great language to begin with.

Developed by Yukihiro "Matz" Matsumoto, it is a pure object-oriented programming language. It is a cross-platform language with support for multiple operating systems such as Windows, macOS, and several versions of UNIX. It is an open-source language and its latest version is 2.5.

Ruby inherits some features of languages like Smalltalk, Perl, and Python. Hence, it is widely used as server-side scripting language. It is a general-purpose, interpreted, and high-level language. It is also used to create Common Gateway Interface (CGI) scripts and can be easily embedded into languages like HTML. Its syntax is similar to other languages such as C++ and Perl.

Ruby provides support for multiple programming paradigms, dynamic type system, and automatic memory management. In addition to this, it also supports several GUI tools such as OpenGL, GTK, and Tcl/Tk. It also provides support for different databases such as DB2, MySQL, Oracle, and Sybase.

1.1 CONCEPT OF RUBY BASICS

Ruby is the programming language Ruby on Rails is written in. So most of the time you will be writing Ruby code. Therefore it is good to grasp the basics of Ruby. If you just want to play with Ruby, type **irb** into your console to start interactive ruby. There you can easily experiment with Ruby. To leave irb, type **exit**.

This is just a very small selection of concepts. This is especially true later on when we talk about what Arrays, Strings etc. can do.

Everything is an Object

Everything in Ruby is an Object.

This is a comment

#=> Output will be shown like this

#Everything is an Object of a Class

3.**class** #=> Fixnum

3.0.**class** #=> Float

"Hello".**class** #=> String

'hi'.**class** #=> String

Special values are objects too

nil.class #=> NilClass

true.class #=> TrueClass

false.class #=> FalseClass

Basic arithmetic

Ruby uses the standard **arithmetic operators**:

1 + 1 #=> 2

10 * 2 #=> 20

35 / 5 #=> 7

10.0 / 4.0 #=> 2.5

4 % 3 #=> 1 #Modulus

2 ** 5 #=> 32 #Exponent

Arithmetic is just syntactic sugar for calling a method on Numeric objects.

1.+(3) #=> 4

10.* 5 #=> 50

The Integer class has some integer-related functions, while the Math module contains trigonometric and other functions.

2.even? #=> true

12.gcd(8) #=> 4

Equality and Comparisons

#equality

1 == 1 #=> true

2 == 1 #=> false

Inequality

1 != 1 #=> false

2 != 1 #=> true

!**true** #=> false

!**false** #=> true

#Logical Operators

3>2 && 2>1 #=> true

2>3 && 2>1 #=> false

2>3 || 2>1 #=> true

#In Ruby, you can also use words

true and false #=> false

Keyword

Arithmetic operators take numerical values (either literals or variables) as their operands and return a single numerical value.

```
    true or false #=> true
true and not false => true
# apart from false itself, nil is the only other 'false' value
!nil   #=> true
!false #=> true
!1     #=> false
!0     #=> false
#comparisons
1 < 10 #=> true
1 > 10 #=> false
2 <= 2 #=> true
2 >= 2 #=> true
```

Challenge

What will the following code return?

```
true and 0 && !nil and 3 > 2
```

(Note: don't use such code in real programs!)

Please sign in or sign up to submit answers.

Alternatively, you can try out Learneroo before signing up.

Variables

```
x = 25 #=> 25
x #=> 25
# Note that assignment returns the value assigned
# This means you can do multiple assignment:
x = y = 10 #=> 10
x #=> 10
y #=> 10
# Variables can be dynamically assigned to different types
thing = 5   #=> 5
thing = "hello" #=> "hello"
thing = true #=> true
# By convention, use snake_case for variable names
snake_case = true
# Use descriptive variable names
```

Remember

Interactive Ruby (IRb) provides a shell for experimentation. Within the IRb shell, you can immediately view expression results, line by line.

path_to_project_root = '/good/name/'

path = '/bad/name/'

Challenge

a^b is considered *powerful* if (and only if) both of the following 2 conditions are met:

- $a^b >= 2 * b^2$
- $a^b >= (a*b)^2$

return true if a^b is powerful and false otherwise.

Please sign in or sign up to submit answers.

Alternatively, you can try out Learneroo before signing up.

1.1.1 General Concepts

Concept	Usage	Examples	Description
Comment	*# Comment text*	*# This text is a comment* some.ruby_code # A comment # some.ignored_ruby_code	Ruby ignores everything that is marked as a comment. It does not try to execute it. Comments are just there for you as information. Comments are also commonly used to *comment out code*. That is when you don't want some part of your code to execute but you don't want to delete it just yet, because you are trying different things out.
Variables	**variable** = *some_value*	**name** = *"Tobi"* **name** # => "Tobi" **sum** = *18 + 5* **sum** # => 23	With a variable you tell Ruby that from now on you want to refer to that value by the name you gave it. So for the first example, from now on when you use *name* Ruby will know that you meant *"Tobi"*.
Console output	**puts** *something*	**puts** *"Hello World"* **puts** *[1, 5, "mooo"]*	Prints its argument to the console. Can be used in Rails apps to print something in the console where the server is running.

Call a method	object.**method**(*arguments*)	string.**length** array.**delete_at**(*2*) string.**gsub**(*"ae", "ä"*)	Calling a method is also often referred to as *sending a message* in Ruby. Basically we are sending an object some kind of message and are waiting for its response. This message may have no arguments or multiple arguments, depending on the message. So we kindly ask the object to do something or give us some information. When you «call a method» or «send a message» something happens. In the first example we ask a String how long it is (how many characters it consists of). In the last example we substitute all occurrences of «ae" in the string with the German "ä". Different kinds of objects (Strings, Numbers, Arrays...) understand different messages.
Define a method	**def** *name(parameter) # method body* **end**	**def** *greet(name) puts "Hi there " + name* **end**	Methods are basically reusable units of behavior. And you can define them yourself just like this. Methods are small and focused on implementing a specific behavior. Our example method is focused on greeting people. You could call it like this: greet("Tobi")
Equality	object == other	true == true # => true 3 == 4 # => false "Hello" == «Hello» # => true "Helo" == "Hello" # => false	With two equal signs you can check if two things are the same. If so, true will be returned; otherwise, the result will be false.
Inequality	object **!=** other	true **!=** true # => false 3 **!=** 4 # => true	Inequality is the inverse to equality, e.g. it results in true when two values are not the same and it results in false when they are the same.
Decisions with if	**if** *condition #* happens when true **else** # happens when false **end**	**if** *input == password* grant_access **else** deny_access **end**	With if-clauses you can decide based upon a *condition* what to do. When the condition is considered true, then the code after it is executed. If it is considered false, the code after the «else» is executed. In the example, access is granted based upon the decision if a given input matches the password.

Constants	**CONSTANT =** *some_value*	**PI = 3.1415926535 PI** # => 3.1415926535 **ADULT_AGE** = 18 **ADULT_AGE** # => 18	Constants look like variables, just in UPCASE. Both hold values and give you a name to refer to those values. However while the value a variable holds may change or might be of an unknown value (if you save user input in a variable) constants are different. They have a known value that should never change. Think of it a bit like mathematical or physical constants. These don't change, they always refer to the same value.

1.1.2 Numbers

Numbers are what you would expect them to be, normal numbers that you use to perform basic math operations.

Concept	Usage	Examples	Description
normal Number	**number_of_your_choice**	**0** **-11** **42**	Numbers are natural for Ruby, you just have to enter them!
Decimals	**main.decimal**	**3.2** **-5.0**	You can achieve decimal numbers in Ruby simply by adding a point.
Basic Math	n **operator** *m*	2 + *3* # => 5 5 - *7* # => -2 8 * *7* # => 56 84 / *4* # => 21	In Ruby you can easily use basic math operations. In that sense you may use Ruby as a super-powered calculator.
Comparison	n **operator** *m*	12 > *3* # => true 12 < *3* # => false 7 >= *7* # => true	Numbers may be compared to determine if a number is bigger or smaller than another number. When you have the age of a person saved in the age variable you can see if that person is considered an adult in Germany: age >= 18 # true or false

1.1.3 Strings

Strings are used to hold textual information. They may contain single characters, words, sentences or a whole book. However you may just think of them as an ordered collection of characters.

Concept	Usage	Examples	Description
Create	'A string'	'Hello World' 'a' 'Just characters 129 _!$%^' ''	A string is created by putting quotation marks around a character sequence. A Ruby style guide recommends using single quotes for simple strings.
Interpolation	"A string and an #{expression}"	"Email: #{user.email}" "The total is #{2 + 2}" "A simple string"	You can combine a string with a variable or Ruby expression using double quotation marks. This is called "interpolation." It is okay to use double quotation marks around a simple string, too.
Length	string.**length**	"Hello"**.length** # => 5 ""**.length** # => 0	You can send a string a message, asking it how long it is and it will respond with the number of characters it consists of. You could use this to check if the desired password of a user exceeds the required minimum length. Notice how we add a comment to show the expected result.
Concatenate	string **+** *string2*	"Hello " **+** *"reader"* # => "Hello reader" "a" **+** *"b"* **+** *"c"* # => «abc"	Concatenates two or more strings together and returns the result.
Substitute	string.**gsub**(*a_string, substitute)*	"Hae"**.gsub**(*"ae", "ä")* # => *"Hä"* "Hae"**.gsub**(*"b", "ä")* # => *"Hae"* "Greenie"**.gsub**(*"e", "u")* # => *"Gruuniu"*	*gsub* stands for «globally substitute». It substitutes all occurrences of a_string within the string with substitute.
Access	string**[***position***]**	"Hello"**[***1***]** # => "e"	Access the character at the given position in the string. Be aware that the first position is actually position *0*.

1.1.4 Arrays

An array is an ordered collection of items which is indexed by numbers. So an array contains multiple objects that are mostly related to each other. So what could you do? You could store a collection of the names of your favorite fruits and name it *fruits*.

This is just a small selection of things an Array can do.

Concept	Usage	Examples	Description
Create	**[***contents***]**	**[]** **[***"Rails", "fun", 5***]**	Creates an Array, empty or with the specified contents.
Number of elements	array**.size**	[]**.size** # => 0 [1, 2, 3]**.size** # => 3 ["foo", "bar"]**.size** # => 2	Returns the number of elements in an Array.
Access	array**[***position***n]**	array = ["hi", "foo", "bar"] array**[***0***]** # => "hi" array[2] # => "bar"	As an Array is a collection of different elements, you often want to access a single element of the Array. Arrays are indexed by numbers so you can use a number to access an individual element. Be aware that the numbering actually starts with "0" so the first element actually is the 0th. And the last element of a three element array is element number 2.
Adding an element	array **<<** *element*	array = [1, 2, 3] array **<<** 4 array # => [1, 2, 3, 4]	Adds the element to the end of the array, increasing the size of the array by one.
Assigning	array**[***number***] =** *value*	array = ["hi", "foo", "bar"] array**[***2***] =** *"new"* array # => ["hi", "foo", "new"]	Assigning new Array Values works a lot like accessing them; use an equals sign to set a new value. Voila! You changed an element of the array! Weehuuuuu!
Delete at index	array**.delete_a***t(i)*	array = [0, 14, 55, 79] array**.delete_at**(2) array # => [0, 14, 79]	Deletes the element of the array at the specified index. Remember that indexing starts at 0. If you specify an index larger than the number of elements in the array, nothing will happen.
Iterating	array**.each** *do |e| .. en*d	persons**.each** *do |p| puts p.name end* numbers**.each** *do |n| n = n * 2 end*	"Iterating" means doing something for *each* element of the array. Code placed between *do* and *end* determines what is done to each element in the array. The first example prints the name of every person in the array to the console. The second example simply doubles every number of a given array.

1.1.5 Hashes

Keyword

A string is called a **special string** if it does not contain a vowel i.e., (a,e,i,o,u) in the first three characters or last three characters

Hashes associate a *key* to some *value*. You may then retrieve the value based upon its key. This construct is called a *dictionary* in other languages, which is appropriate because you use the key to «look up» a value, as you would look up a definition for a word in a dictionary. Each key must be unique for a given hash but values can be repeated.

Hashes can map from anything to anything! You can map from Strings to Numbers, Strings to Strings, Numbers to Booleans... and you can mix all of those! Although it is common that at least all the keys are of the same class. *Symbols* are especially common as keys. Symbols look like this: symbol. A symbol is a colon followed by some characters. You can think of them as **special strings** that stand for (symbolize) something! We often use symbols because Ruby runs faster when we use symbols instead of strings.

Concept	Usage	Examples	Description
Creating	{key => value}	{:hobby => «programming»} {42 => "answer", "score" => 100, :name => "Tobi"}	You create a hash by surrounding the key-value pairs with curly braces. The arrow always goes from the *key* to the *value* depicting the meaning: *"This key points to this value."*. Key-value pairs are then separated by commas.
Accessing	hash[key]	hash = {:key => "value"} hash[:key] # => "value" hash[foo] # => nil	Accessing an entry in a hash looks a lot like accessing it in an *array*. However with a hash the key can be anything, not just numbers. If you try to access a key that does not exist, the value nil is returned, which is Ruby's way of saying "nothing", because if it doesn't recognize the key it can't return a value for it.

Assigning	hash**[**key**]** = value	hash = {:a => "b"} hash**[**:key**]** = "value" hash # => {:a=>"b", :key=>"value"}	Assigning values to a hash is similar to assigning values to an array. With a hash, the key can be a number or it can be a symbol, string, number... or anything, really!
Deleting	hash**.** **delete**(*key*)	hash = {:a => "b", :b => 10} hash**.** **delete**(*:a*) hash # => {:b=>10}	You can delete a specified key from the hash, so that the key and its value can not be accessed.

Keyword

In computer systems, a framework is often a layered structure indicating what kind of programs can or should be built and how they would interrelate. Some computer system **frameworks** also include actual programs, specify programming interfaces, or offer programming tools for using the frameworks.

1.2 RAILS BASICS

We look at the general structure of a Rails application and the important commands used in the terminal. If you do not have Rails installed yet, there is a well maintained guide by Daniel Kehoe on how to install Rails on different platforms.

example

Running the ruby command launched the Ruby interpreter. The Ruby interpreter read the file you specified and evaluated its contents. It executed the line puts "Hello, World!" by calling the puts function. The string value of Hello, World! Was passed to the function. In this example, the string Hello, World! is also called an argument since it is a value that is passed to a method.

Currently much of the excitement surrounding Ruby can be attributed to a web development framework called Rails – popularly known as 'Ruby on Rails'. While Rails is an **impressive framework**, it is not the be-all and end-all of Ruby. Indeed, if you decide to leap right into Rails development without first mastering Ruby, you may find that you end up with an application that you don't even understand. While the Little Book of Ruby won't cover the special features of Rails, it will give you the grounding you need to understand Rails code and write your own Rails applications.

Installing and Using Ruby with Ruby in Steel

Keyword

A **debugger** is a computer program used by programmers to test and debug a target program.

Ruby in Steel is a Windows-based IDE which comes with an all-in-one installer to install Ruby, Visual Studio, Ruby in Steel .and various other optional packages including Rails

Installing Ruby Yourself

If you are using some other IDE or editor, you will need to download the latest version of Ruby from www.ruby-lang.org. .Be sure to download the binaries

Running Ruby Programs

It is often useful to keep a Command window open in the source directory containing your Ruby program files. Assuming that the Ruby interpreter is correctly pathed on your system, you will then be able to run programs by entering ruby like this:

ruby 1helloworld.rb

If you are using Ruby In Steel you can run the programs in the interactive console by pressing CTRL+F5 or (in some editions) you may run them in the **debugger** by pressing F5.

1.2.1 The Structure of a Rails app

Here is an overview of all the folders of a new Rails application, outlining the purpose of each folder, and describing the most important files.

Name	Description
app	This folder contains your application. Therefore it is the **most important folder** in Ruby on Rails and it is worth digging into its subfolders. See the following rows.
app/assets	Assets basically are your front-end stuff. This folder contains *images* you use on your website, *javascripts* for all your fancy front-end interaction and *stylesheets* for all your CSS making your website absolutely beautiful.

app/ controllers	The controllers subdirectory contains the controllers, which handle the requests from the users. It is often responsible for a single resource type, such as places, users or attendees. Controllers also tie together the *models* and the *views*.
app/helpers	Helpers are used to take care of logic that is needed in the views in order to keep the views clean of logic and reuse that logic in multiple views.
app/mailers	Functionality to send emails goes here.
app/models	The models subdirectory holds the classes that model the business logic of our application. It is concerned with the things our application is about. Often this is data, that is also saved in the database. Examples here are a Person, or a Place class with all their typical behaviour.
app/views	The views subdirectory contains the display templates that will be displayed to the user after a successful request. By default they are written in HTML with embedded ruby (.html.erb). The embedded ruby is used to insert data from the application. It is then converted to HTML and sent to the browser of the user. It has subdirectories for every resource of our application, e.g. places, persons. These subdirectories contain the associated view files. Files starting with an underscore (_) are called *partials*. Those are parts of a view which are reused in other views. A common example is *_form.html.erb*which contains the basic form for a given resource. It is used in the *new* and in the *edit* view since creating something and editing something looks pretty similar.
config	This directory contains the configuration files that your application will need, including your database configuration (in *database.yml*) and the particularly important *routes.rb* which decides how web requests are handled. The *routes.rb* file matches a given URL with the *controller* which will handle the request.
db	Contains a lot of *database* related files. Most importantly the *migrations* subdirectory, containing all your database migration files. Migrations set up your database structure, including the attributes of your models. With migrations you can add new attributes to existing models or create new models. So you could add the *favorite_color* attribute to your Person model so everyone can specify their favorite color.
doc	Contains the documentation you create for your application. Not too important when starting out.
lib	Short for library. Contains code you've developed that is used in your application and may be used elsewhere. For example, this might be code used to get specific information from Facebook.

log	See all the funny stuff that is written in the console where you started the Rails server? It is written to your *development.log*. Logs contain runtime information of your application. If an error happens, it will be recorded here.
public	Contains static files that do not contain Ruby code, such as error pages.
script	By default contains what is executed when you type in the *rails* command. Seldom of importance to beginners.
test	Contains the tests for your application. With tests you make sure that your application actually does what you think it does. This directory might also be called *spec*, if you are using the RSpec gem (an alternative testing framework).
vendor	A folder for software code provided by others ("libraries"). Most often, libraries are provided as *ruby gems* and installed using the *Gemfile*. If code is not available as a ruby gem then you should put it here. This might be the case for jQuery plugins. Probably won't be used that often in the beginning.
Gemfile	A file that specifies a list of gems that are required to run your application. Rails itself is a gem you will find listed in the Gemfile. Ruby gems are self-contained packages of code, more generally called libraries that add functionality or features to your application. If you want to add a new gem to your application, add "**gem** *gem_name*" to your Gemfile, optionally specifying a version number. Save the file and then run *bundle install* to install the gem.
Gemfile.lock	This file specifies the exact versions of all gems you use. Because some gems depend on other gems, Ruby will install all you need automatically. The file also contains specific version numbers. It can be used to make sure that everyone within a team is working with the same versions of gems. The file is auto-generated. *Do not edit this file.*

Keyword

A **command** is a specific instruction given to a computer application to perform some kind of task or function.

1.2.2 Important Rails Commands

Here is a summary of important commands that can be used as you develop your Ruby on Rails app. You must be in the root directory of your project to run any of these **commands** (with the exception of the *rails new* command). The project or application root directory is the folder containing all the subfolders described above (app, config, etc.).

Concept	Usage	Description
Create a new app	**rails new** *name*	Create a new Ruby on Rails application with the given name here. This will give you the basic structure to immediately get started. After this command has successfully run your application is in a folder with the same name you gave the application. You have to *cd* into that folder.
Start the server	**rails server**	You have to start the server in order for your application to respond to your requests. Starting the server might take some time. When it is done, you can access your application underlocalhost:3000 in the browser of your choice. In order to stop the server, go to the console where it is running and press **Ctrl + C**
Scaffolding	**rails generate scaffold** *name attribute:type*	The scaffold command magically generates all the common things needed for a new resource for you! This includes *controllers, models and views*. It also creates the following basic actions: create a new resource, edit a resource, show a resource, and delete a resource. That's all the basics you need. Take this example: **rails generate scaffold** *product name:string price:integer* Now you can create new products, edit them, view them and delete them if you don't need them anymore. Nothing stops you from creating a full fledged web shop now ;-)
Run migrations	**rake db:migrate**	When you add a new migration, for example by creating a new *scaffold*, the migration has to be applied to your database. The command is used to update your database.
Install dependencies	**bundle install**	If you just added a new gem to your Gemfile you should run bundle install to install it. Don't forget to restart your server afterwards!

Check dependencies	**bundle check**	Checks if the dependencies listed in Gemfile are satisfied by currently installed gems
Show existing routes	**rake routes**	Shows complete list of available routes in your application.

1.2.3 ERB: Embedded Ruby

In your views (that is, under app/views in your Rails app) you will find .html.**erb** files. **ERB** stands for **E**mbedded **R**u**B**y. This just means that Rails processes some special .tags in those files and produces HTML output to send back to the user

There are two ERB tags that you need to remember: <%= *ruby_code* %> and <% .*ruby_code* %>. Notice that the difference is the = in the first tag

Tag	Examples	Description
<%= %>	**<%= @product.price %>**	It runs the Ruby code and inserts the result to the HTML at that position. You can put *any kind of Ruby code* between <%= and%>, for instance, <%= 9 * 3 %> will translate to **27** in the page that the user is viewing. However, typically this tag is used to display some data from a model, such as the price of a product, as shown in the example here.
<% %>	**<% if user.admin? %>** Hello Admin! <% end %>	The Ruby code between the delimiters <% and %> is run but the result will **not** be inserted at this point in the HTML. Therefore these tags are most commonly used for control flow structures such as an if statement in the example, or loops.

1.2.4 Editor tips

When you write code you will be using a text editor. Of course each text editor is different and configurable. Here are just some functions and their most general short cuts. All of them work in Sublime Text 2. Your editor may differ!

The shortcuts listed here are for Linux/Windows. On a Mac you will have to replace *Ctrl* with *Cmd*.

Function	Shortcut	Description
Save file	**Ctrl + S**	Saves the currently open file. If it was a new file you may also be asked where to save it.
Undo	**Ctrl + Z**	Undo the last change you made to the current file. Can be applied multiple times in succession to undo multiple changes.
Redo	**Ctrl + Y** *or Ctrl + Shift + Z*	Redo what you just undid with *undo*, can also be done multiple times.
Find in File	**Ctrl + F**	Search for a character sequence within the currently open file. Hit *Enter* to progress to the next match.
Find in all Files	**Ctrl + Shift + F**	Search for a character sequence in all files of the project.
Replace	**Ctrl + H** *or Ctrl + R*	Replace occurrences of the supplied character sequence with the other supplied character sequence. Useful when renaming something.
Copy	**Ctrl + C**	Copy the currently highlighted text into the clipboard.
Cut	**Ctrl + X**	Copy the highlighted text into the clipboard but delete it.
Paste	**Ctrl + V**	Insert whatever currently is in the clipboard (through *Copy* or *Cut*) at the current caret position. Can insert multiple times.
New File	**Ctrl + N**	Create a new empty file.
Search and open file	**Ctrl + P**	Search for a file giving part of its name (*fuzzy search*). Pressing *enter* will open the selected file.
Comment	**Ctrl + /**	Marks the selected text as a comment, which means that it will be ignored. Useful when you want to see how something behaves or looks without a specific section of code being run.

1.3 THE HISTORY OF RUBY

Keyword

Object-oriented programming (OOP) refers to a type of computer programming (software design) in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure.

The Ruby language is 21 years old. Its strong community and adoption by the open source community has kept this language steady and improving. Ruby has changed drastically over the years. It has grown from a young child to the strong adult that it is today. But it didn't get that way overnight. Let's take a look at the life of the Ruby programming language.

(Birth (1993

Ruby was born in 1993, conceived in a discussion between Yukihiro Matsumoto ("Matz") and a colleague. They were discussing the possibility of an object-oriented scripting-language. Matz stated in ruby-talk: 00382 that he knew Perl, but did not like it very much; that it had the smell of a "toy" language. He also discussed that he knew Python, but didn't like it because it wasn't a true **object-oriented programming** language.

Matz wanted a language perfect for his needs:

- Syntactically Simple
- Truly Object-Oriented
- Having Iterators and Closures
- Exception Handling
- Garbage Collection
- Portable

Did You Know?

Ruby 1.8 was released in 2003. This release made large amounts of changes to the agile 10-year-old language.

1.3.1 Toddler Years

Ruby became a toddler (n.: a young child who is just beginning to walk) when Ruby 1.0 was released in December, 1996. Ruby 1.1 shortly followed in August of 1997, and the first stable version of Ruby (1.2) was released in December of 1998.

At this point in time, Ruby was localized to Japan only, but it would soon spread to other parts of the world…

Primary School Years

In 1998, Matz created a simple English homepage for Ruby. However, Ruby was still much localized to Japan. In trying

to further this expansion, the first English language Ruby mailing list, Ruby-Talk, was created. Ruby was beginning to spread beyond Japan.

Ruby-Talk is still very active today, and you can subscribe to it here.

In October of 1999, Yukihiro Matsumoto and Keiju Ishitsuka wrote the first book on the Ruby programming language: *The Object-oriented Scripting Language Ruby*. Ruby was beginning to get very popular in Japan, and spreading rapidly to English-speaking countries.

In 2001, the first English book on Ruby, *Programming Ruby* ("The Pickaxe"), was published in 2001. With this new information on Ruby, many more people were able to learn the language.

Including:

- Duck Typing (looks like a duck, swims like a duck, quacks like a duck: it's a duck)
- Fully Qualified Names (Foo::Bar)
- Native YAML Support
- WEBrick
- StringIO
- open-uri
- PP (Pretty Printer for Hash#inspect)
- ruby -run (UNIX commands for all! ruby -run -e mkdir foo)
- And many other minor features

In 2004, RubyGems was released to the public. Good things started happening next…

Remember

Having looked around and not found a language suited for him, Yukihiro Matsumoto decided to create his own. After spending several months writing an interpreter, Matz finally published the first public version of Ruby (0.95) to various Japanese domestic newsgroups in December, 1995. You can still download the infant version of Ruby here at your own risk.

1.3.2 The Rebellious Teenager

In 2005, Ruby use took off. The reason: **Ruby on Rails**. This new framework changed the history of rapid web development. Ruby had been used in the past to write CGI scripts, but Ruby on Rails took this a step further. Rails has a Model-View-Controller structure that focuses on "convention over configuration", which is great for developing web applications.

People loved it. So much so that, the Ruby community was almost taken over by the Rails framework. Ruby in turn became very popular. In March of 2007, Ruby 1.8.6 was released, with 1.8.7 following in May of 2008. At this point, Ruby was at its peak. Mac OS X even began shipping with it in 2007. At this point, Ruby was 15 years old. Ruby 1.9 (development version) was released in December, 2007, then stabilized 4 years later (2011) as Ruby 1.9.3. Ruby 1.9.3 was the production version of 1.9.2. These versions brought new changes to the language, such as:

- Significant speed improvements
- New methods
- New hash syntax ({ foo: 'bar' })
- RubyGems included
- New Socket API (IPv6 support)
- Several random number generators
- Regular Expression improvements
- File loading performance improvements
- Test::Unit Improvements
- New encoding support
- More string formatting tweaks
- And so much more

Ruby was making the transition from a rebellious teenager to a strong adult as it turned 18 with Ruby 1.9.3.

Remember

Ruby is said to follow the principle of least astonishment (POLA), meaning that the language should behave in such a way as to minimize confusion for experienced users.

Strong Adult

Ruby 2.0.0 was released in February 2013 and brought many stabilizing changes to the language. Among them are:

- More speed improvements
- Refinements (safe monkey patching)
- Keyword arguments
- UTF-8 by default
- New regular expressions engine
- Optimized garbage collection
- The addition of built-in syntax documentation (ri ruby:syntax)

Unlike 1.9.x, which broke numerous gems with its changes, 2.0.0 was almost completely backwards compatible with 1.9.3. In addition, Heroku, one of the leading Ruby/Rails hosts upgraded to 2.0.0 quickly, causing earlier than usual adoption by new and existing projects. The Ruby language was (and is) in its golden age.

Ruby 2.1.0 was released on Christmas day of 2013. It brought several minor changes to the language. But the biggest news of 2.1.0 was semantic versioning, a way to properly version a project without breaking dependencies by accident. Ruby 2.1.1 was released on Ruby's 21st birthday (February 24, 2014). Ruby is now legally allowed to drink in the US (not that we'd want it to). This version was mainly speed improvements and bugfixes. Shortly after 2.1.1, Ruby 2.1.2 was released in May of 2014. 2.1.2 consists of more bugfixes and is the current stable version of Ruby.

1.3.3 The Future

Ruby is a great language. Matz wanted a programming language that suited his needs, so he built one. This is an inspiring story of software development: if you can't find something that you like, program it yourself. From 0.95 to 2.1.2, Ruby has struck the awe of those who wished to program the way they wanted, not the way the machine wanted.

We can't know the future of the Ruby language, but we can predict it based on the past. We believe that the Ruby language, and its fantastic community will continue furthering the language above and beyond what others think is possible, and projects built using it will do the same.

ROLE MODEL

YUKIHIRO MATSUMOTO: JAPANESE COMPUTER SCIENTIST AND SOFTWARE PROGRAMMER

"Matsumoto Yukihiro, a.k.a. Matz, is a Japanese computer scientist and software programmer best known as the chief designer of the Ruby programming language and its reference implementation, Matz's Ruby Interpreter (MRI).

As of 2011, Matsumoto is the Chief Architect of Ruby at Heroku, an online cloud platform-as-a-service in San Francisco. He is a fellow of Rakuten Institute of Technology, a research and development organisation in Rakuten Inc. He was appointed to the role of technical advisor for VASILY, Inc. starting in June 2014.

Early life

Born in Osaka Prefecture, Japan, he was raised in Tottori Prefecture from the age of four. According to an interview conducted by Japan Inc., he was a self-taught programmer until the end of high school. He graduated with an information science degree from University of Tsukuba, where he was a member of Ikuo Nakata's research lab on programming languages and compilers.

Work

He works for the Japanese open source company, netlab.jp. Matsumoto is known as one of the open source evangelists in Japan. He has released several open source products, including cmail, the Emacs-based mail user agent, written entirely in Emacs Lisp. Ruby is his first piece of software that has become known outside Japan.

Ruby

Matsumoto released the first version of the Ruby programming language on 21 December 1995. He still leads the development of the language's reference implementation, MRI (for Matz's Ruby Interpreter).

MRuby

In April 2012, Matsumoto open-sourced his work on a new implementation of Ruby called mruby. It is a minimal implementation based on his virtual machine, called ritevm, and is designed to allow software developers to embed Ruby in other programs while keeping memory footprint small and performance optimised.

streem

In December 2014, Matsumoto open-sourced his work on a new scripting language called streem, a concurrent language based on a programming model similar to shell, with influences from Ruby, Erlang and other functional programming languages.

Treasure Data

Matsumoto has been listed as an investor for Treasure Data; many of the company's programs such as Fluentd use Ruby as their primary language.

Recognition

Matsumoto received the 2011 Award for the Advancement of Free Software from the Free Software Foundation (FSF) at the 2012 LibrePlanet conference at the University of Massachusetts Boston in Boston

Personal life

Matsumoto is married and has four children. He is a member of The Church of Jesus Christ of Latter-day Saints, did standard service as a missionary and is now a counselor in the bishopric in his church ward.

SUMMARY

- Ruby is an interpreted, high-level, general-purpose programming language. It was designed and developed in the mid-1990s by Yukihiro "Matz" Matsumoto in Japan.
- Ruby inherits some features of languages like Smalltalk, Perl, and Python.
- Ruby provides support for multiple programming paradigms, dynamic type system, and automatic memory management.
- Ruby ignores everything that is marked as a comment. It does not try to execute it. Comments are just there for you as information.
- Numbers are what you would expect them to be, normal numbers that you use to perform basic math operations.
- Ruby became a toddler (n.: a young child who is just beginning to walk) when Ruby 1.0 was released in December, 1996.
- Ruby 2.0.0 was released in February 2013 and brought many stabilizing changes to the language.
- Unlike 1.9.x, which broke numerous gems with its changes, 2.0.0 was almost completely backwards compatible with 1.9.3.
- In December 2014, Matsumoto open-sourced his work on a new scripting language called streem, a concurrent language based on a programming model similar to shell, with influences from Ruby, Erlang and other functional programming languages.

KNOWLEDGE CHECK

1. **Ruby was developed in mid**
   a. 1960s
   b. 1970s
   c. 1980s
   d. 1990s
2. **Programming language which is an open-source, object oriented programming language with simple syntax similar to Perl and Python is**
   a. C
   b. C++
   c. Java
   d. Ruby
3. **A web application development framework written in Ruby language is**
   a. Rail
   b. Ada
   c. Pascal
   d. Java
4. **Ruby on Rails was used to build user's interface of**
   a. Twitter
   b. Facebook
   c. Wikipedia
   d. Google
5. **Rails' application framework is called .....**
   a. ActionPack
   b. ActiveRecord
   c. a web page
   d. an object
6. **When Whitespace characters such as spaces and tabs cannot ignored in Ruby code?**
   a. While using strings
   b. While using integer
   c. while using float value
   d. All of the above

7. Which character is used to give comment in ruby?

a. !

b. @

c. #

d. $

REVIEW QUESTIONS

1. Define the Ruby code.
2. Focus on frameworks for Ruby programming.
3. Define html.**erb** files.
4. What will the following code return?

   true and 0 && !**nil and** 3 > 2
5. Explain the strings.

Check Your Result

1. (d) 2. (d) 3. (a) 4. (a) 5. (a)
6. (a) 7. (c)

REFERENCES

1. Carlson, Lucas and Richardson, L. (2006) Ruby Cookbook. O'Reilly Media. 2006.
2. Cockburn, Alistair (2007) Agile Software Development: The Cooperative Game. 2nd Edition. Pearson Education Inc.
3. Cooper, S., Dann, W. and Pausch, R. (2000). Alice: a 3-D tool for introductory programming concepts, Proceedings of the fifth annual CCSC northeastern conference on The journal of computing in small colleges. Ramapo College of New Jersey, Mahwah, New Jersey, United States.
4. Cooper, S., Dann, W. and Pausch, R. (2000). Alice: a 3-D tool for introductory programming concepts, Proceedings of the fifth annual CCSC northeastern conference on the journal of computing in small colleges. Ramapo College of New Jersey, Mahwah, New Jersey, United States.
5. Guzdial, Mark (2000) Squeak: Object-Oriented Design with Multimedia Applications. Prentice Hall.
6. Guzdial, Mark and Kim Rose, Kim (Eds.) (2002) Squeak: Open Personal Computing and Multimedia. Prentice Hall.
7. Guzdial, Mark and Kim Rose, Kim (Eds.) (2002) Squeak: Open Personal Computing and Multimedia. Prentice Hall.
8. Kelleher, Caitlin and Pausch, R. (2005) Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers, ACM Computing Surveys, 37(2), 83–137.
9. Kortenkamp, U., Modrow, E., Oldenburg, R., Poloczek, J. and Rabel, M. (2009) Objektorientierte Modellierung – aber wann und wie?, LOG IN Heft Nr. 160/161, 22–28.
10. Maloney, J., L. Burd, et al. (2005) Scratch: A Sneak Preview. International Conference on Creating, Connecting, and Collaborating through Computing., Kyoto, Japan.
11. Perrotta, P. (2010) Metaprogramming Ruby. Pragmatic Programmers
12. Reas, C. and Fry, B. (2007) Processing: A Programming Handbook for Visual Designers and Artists. MIT Press.
13. Smith, D.A., Kay, A., Raab, A. & Reed, D. (2003) Croquet – a collaboration system architecture. First Conference on Creating, Connecting and Collaborating through Computing: 2.
14. Thomas, D. (2009). Programming Ruby - The Programmatic Programmer's Guide. Pragmatic Programmers.

CHAPTER

2

WORKING WITH STRINGS, OBJECTS, AND VARIABLES

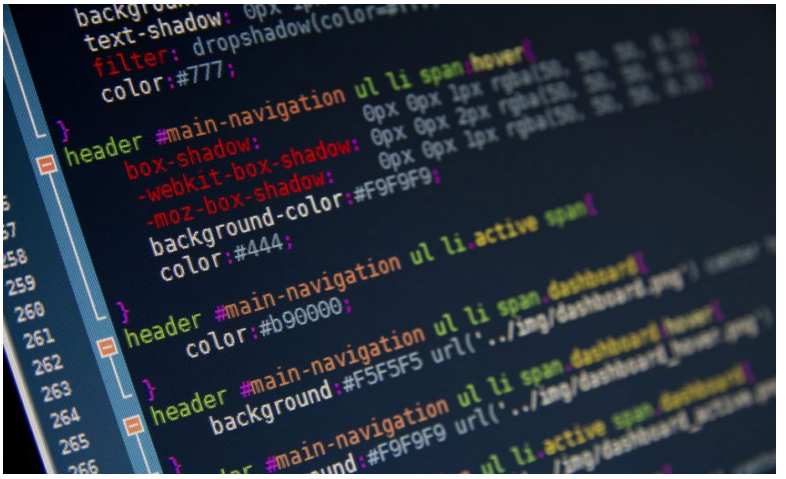
"Web servers are written in C, and if they're not, they're written in Java or C++, which are C derivatives, or Python or Ruby, which are implemented in C."

–Rob Pike

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Discuss about Ruby – strings
2. Explain objects and methods
3. Describe variable in Ruby



INTRODUCTION

In Ruby, string is a sequence of one or more characters. It may consist of numbers, letters, or symbols. Strings are the objects, and apart from other languages, strings are

mutable, i.e. strings can be changed in place instead of creating new strings. String's object holds and manipulates an arbitrary sequence of the bytes that commonly represents a sequence of characters.

Ruby is a perfect object oriented programming language. The features of the object-oriented programming language include:

- Data encapsulation
- Data abstraction
- Polymorphism
- Inheritance

An object-oriented program involves classes and objects. A class is the blueprint from which individual objects are created. In object-oriented terms, we say that your bicycle is an instance of the class of objects known as bicycles.

Take the example of any vehicle. It comprises wheels, horsepower, and fuel or gas tank capacity. These characteristics form the data members of the class Vehicle. You can differentiate one vehicle from the other with the help of these characteristics

A vehicle can also have certain functions, such as halting, driving, and speeding. Even these functions form the data members of the class Vehicle. You can, therefore, define a class as a combination of characteristics and functions.

Ruby provides four types of variables:

Local Variables – Local variables are the variables that are defined in a method. Local variables are not available outside the method. Local variables begin with a lowercase letter or _.

Instance Variables – Instance variables are available across methods for any particular instance or object. That means that instance variables change from object to object. Instance variables are preceded by the at sign (@) followed by the variable name.

Class Variables – Class variables are available across different objects. A class variable belongs to the class and is a characteristic of a class. They are preceded by the sign @@ and are followed by the variable name.

Global Variables – Class variables are not available across classes. If you want to have a single variable, which is available across classes, you need to define a global variable. The global variables are always preceded by the dollar sign ($)

Objects are instances of the class. You will now learn how to create objects of a class in Ruby. You can create objects in Ruby by using the method new of the class.

The method new is a unique type of method, which is predefined in the Ruby library. The new method belongs to the class methods.

2.1 RUBY - STRINGS

A String object in Ruby holds and manipulates an arbitrary sequence of one or more bytes, typically representing characters that represent human language.

The simplest string literals are enclosed in single quotes (the apostrophe character). The text within the quote marks is the value of the string −

'This is a simple Ruby string literal'

If you need to place an apostrophe within a single-quoted string literal, precede it with a backslash, so that the Ruby interpreter does not think that it terminates the string −

'Won\'t you read O\'Reilly\'s book?'

The backslash also works to escape another backslash, so that the second backslash is not itself interpreted as an escape character.

2.1.1 Concatenation

One of the most basic things you will find yourself doing with strings is concatenation. This is where you join two strings together. There are a few different ways to do this in Ruby.

Firstly you can join two strings together using the + operator:

```
greeting = "hello " + "world"
=> "hello world"
```

If you already have a string you can append another string using the ``` operator:

```
name = "Philip "<br></br>
=> "Philip "```

name << "Brown"
 => "Philip Brown"
```

You can also multiple a string by an integer to return three copies of that string as a new string:

```
"yo" * 3
=> "yoyoyo"
```

2.1.2 Case

We can convert a string into capital case by calling the capitalize method:

```
"philip".capitalize
=> "Philip"
```

When you call a method on a string, such as capitalize, you will be returned a new string.

For example, try the following in IRB:

```
name = "philip"
=> "philip"

name.capitalize
=> "Philip"
```

Now if you puts the name variable, what do you think it will be?

```
puts name
=> "philip"
```

When you call a method on a String object it creates a new string, rather than changing the existing one.

To modify the original instance we can call the same method but with a !:

```
name.capitalize!
=> "Philip"

name
=> "Philip"
```

If you want to convert a string to lowercase you can call the downcase method:

```
"HELLO WORLD".downcase
=> "hello world"
```

Alternatively you can covert a string to **uppercase** using the upcase method:

```
"hello world".upcase
=> "HELLO WORLD"
```

Keyword

Uppercase characters are capital letters; lowercase characters are small letters.

2.1.3 Length

If you need to count the number of characters in a string you can call use the length method:

```
"abcdefghi".length
=> 9
```

2.1.4 Strip

Whenever you accept input from a user you should make sure they haven't included any whitespace. You can remove white space using the strip method:

```
"philip brown ".strip
=> "philip brown"
```

If you want to modify the current instance of the string rather than create a new string, you should call the method with a !:

```
name = " philip brown "
=> " philip brown "

name.strip!
=> "philip brown"

name
=> "philip brown"
```

Alternatively you can only remove whitespace from the left or the right of the string by calling lstrip or rstrip.

2.2 OBJECTS AND METHODS

"Everything in Ruby is an Object" is something you'll hear rather frequently. "Pretty much everything else is a method" could also be said. The goal here is for you to see the Matrix... that everything in Ruby is an Object, every object has a class, and being a part of that class gives the object lots of cool methods that it can use to ask questions or do things. Being incredibly object-oriented gives Ruby lots of power and makes your life easier.

Keyword

Black box is a device, system or object which can be viewed in terms of its inputs and outputs (or transfer characteristics), without any knowledge of its internal workings.

Think of every "thing" in Ruby as a having more than meets the eye. The number 12 is more than just a number... It's an **object** and Ruby lets you do all kinds of interesting things to it like adding and multiplying and asking it questions like > 12.class or > 12+3

Ruby gives all objects a bunch of neat **methods**. If you ever want to know what an object's methods are, just use the #methods method! Asking > 12345.methods in IRB will return a whole bunch of methods that you can try out on the number 12345. You'll also see that the basic operators like + and - and / are all methods too (they're included in that list). You can call them using the dot > 1.+2 like any other method or, luckily for you, Ruby made some special shortcuts for them so you can just use them as you have been: > 1+2

Some methods ask true/false questions, and are usually named with a question mark at the end like #is_a?, which asks whether an object is a type of something else, e.g. 1.is_a?Integer returns true while "hihi".is_a?Integer returns false You'll get used to that naming convention. Methods like #is_a?, which tell you something about the object itself, are called **Reflection Methods** (as in, "the object quietly reflected on its nature and told me that it is indeed an Integer"). ::class was another one we saw, where the object will tell you what class it is.

What is a method? A method is just a function or a **black box**. You put the thing on the left in, and it spits something out on the right. *Every method returns something*, even if it's just nil.

Some methods are more useful for their **Side Effects** than the thing they actually return, like #puts That's why when you say > puts "hi" in IRB, you'll see a little => nil down below... the method prints out your string as a "side effect" and then returns nil after it's done. When you write your own methods, if you forget to think about the return statement, sometimes you'll get some wierd behavior so always think about what's going in and what's coming out of a method.

Methods can take **inputs** too, which are included in parentheses to the right of the method name (though they can be omitted, as you do with > puts("hi") becoming > puts "hi".. it's okay to be lazy, as long as you know what you're doing). Going back to the addition example, > 1+2==3 is asking whether 1+2 will equal 3 (it returns => true), but it can more explicitly be written > 1.+(2).==(3) So, in this case, you can see there's more going on than meets the eye at first.

That example also shows **Method Chaining**, which is when you stick a bunch of methods onto each other. It behaves like you'd expect -- evaluate the thing on the left first, pass whatever it returns to the method on the right and keep going. So > 1+2==3 first evaluates 1+2 to be 3 and then evaluates 3==3 which is true This is great because it lets you take what would normally be many lines of code and combine them into one elegant expression.

Bang Methods are finished with an exclamation point ! like #sort!, and they actually modify the original object. The exclamation point lets you know you're in dangerous territory.

Bang methods save over the original object (they are "destructive"):

```
> my_numbers = [1,5,3,2]
=> [1, 5, 3, 2]
> my_numbers.sort
=> [1, 2, 3, 5]
> my_numbers
=> [1, 5, 3, 2]            # still unsorted
> my_numbers.sort!
=> [1, 2, 3, 5]
> my_numbers
```

Remember

When you run a normal method in IRB, it will output whatever the method returns but it will preserve the original object.

=> [1, 2, 3, 5] # overwrote the original my_numbers object!

Methods ending with a question mark ? return true or false.

Let's answer the question, "Where did all those methods come from?" **Classes** are like umbrellas that let us give an object general behaviors just based on what it is. An object is an instance of a class -- you (yes, you) are an instance of the Person class. There are lots of behaviors (methods) that you can do just by virtue of being a Person... #laugh, #jump, speak("hello") This is really useful in programming because you often need to create lots of instances of something and it's silly to have to rewrite all the methods you want all of them to have anyway, so you write them at the class level and all the instances get to use them.

Instances of a class get to **inherit** the behaviors of that class. Inheritance works for classes too! Your class Person has lots of methods but many of them are inherited just by virtue of you also being a Mammal or even just a LivingThing You get to use all the methods of your ancestor classes

An interesting exercise to try in Ruby is to use the method ::superclass to ask a class what its parent is. If you just keep on going and going, you'll see that everything eventually inherits from BasicObject, which originates most of the methods you have access to in the original object:

> 1.class.superclass.superclass.superclass

=> BasicObject

> BasicObject.methods

=> # giant list of methods

Random Note: Running the ::methods method on a class only returns the class methods, whereas ::instance_methods will return all methods available to any instance of that class (so String.methods will return a list of class methods, while "hello". methodswill return a longer list that is the same as String.instance_methods).

Other Random Note: Use object_id to see an object's id, and this can be useful if you're running into odd errors where you thought you were modifying and object but it's not changing. If you debug and look at the id's along the way, you may find that you're actually only modifying a COPY of that object.

To **Write Your Own Methods**, just use the syntax def methodname(argument1, argument2), though the parentheses around the arguments are optional. The method will return ("spit out") either whatever follows the return statement or the result of the last piece of code that was evaluated (an **Implicit Return** statement). You call the inputs by whatever name you defined them at the top.

You can write methods in IRB... it will let you use multiple lines if it detects that you have unfinished business (a def without an end or unclosed parentheses):

> def speak(words)

```
>    puts words
>    return true
> end
=> nil                # ignore this
> speak("hello!")
hello!
=> true
```

What if you want to assume that the input to a method is a particular value if there hasn't been any supplied? That's easy, just specify the **Default Input** by assigning it to something where it's listed as an input:

```
> def speak(words="shhhhh")
>     puts words
> end          # implicitly returns what puts returns... nil!
=> nil         # ignore this
> speak        # no input
shhhhh
=> nil
```

2.2.1 Objects and Attributes

If the object that you're attempting to bind is neither Serializable nor Referenceable, then you can still bind it if it has attributes, provided that binding DirContextapi objects is a feature supported by the underlying **service provider**. Sun's LDAP service provider supports this feature.

Keyword

Service provider can be an organizational sub-unit, it is usually a third party or outsourced supplier, including telecom-munications service providers (TSPs), application service providers (ASPs), storage service providers (SSPs), and internet service providers (ISPs).

Interoperability

Binding DirContext objects is especially useful for interoperability with non-Java applications. An object is represented by a set of attributes, which can be read and used by non-Java applications from the directory. The same attributes can also be read and interpreted by a JNDI service provider, which, in conjunction with an object factory, converts them into a Java object.

For example, an object might have, as some of its attribute values, URLs that the JNDI service could use to generate an

instance of a Java object that the application could use. These same URLs could be used also by non-Java applications.

Binding an Object by Using Its Attributes

The following example shows a Drink class that implements the DirContext interface. Most DirContext methods are not used by this example and so simply throw anOperationNotSupportedException[api].

```
 public class Drink implements DirContext {
String type;
Attributes myAttrs;

public Drink(String d) {
    type = d;
    myAttrs = new BasicAttributes(true);  // Case ignore
    Attribute oc = new BasicAttribute("objectclass");
    oc.add("extensibleObject");
    oc.add("top");

    myAttrs.put(oc);
    myAttrs.put("drinkType", d);
}
    public Attributes getAttributes(String name) throws NamingException {
    if (! name.equals("")) {
        throw new NameNotFoundException();
    }
    return (Attributes)myAttrs.clone();
}

public String toString() {
    return type;
}
...
}
```

The Drink class contains the "objectclass" and "drinkType" attributes. The "objectclass" attribute contains two values: "top" and "extensibleObject". The "drinkType" attribute is set by using the string passed to the Drink constructor. For example, if the instance was created by using new Drink("water"), then its "drinkType" attribute will have the value "water".

The following example creates an instance of Drink and invokes Context.bind()[api] to add it to the directory.

```
// Create the object to be bound
Drink dr = new Drink("water");
// Perform the bind
ctx.bind("cn=favDrink", dr);
```

When the object is bound, its attributes are extracted and stored in the **directory**.

When that object is subsequently looked up from the directory, its corresponding object factory will be used to return an instance of the object. The object factory is identified by theContext.OBJECT_FACTORIES[api] environment property when the initial context for reading the object is created.

```
Hashtable env = ...;
// Add property so that the object factory can be found
env.put(Context.OBJECT_FACTORIES, "DrinkFactory");

// Create the initial context
DirContext ctx = new InitialDirContext(env);

// Read back the object
Drink dr2 = (Drink) ctx.lookup("cn=favDrink");
System.out.println(dr2);
```

This produces the following output, "water", produced by Drink.toString().

```
# java DirObj
water
```

From the perspective of the application using the JNDI, it is dealing only with bind() and lookup(). The service provider

Keyword

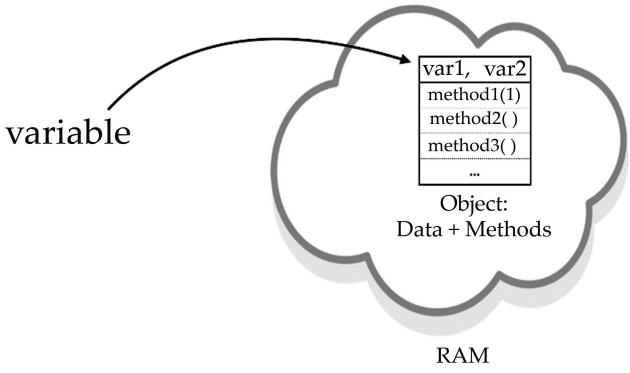
Directory is a file system cataloging structure which contains references to other computer files, and possibly other directories.

takes care of getting the attributes from the object and converting them to/from the actual object itself.

Note that you can store a DirContext object in the directory only if the underlying service provider supports that.

2.3 VARIABLE IN RUBY

Variable is a symbol or name that stands for a value. Variables locate in memory locations and are used to store values such as numeric values, characters, character strings, or memory addresses so that they can be used in any part of the program.



We have discussed the following types of variables and constants supported by Ruby :

- Local Variables
- Instance Variables
- Class Variables
- Global Variables
- Constants

2.3.1 Local Variable

A local variable name must start with a lowercase letter (a-z) or underscore (_) with the eight-bit set.

Local Variable Scope

A local variable is only accessible from within the block of its initialization. See the following example:

In the following example, there are three local variables called color. One is assigned the value, 'Red' within the 'main scope' of the program; two others are assigned integers within the scope of two separate methods. As each local variable has a different scope, the assignments have no affect on the other local variables with the same name in different scopes. We have verified it by calling the methods at the end of the program.:

Example:

```
color = "Red"
def method1
color = 192
puts("Color Value in method1 : ",color)
end

def method2
color = 255
puts("Color Value method2: ",color)
end

method1
method2
method1
puts("Color Value outside methods : "+color)
```

Copy

Output:

```
H:\>ruby abc.rb
Color Value in method1 :
192
Color Value method2:
255
Color Value in method1 :
192
Color Value outside methods : Red
```

Remember

Local variable names must begin with either an underscore or a lower case letter.

Local Variables and Methods

In Ruby, local variable names and method names are nearly identical. If you have not assigned to one of these ambiguous names ruby will assume you wish to call a method. Once you have assigned to the name ruby will assume you wish to reference a local variable.

The local variable is created when the parser encounters the assignment, not when the assignment occurs:

```
a = 0 if false # does not assign to a
p local_variables # prints [:a]
p a # prints nil
```

Copy

```
def big_calculation
  42 # pretend this takes a long time
end
big_calculation = big_calculation()
```

Copy

Now any reference to big_calculation is considered a local variable and will be cached. To call the method, use self.big_calculation.

Keyword

Parenthesis is a tall, curvy punctuation mark used to set off material that isn't fundamental to the main topic, like an afterthought or an aside (or a funny joke).

You can force a method call by using empty argument **parentheses** as shown above or by using an explicit receiver like self. Using an explicit receiver may raise a NameError if the method's visibility is not public.

Another commonly confusing case is when using a modifier if:

```
p a if a = 0.zero?
```

Copy

Rather than printing "true" you receive a NameError, "undefined local variable or method `a'". Since ruby parses the bare a left of the if first and has not yet seen an assignment to it assumes you wish to call a method. Ruby then sees the assignment to a and will assume you are referencing a local method.

The confusion comes from the out-of-order execution of the expression. First, the local variable is assigned to then you attempt to call a nonexistent method.

2.3.2 Instance Variables

Instance variables are shared across all methods for the same object. The instance variable name must start with '@' character, otherwise instance variable names follow the rules as a local variable name.

Syntax:

@insvar

In the following example, Student.new creates a new object - an instance of the class Student. The instance variables @student_id and @student_name.

```
class Student
  def initialize(student_id, student_name)
    @student_id = student_id
    @student_name = student_name
  end

  def show
    puts "Student Name and ID : "
    puts(@student_id, @student_name)
  end
end
Student.new(1, "Sara").show
Student.new(2, "Raju").show
```

Copy

Output:

```
Student Name and ID :
1
Sara
Student Name and ID :
2
Raju
```

Note : An uninitialized instance variable has a value of nil.

2.3.3 Class Variables

Class variables are shared between a class, its subclasses, and its instances.

A class variable must start with a @@ (two "at" signs). The rest of the name follows the same rules as instance variables.

Syntax:

@@classvar

The following example shows that all classes change the same variable. Class variables behave like global variables which are visible only in the inheritance tree. Because Ruby resolves variables by looking up the inheritance tree first, this can cause problems if two subclasses both add a class variable with the same name.

Example:

```
class School
    @@no_off_students = 650
    end

    # class V sub-class of School
    class V < School
    @@no_off_students = 75
    end
    # class VI sub-class of School
    class VI < School
    @@no_off_students = 80
    end
    puts School.class_eval("@@no_off_students")
    puts V.class_eval("@@no_off_students")
    puts VI.class_eval("@@no_off_students")
```

Copy

Output:

```
80
80
80
```

Remember

Class variables must be initialized at creation time.

Using the class variable @@no_of_customers, you can determine the number of objects that are being created. This enables in deriving the number of customers.


```
class Customer

@@no_of_customers = 0

end
```

2.3.4 Global Variables

In Ruby a global variables start with a $ (dollar sign). The rest of the name follows the same rules as instance variables. Global variables are accessible everywhere.

Syntax:

$globalvar

Example:

```
$global = 0

class C
  puts "in a class: #{$global}"

  def my_method
    puts "in a method: #{$global}"

    $global = $global + 1
    $other_global = 3
  end
end

C.new.my_method

puts "at top-level, $global: #{$global}, $other_global: #{$other_global}"
```

Copy

Output:

If we want to display floating point numbers we need to use %f. We can specify the number of decimal places we want like this: %0.2f.

in a class: 0

in a method: 0

at top-level, $global: 1, $other_global: 3

An uninitialized global variable has a value of nil.

Ruby has some special globals that behave differently depending on context such as the regular expression match variables or that have a side-effect when assigned to.

2.3.5 Ruby Constants

Keyword

Initialization is the assignment of an initial value for a data object or variable.

A variable whose name begins with an uppercase letter (A-Z) is a constant. A constant can be reassigned a value after its **initialization**, but doing so will generate a warning. Every class is a constant.

Trying to access an uninitialized constant raises the NameError exception.

Syntax:

ABCD

Example :

```
class Student
 NO_Students = 800
end
puts  'No of students in the school : ', Student::NO_Students
```

Copy

Here NO_Students is the constant.

Output:

No of students in the school :

800

Table 1: Ruby- Pseudo Variables

Name	Description
self	Execution context of the current method.
nil	Expresses nothing. nil also is considered to be false, and every other value is considered to be true in Ruby.

true	Expresses true.
false	Expresses false.
$1, $2 ... $9	These are contents of capturing groups for regular expression matches. They are local to the current thread and stack frame!

Table 2: Ruby- Pre-defined Variables

Name	Description
$!	The exception information message set by the last 'raise' (last exception thrown).
$@	Array of the backtrace of the last exception thrown.
$&	The string matched by the last successful pattern match in this scope.
$`	The string to the left of the last successful match.
$'	The string to the right of the last successful match.
$+	The last group of the last successful match.
$1 to $9	The Nth group of the last successful regexp match.
$~	The information about the last match in the current scope.
$=	The flag for case insensitive, nil by default (deprecated).
$/	The input record separator, newline by default.
$\	The output record separator for the print and IO#write. Default is nil.
$,	The output field separator for the print and Array#join.
$;	The default separator for String#split.
$.	The current input line number of the last file that was read.
$<	An object that provides access to the concatenation of the contents of all the files given as command-line arguments, or $stdin (in the case where there are no arguments). Read only.
$FILENAME	Current input file from $<. Same as $<.filename.
$>	The destination of output for Kernel.print and Kernel.printf. The default value is $stdout.
$_	The last input line of the string by gets or readline.
$0	Contains the name of the script being executed. Maybe assignable.
$*	Command line arguments given for the script. Also known as ARGV
$$	The process number of the Ruby running this script.
$?	The status of the last executed child process.
$:	Load path for scripts and binary modules by load or require.
$"	The array contains the module names loaded by requiring.
$stderr	The current standard error output.

Did You Know?

The first public release of Ruby 0.95 was announced on Japanese domestic newsgroups on December 21, 1995. Subsequently, three more versions of Ruby were released in two days. The release coincided with the launch of the Japanese-language ruby-list mailing list, which was the first mailing list for the new language.

$stdin	The current standard input.
$stdout	The current standard output.
$-d	The status of the -d switch. Assignable.
$-K	Character encoding of the source code.
$-v	The verbose flag, which is set by the -v switch.
$-a	True if option -a (“autosplit” mode) is set. Read-only variable.
$-i	If an in-place-edit mode is set, this variable holds the extension, otherwise nil.
$-l	True if option -l is set (“line-ending processing” is on). Read-only variable.
$-p	True if option -p is set (“loop” mode is on). Read-only variable.
$-w	True if option -w is set.

Table 3: Ruby- Pre-defined Constants

Name	Description
__FILE__	Current file.
__LINE__	Current line.
__dir__	Current directory.

CASE STUDY

RUBY CLASS

For your case study, you will create a Ruby Class called Customer and you will declare two methods −

display_details − This method will display the details of the customer.

total_no_of_customers − This method will display the total number of customers created in the system.

```
#!/usr/bin/ruby
class Customer
   @@no_of_customers = 0
   def initialize(id, name, addr)
      @cust_id = id
      @cust_name = name
      @cust_addr = addr
   end
   def display_details()
      puts “Customer id #@cust_id”
      puts “Customer name #@cust_name”
      puts “Customer address #@cust_addr”
   end
   def total_no_of_customers()
      @@no_of_customers += 1
      puts “Total number of customers: #@@no_of_customers”
   end
end
```

The *display_details* method contains three *puts* statements, displaying the Customer ID, the Customer name, and the Customer address. The puts statement will display the text Customer id followed by the value of the variable @cust_id in a single line as follows −

```
puts “Customer id #@cust_id”
```

When you want to display the text and the value of the instance variable in a single line, you need to precede the variable name with the hash symbol (#) in the puts statement. The text and the instance variable along with the hash symbol (#)

should be enclosed in double quotation marks.

The second method, total_no_of_customers, is a method that contains the class variable @@no_of_customers. The expression @@no_of_ customers+=1 adds 1 to the variable no_of_customers each time the method total_no_of_customers is called. In this way, you will always have the total number of customers in the class variable.

Now, create two customers as follows –

```
cust1 = Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2 = Customer.new("2", "Poul", "New Empire road, Khandala")
```

Here, we create two objects of the Customer class as cust1 and cust2 and pass the necessary parameters with the new method. The initialize method is invoked, and the necessary properties of the object are initialized.

Once the objects are created, you need to call the methods of the class by using the two objects. If you want to call a method or any data member, you write the following –

```
cust1.display_details()
cust1.total_no_of_customers()
```

The object name should always be followed by a dot, which is in turn followed by the method name or any data member. We have seen how to call the two methods by using the cust1 object. Using the cust2 object, you can call both methods as shown below –

```
cust2.display_details()
cust2.total_no_of_customers()
```

Save and Execute the Code

Now, put all this source code in the main.rb file as follows –

```
#!/usr/bin/ruby
class Customer
   @@no_of_customers = 0
   def initialize(id, name, addr)
      @cust_id = id
      @cust_name = name
      @cust_addr = addr
   end
   def display_details()
      puts "Customer id #@cust_id"
```

```
      puts "Customer name #@cust_name"
      puts "Customer address #@cust_addr"
   end
   def total_no_of_customers()
      @@no_of_customers += 1
      puts "Total number of customers: #@@no_of_customers"
   end
end

# Create Objects
cust1 = Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2 = Customer.new("2", "Poul", "New Empire road, Khandala")

# Call Methods
cust1.display_details()
cust1.total_no_of_customers()
cust2.display_details()
cust2.total_no_of_customers()
```

Now, run this program as follows −

```
$ ruby main.rb
```

This will produce the following result −

```
Customer id 1
Customer name John
Customer address Wisdom Apartments, Ludhiya
Total number of customers: 1
Customer id 2
Customer name Poul
Customer address New Empire road, Khandala
Total number of customers: 2
```

SUMMARY

- In Ruby, string is a sequence of one or more characters. It may consist of numbers, letters, or symbols.
- String's object holds and manipulates an arbitrary sequence of the bytes that commonly represents a sequence of characters.
- Ruby belongs to the family of dynamic languages. Unlike strongly typed languages like Java, C or Pascal, dynamic languages do not declare a variable to be of certain data type.
- Bang Methods are finished with an exclamation point ! like #sort!, and they actually modify the original object.
- Binding DirContext objects is especially useful for interoperability with non-Java applications.
- Variables locate in memory locations and are used to store values such as numeric values, characters, character strings, or memory addresses so that they can be used in any part of the program.
- Class variables are shared between a class, its subclasses, and its instances.
- A variable whose name begins with an uppercase letter (A-Z) is a constant. A constant can be reassigned a value after its initialization, but doing so will generate a warning. Every class is a constant.

KNOWLEDGE CHECK

1. **Objects of which class does the integer from the range -2^30 to 2^(30-1) belong to?**
   a. Bignum
   b. Octal
   c. Fixnum
   d. Binary
2. **What is the sequence of ruby strings?**
   a. 16-bit bytes
   b. 8-bit bytes
   c. 10-bit bytes
   d. None of the mentioned
3. **What is the output of the given code?**

```
my_string=Ruby
puts(my_string)
```

   a. Ruby
   b. Nil
   c. Error
   d. None of the mentioned
4. **Which of the following is not a valid datatype?**
   a. Float
   b. Integer
   c. Binary
   d. Timedate
5. **What will any variable evaluate to if it is of Boolean data type?**
   a. True
   b. Nil
   c. False
   d. Either True or False
6. **..... syntax matches with the Ruby's syntax.**
   a. Java
   b. Perl
   c. PHP

d. None of above

7. Which of the following type of comments are valid in ruby?

a. Single line

b. Multiple line

c. Both single and multiple line

d. None of above

REVIEW QUESTIONS

1. What is strings? Explain in detail.
2. Define concatenation. Describe with suitable example.
3. Explain how you define an instance variable, global variable and class variable in Ruby?
4. How does a symbol differ from a string?
5. What is the difference between a symbol and a variable in Ruby?

Check Your Result

1. (c) 2. (b) 3. (c) 4. (d) 5. (d)
6. (b) 7. (c)

REFERENCES

1. A. Kalyanpur, D. Pastor, S. Battle, and J. Padget. Automatic mapping of owl ontologies into java. In SEKE. 2004.
2. Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D.S., Smith, K., 2011. Cython: The Best of Both Worlds. Comput. Sci. Eng. 13, 31–39. https://doi.org/10.1109/MCSE.2010.118
3. Catanio, J.J., 2018. Leave the Features: Take the Cannoli. California Polytechnic State University, San Luis Obispo, California. https://doi.org/10.15368/theses.2018.25
4. D. Schwabe, D. Brauner, D. A. Nunes, and G. Mamede. Hypersd: a semantic desktop as a semantic web application. In SemDesk in ISWC. 2005.
5. D. Vrandeˇci´c. Deep integration of scripting language and semantic web technologies. In Scripting for the Semantic Web. 2005.Bebenita, M., Brandner, F., Fahndrich, M., Logozzo, F., Schulte, W., Tillmann, N., Venter, H., 2010. SPUR: a trace-based JIT compiler for CIL, in: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10. Association for Computing Machinery, Reno/Tahoe, Nevada, USA, pp. 708–725. https://doi.org/10.1145/1869459.1869517
6. Deliveroo Adopts Rust to Improve Performance in Core Service [WWW Document], 2020. InfoQ. URL https://www.infoq.com/news/2019/03/from-ruby-to-rust-at-deliveroo/ (accessed 2.16.20).
7. E. Oren, et al. Annotation and navigation in semantic wikis. In SemWiki in ESWC. 2006.
8. Garbage Collection | Ruby Hacking Guide [WWW Document], 2020. URL https://rubyhacking guide.github.io/gc.html (accessed 2.16.20).
9. Grimmer, M., Seaton, C., Würthinger, T., Mössenböck, H., 2015. Dynamically composing languages in a modular way: supporting C extensions for dynamic languages, in: Proceedings of the 14th International Conference on Modularity, MODULARITY 2015. Association for Computing Machinery, Fort Collins, CO, USA, pp. 1–13. https://doi.org/10.1145/2724525.2728790
10. Helix: Native Ruby Extensions Without Fear [WWW Document], 2020. URL https://usehelix.com/ (accessed 2.20.20).
11. Hunt, J., 2019. Monkey Patching and Attribute Lookup, in: Hunt, J. (Ed.), A Beginners Guide to Python 3 Programming, Undergraduate Topics in Computer Science. Springer International Publishing, Cham, pp. 325–336. https://doi.org/10.1007/978-3-030-20290-3_28
12. Lin, Y., Wang, K., Blackburn, S.M., Hosking, A.L., Norrish, M., 2015. Stop and go: understanding yieldpoint behavior. ACM SIGPLAN Not. 50, 70–80. https://doi.org/10.1145/2887746.2754187

13. Ludvigh, R., Rebok, T., Tunka, V., Nguyen, F., 2015. Ruby benchmark tool using docker, in: 2015 Federated Conference on Computer Science and Information Systems (FedCSIS). Presented at the 2015 Federated Conference on Computer Science and Information Systems (FedCSIS), pp. 947–952. https://doi.org/10.15439/2015F99
14. M. Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002.
15. Prokopec, A., Leopoldseder, D., Duboscq, G., Würthinger, T., 2017. Making collection operations optimal with aggressive JIT compilation, in: Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA 2017. Association for Computing Machinery, Vancouver, BC, Canada, pp. 29–40. https://doi.org/10.1145/3136000.3136002
16. T. Berners-Lee. Weaving the Web – The Past, Present and Future of the World Wide Web by its Inventor. Texere, 2000.

CHAPTER

3

IMPLEMENTING CONDITIONAL LOGIC

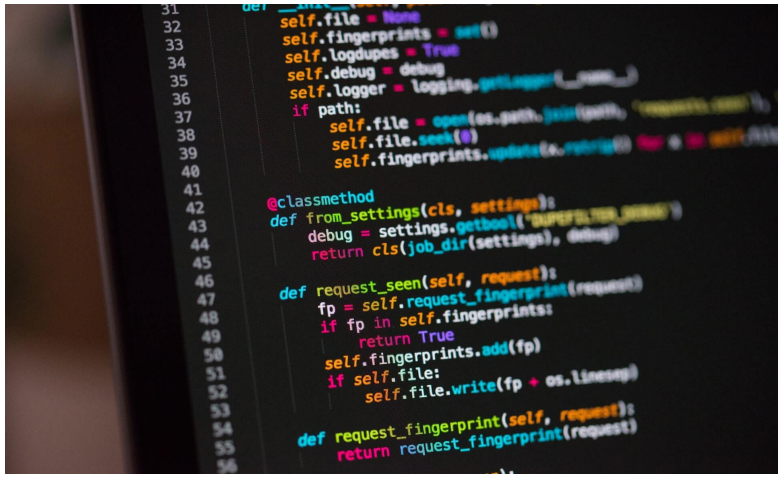
"It is not the responsibility of the language to force good looking code, but the language should make good looking code possible."

–Yukihiro Matsumoto

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Discuss the conditional statement
2. Define comparison operators and assignment operators
3. Explain the logical operators and ternary operator



INTRODUCTION

Conditional statements are also known by the name of conditional processing or conditional expressions. They are used to perform a certain set of instructions if a specified

condition is met. The conditions are generally of Boolean type and return either true or false as the result. Conditionals are formed using if or case with some comparison operators. A conditional assists data about where to move next. In Ruby, 0 is considered as true whereas in other programming languages it is considered false.

A conditional is a fork (or many forks) in the road. Your data approaches a conditional and the conditional then tells the data where to go based on some defined parameters. Conditionals are formed using a combination of if statements and comparison and logical operators (<, >, <=, >=, ==, !=, &&, ||). They are basic logical structures that are defined with the reserved words if, else, elsif, and end. Note that elsif is missing an "e". Enough talking, time to code.

Keyword

Conditional statement is a mechanism that allows for conditional execution of instructions based upon the outcome of a conditional statement, which can either be true or false.

3.1 CONDITIONAL STATEMENT

Many of Ruby's control structures, such as if and while, are standard fare and quite familiar to programmers, while others, like unless and until, are not. Think of control structures, which contain **conditional statements**, as lie detector tests. In every instance, when you use a control structure with a conditional, you are asking if something is true or false. When you get the desired answer—true or false depending on how you've designed your co0de—the code block associated with .the control is executed

3.1.1 The if Statement

Let's start out really simple and build from there.

```
if 1 == 1 then
 print "True!"
end
```

If it's true that 1 equals (==) 1, which it does, then the if statement returns true, and the code block, consisting only of a print statement, will execute. (This if statement, by the way, could be typed out on one line.)

Now you'll create a variable and compare it with a number. If the variable and number are equal, the code is executed.

```
x = 256
if x == 256
puts “x equals 256”
end
# => x equals 256
```

Notice that we dropped then from the if statement. You don’t have to use it in this instance. In addition, you don’t have to use end if you write this code all on one line, like so:

```
x = 256
if x == 256 then puts “x equals 256” end
```

In fact, you can change the order of things, placing if after puts, and you can drop then and end.

```
x = 256
puts “x equals 256” if x == 256
```

When you change the order like this, the if is referred to as a statement modifier. You can do this with other control structures.

Another way you can lay out an if statement is by replacing the then with a colon (:), like this:

```
x = 256
if x == 256: puts “x equals 256” end
```

Play with that code a little bit. Change the value of x so that it won’t return true when fed to if. Change the text that the statement outputs. Put something else in the block. Do this until you feel the code in your soul. Now I’ll show you some other operators for testing the truth or falsehood of a statement or set of statements. For example, the && operator means “and.”

```
ruby = “nifty”
programming = “fun”
if ruby == “nifty” && programming == “fun”
 puts “Keep programming!”
end
# => Keep programming!
```

In other words, if both these statements are true, execute the code in the block. You can have more than two statements separated by &&:

```
if a == 10 && b == 27 && c == 43 && d == -14
```

```
 print sum = a + b + c + d
end
```

If all these statements are true, sum will be printed.

You can also use the keyword and instead of &&.

```
if ruby == "nifty" and programming == "fun" and weather == "nice"
 puts "Stop programming and go outside for a break!"
end
```

Another choice is the || operator; a synonym for this operator is or. When you use || or or, if any of the statements are true, the code executes:

```
if ruby == "nifty" or programming == "fun"
 puts "Keep programming!"
end
```

If either of the two statements is true, the string keep programming! will print. Are more than two statements OK? Of course:

```
if a == 10 || b == 27 || c = 43 || d = -14
 print sum = a + b + c + d
end
```

|| and or, and && and and, are considered logical operators. Lots of other operators are possible, too, such as:

```
delete_record if record != 0x8ff # not equal to
if amt > 1.00 then desc = "dollars" end # greater than
desc = "cents" if amt < 1.00 # less than
if height >= 6 then print "L or XL" end # greater than or equal to
print "shrimpy" if weight <= 100 # less than or equal to
```

Two other operators reverse the meaning of a test. They are ! and not.

```
if !queue then print "The queue is empty." end
```

What this is saying is that if queue is not equal to true, the statement evaluates as true and the print statement prints The queue is empty!.An alternative to ! is the not keyword.

```
if not queue then print "The the queue is empty." end
```

Using else and elsif

Sometimes you set flags in programming in order to tell a program to carry out a task. A flag usually just carries a value of true or false. For example, let's say your program had **queue** and print flags. If the flag is true, then the code in the block is executed; if false, the block is ignored.

Keyword

A **Queue** is a linear structure which follows a particular order in which the operations are performed.

```
if queue
 pr = true
else
 pr = false
end

start_printer if pr # starts if pr is is true
```

The else keyword gives if an escape hatch. In other words, if the if statement does not evaluate true, the code after else will be executed, and if if evaluates false, the code after else is ignored.

The following if statement contains several elsif statements; they are testing to see which language is currently in use via symbols—English (:en), Spanish (:es), French (:fr), and German (:de)—to decide how to render dog:

Remember

The elsif keyword provides you with one or more intermediate options after the initial if, where you can test various statements.

```
lang = :es
if lang == :en
 print "dog"
elsif lang == :es
 print "perro"
elsif lang == :fr
 print "chien"
elsif lang == :de
 print "Hund"
else
 puts "No language set; default = 'dog'."
end
# "perro" is assigned to dog
```

You can also write this statement a little tighter by using colons after the symbols:

```
if lang == :en: print “dog”
 elsif lang == :es: print “perro”
 elsif lang == :fr: print “chien”
 elsif lang == :de: print “Hund”
 else puts “No language set; default = ‘dog’.”
end
```

3.1.2 The case Statement

Ruby’s case statement provides a way to express conditional logic in a succinct way. It is similar to using elsifs with colons, but you use case in place of if, and when in place of elsif. Here is an example similar to what you saw earlier using lang with the possible symbols :en, :es, :fr, and :de:

```
lang = :fr

dog = case lang
when :en: “dog”
when :es: “perro”
when :fr: “chien”
when :de: “Hund”
else “dog”
end
# “chien” is assigned to dog
```

Keyword

A **switch statement** is a type of selection control mechanism used to allow the value of a variable or expression to change the control flow of program execution via a multiway branch.

case/when is more convenient and terse than if/elsif/else because the logic of == is assumed—you don’t have to keep retyping == or the variable name: Ruby’s case is similar to the switch statement, a familiar C construct, but case is more powerful. One of the annoying things to me about **switch statements** in C, C++, and Java, is that you can’t switch on strings in a straightforward way.

If the lang variable held a string instead of symbols, your code would look like this:

4lang = “de”

```
dog = case lang
when “en”: “dog”
when “es”: “perro”
when “fr”: “chien”
when “de”: “Hund”
 else “dog”
end
# “Hund” is assigned to dog
```

The next example uses several ranges to test values. A range is a range of numbers.

```
scale = 8
case scale
 when 0: puts “lowest”
 when 1..3: puts “medium-low”
 when 4..5: puts “medium”
 when 6..7: puts “medium-high”
 when 8..9: puts “high”
 when 10: puts “highest”
 else puts “off scale”
end
# => high
```

The range 1..3 means a range of numbers from 1 to 3, inclusive. Because scale equals 8, scale matches the range 8..9 and case returns the string high. However, when you use three dots as in the range 1...5, the ending value 5 is excluded. The sets of dots, .. and ..., are called range operators; two dots includes all the numbers in the range, and three dots excludes the last value in the range. Underneath the hood, case uses the === operator from Range to test whether a value is a member of or included in a range.

 # case_statement.rb

Did You Know?

In 1997, the first article about Ruby was published on the Web. In the same year, Matsumoto was hired by netlab.jp to work on Ruby as a full-time developer.


```
a = 5

case a
when 5
puts "a is 5"
when 6
puts "a is 6"
else
puts "a is neither 5, nor 6"
end
```

3.1.3 The While Loop

A while loop executes the code it contains as long as its conditional statement remains true. The following piece of code initializes a counter i to 0 and sets up an array containing four elements called breeds (horse breeds).It also creates a temporary array named temp. The following few paragraphs are fairly fundamental, and are provided for beginning programmers. If you already have plenty of programming under your belt, skip ahead to the code itself.

The while loop will execute as long as its conditional (i < breeds.size) is true. The i variable starts out its little life equaling 0, and the size or length of the breeds array is 4. As you come to the end of the loop, i is incremented by 1, and then control returns to the top of the loop. In the first loop, i equals 0, and is fed as 0 as an argument to breeds[i], which retrieves the first element (numbered 0).This is the string value quarter. That element is appended via << to the temp array. The capitalize method from String changes quarter to Quarter. At this point, 1 is added to i by the += operator, so i equals 1. And we take it again from the top.

This continues until i equals 4, whereupon the conditional test for while fails. The Ruby interpreter moves to the next valid statement, that is, temp.sort!, which sorts the new array alphabetically. It does not make a copy but changes the array

in place. You know this by the tell-tale ! at the end of the method name (sort!).Then the contents of temp replace breeds, and we have cleaned up the array.

```
i = 0
breeds = [ “quarter”, “arabian”, “appalosa”, “paint” ]
puts breeds.size # => 4
temp = []

while i < breeds.size do
 temp << breeds[i].capitalize
 i +=1
end

temp.sort! # => [“Appalosa”, “Arabian”, “Paint”, “Quarter”]
breeds.replace( temp )
p breeds # => [“Appalosa”, “Arabian”, “Paint”, “Quarter”]
```

By the way, the do is optional here, so this form of the loop is legitimate, too:

```
while i < breeds.size
 temp << breeds[i].capitalize
 i +=1
end
```

Another form you can use is with begin/end:

```
 temp = 98.3

begin
 print “Your temperature is “ + temp.to_s + “ Fahrenheit. “
 puts “I think you’re okay.”
 temp += 0.1
end while temp < 98.6
puts “Your temperature is “ + temp.to_s + “ Fahrenheit. Are you okay?”
```

When you use while like this, with while at the end, the statements in the loop are evaluated once before the conditional is checked. This is like the do/while loop from C.

Also, like if, you can use while as a statement modifier, at the end of a statement:

```
cash = 100_000.00
sum = 0
cash += 1.00, sum while cash < 1_000_000.00 # underscores ignored
```

So cash just keeps adding up until it equals $1,000,000.00.

You can break out of a while loop with the keyword break. For example, let's say you were just looping along as before, but you wanted to stop processing once you got to a certain element in the array. You could use break to bust out, like this:

```
while i < breeds.size
 temp << breeds[i].capitalize
break if temp[i] == "Arabian"
 i +=1
end
p => temp # => ["Quarter", "Arabian"]
```

When the if modifier following break found Arabian in the temp array, it broke out of the loop right then. The next statement (which calls the p method) shows that we didn't get very far appending elements to the temp array.

Unless and Until

The unless and until statements are similar to if and while, except they are executed while their conditionals are false, whereas if and while statements are executed while their conditionals are true. Of course, if and while are used more frequently than unless and until, but the nice thing about having them is that Ruby offers you more expressiveness.

An unless statement is really like a negated if statement. We'll show you an if statement first:

```
if lang == "de"
 dog = "Hund"
else
 dog = "dog"
end
```

Now I'll translate it into unless:

```
unless lang == "de"
```

```
 dog = “dog”
else
 dog = “Hund”
end
```

This example is saying, in effect, that unless the value of lang is de, then dog will be assigned the value of dog; otherwise, assign dog the value Hund. See how the statements are reversed? In the if statement, the assignment of Hund to dog comes first; in the unless example, the assignment of dog to dog comes first. Like if, you can also use unless as a statement modifier:

```
puts age += 1 unless age > 29
```

As unless is a negated form of if, until is really a negated form of while. Compare the following statements. The first is a while loop:

```
weight = 150
while weight < 200 do
 puts “Weight: “ + weight.to_s
 weight += 5
end
```

Here is the same logic expressed with until:

```
weight = 150
until weight == 200 do
 puts “Weight: “ + weight.to_s
 weight += 5
end
```

And like while, you have another form you can use with until—that is, with begin/ end:

```
weight = 150
begin
 puts “Weight: “ + weight.to_s
 weight += 5
end until weight == 200
```

In this form, the statements in the loop are evaluated once before the conditional is checked. And finally, like while, you can also use until as a statement modifier:

```
puts age += 1 until age > 28
```

3.2 COMPARISON OPERATORS

Comparison operators take simple values (numbers or strings) as arguments and used to check for equality between two values. Ruby provides following comparison operators:

Operator	Name	Example	Result
==	Equal	x==y	True if x is exactly equal to y.
!=	Not equal	x!=y	True if x is exactly not equal to y.
>	Greater than	x>y	Teue if x is greater than y.
<	Less than	x<y	True if x is less than y.
>=	Greater than or equal to	x>=y	True if x is greater than or equal to y.
<=	Less than or equal to	x<=y	True if x is less than or equal to y.
<=>	Combined comparison operator.	x<=>y	x <=> y : = if x < y then return -1 if x =y then return 0 if x > y then return 1 if x and y are not comparable then return nil
===	Test equality	x===y	(10...20) === 9 return false.
.eql?	True if two values are equal and of the same type	x.eql? y	1 == 1.0 #=> true 1.eql? 1.0 #=> false
equal?	True if two things are same object.	obj1. equal? obj2	val = 10 => 10 val.equal?(10) => true

Example: Equality test

```
puts (“Test two numbers for equality with ==, !=, or <=>”)
 puts 14 == 16
 puts 14 != 16
 puts 14 <=> 14
 puts 14 <=> 12
 puts 14 <=> 16
```

Output:

```
Test two numbers for equality with ==, !=, or <=>
false
true
0
1
-1
```

Example: eql? and eqlity? operators

```
irb(main):023:0> 1 == 1.0
=> true
irb(main):024:0> 1.eql?1.0
=> false
irb(main):025:0> obj1 = “123”
=> “123”
irb(main):026:0> obj2 = obj1.dup
=> “123”
irb(main):027:0> obj1 == obj2
=> true
irb(main):028:0> obj1.equal?obj2
=> false
irb(main):029:0> obj1.equal?obj1
=> true
irb(main):030:0>
```

Example: Equal, less than, or greater than each other

```
puts (“Test if two numbers are equal, less than, or greater than each other”)
puts 14 < 16
puts 14 < 14
puts 14 <= 14
puts 14.0 > 12.5
puts 14.0 >= 14
```

Output:
Test if two numbers are equal, less than, or greater than each other
true
false
true
true
true

Example: Spaceship operator returns -1, 0, or 1

```
puts (“the <=> (spaceship operator) returns -1, 0, or 1,”)
puts 2 <=> 3
puts 2 <=> 2
puts 3 <=> 2
```

Output:
the <=> (spaceship operator) returns -1, 0, or 1,
-1
0
1

Example: Test the value in a range

```
puts (“test if a value is in a range”)
puts (12...16) === 8
puts (12...16) === 14
puts (12...16) === 16
puts (12...14) === 12
puts (12...16) === 14
```

Output:
test if a value is in a range
false
true

false

true

true

3.3 ASSIGNMENT OPERATORS

In Ruby assignment operator is done using the equal operator "=". This is applicable both for variables and objects, as strings, floats, and integers are actually objects in Ruby, you're always assigning objects.

Operator	Name	Description	Example
=	Equal operator "="	Simple assignment operator, Assigns values from right side operands to left side operand	z = x + y will assign value of a + b into c
+=	Add AND	Adds right operand to the left operand and assign the result to left operand	x += y is equivalent to x = x + y
-=	Subtract AND	Subtracts right operand from the left operand and assign the result to left operand	x -= y is equivalent to x = x - y
*=	Multiply AND	Multiplies right operand with the left operand and assign the result to left operand	x *= y is equivalent to x = x * y
/=	Divide AND	Divides left operand with the right operand and assign the result to left operand	x /= y is equivalent to x = x / y
%=	Modulus AND	Takes modulus using two operands and assign the result to left operand	x %= y is equivalent to x = x % y
**=	Exponent AND	Performs exponential calculation on operators and assign value to the left operand	x **= y is equivalent to x = x ** y

Example: Ruby assignment operator

```
puts ("assignment operator in Ruby")
x = 47
puts ("abbreviated assignment add")
```

```
puts x += 20
puts (“abbreviated assignment subtract”)
puts x -= 20
puts (“abbreviated assignment multiply”)
puts x *= 4
puts (“abbreviated assignment divide”)
puts x /= 4
puts (“abbreviated assignment modulus”)
puts x %= 6
puts (“abbreviated assignment exponent”)
puts x **= 4
```

Output:

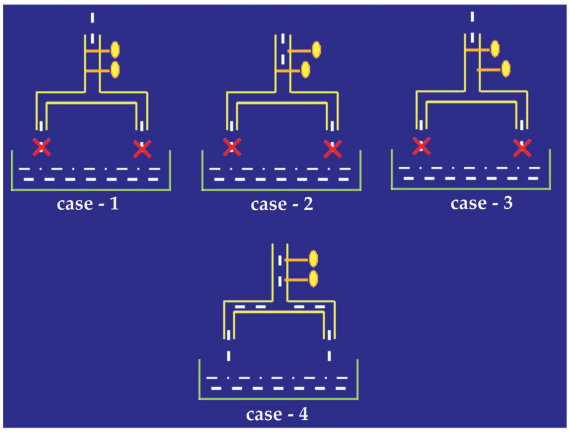
```
assignment operator in Ruby
abbreviated assignment add
67
abbreviated assignment subtract
47
abbreviated assignment multiply
188
abbreviated assignment divide
47
abbreviated assignment modulus
5
abbreviated assignment exponent
625
```

3.4 LOGICAL OPERATORS

The standard logical operators and, or and not are supported by Ruby. Logical operators first convert their operands to boolean values and then perform the respective comparison.

3.4.1 Logical and

The binary "and" operator returns the **logical conjunction** of its two operands. The condition becomes true if both the operands are true. It is the same as "&&" but with a lower precedence.



Keyword

Logical conjunction is an operation on two logical values, typically the values of two propositions, that produces a value of true if and only if both of its operands are true.

This above pictorial helps you to understand the concept of LOGICAL AND operation with an analogy of taps and water.

In case-1 of the picture, both of the taps are closed, so the water is not flowing down. Which explains that if both of conditions are FALSE or 0, the return is FALSE or 0.

In case-2 of the picture, one of the taps are closed, even then, the water is not flowing down. Which explains that even if any of conditions are FALSE or 0, the return is FALSE or 0.

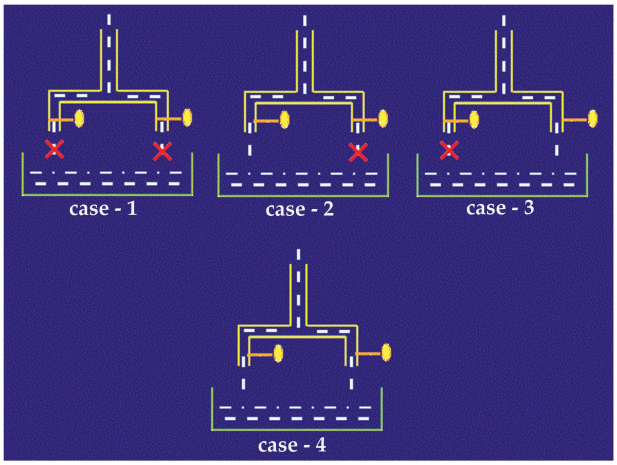
case-3 of the picture resembles CASE -2.

In case-4 of the picture, both of the taps are open, so the water is flowing down. Which explains that if both of conditions are TRUE or 1, the return is TRUE or 1.

So we can conclude that if and only if, both of the conditions are TRUE or 1, LOGICAL AND operations returns TRUE or 1.

3.4.2 Logical or

The binary "or" operator returns the logical disjunction of its two operands. The condition becomes true if both the operands are true. It is the same as "||" but with a lower precedence.



The above pictorial helps you to understand the concept of **LOGICAL OR** operation with an analogy of taps and water.

In case-1 of the picture, both of the taps are closed, so the water is not flowing down. Which explains that if both of conditions are FALSE or 0, the return is FALSE or 0.

In case-2 of the picture, one of the taps are closed, and we can see that the water is flowing down. Which explains that if any of conditions are TRUE or 1, the return is TRUE or 1.

case-3 of the picture, resembles CASE -2.

In case-4 of the picture, both of the taps are open, so the water is flowing down. Which explains that if both of conditions are TRUE or 1, the return is TRUE or 1.

So we can conclude that in LOGICAL OR operation, if any of the conditions are true, the output is TRUE or 1.

3.4.3 Logical not

The logical not or ! operator is used to reverse the logical state of its operand. If a condition is false, the logical not operator makes it true.

Example: Ruby logical operator

```
puts (“logical operators in Ruby”)
 ruby = “x”
 programming = “y”
      if ruby == “foo” && programming == “bar”
```

```
  puts "&&"
end

if ruby == "foo" and programming == "bar"
  puts "&& and"
end

p, q, r, s = 1, 2 ,3 , 4
if p == 1 && q == 2 && r == 3 && s == 4
  puts sum = p + q + r + s
end

programming = "ruby"

if ruby == "foo" || programming == "bar"
  puts "||"
end

if ruby == "foo" or programming == "bar"
  puts "|| or"
end

ruby = "awesome"
if ruby == "foo" or programming == "bar"
  puts "|| or"
else
   puts "sorry!"
end

if not (ruby == "foo" || programming == "bar")
   puts "nothing!"
end
```

```
    if !(ruby == “foo” or programming == “bar”)
      puts “nope!”
    end
```

Output:

```
logical operators in Ruby
10
sorry!
nothing!
nope!
```

Example-1: Ruby operators (&&, and) precedence

```
irb(main):016:0> foo = :foo
=> :foo
irb(main):017:0> bar = nil
=> nil
irb(main):018:0> x = foo and bar
=> nil
irb(main):019:0> x
=> :foo
irb(main):020:0> x = foo && bar
=> nil
irb(main):021:0> x
=> nil
irb(main):022:0> x = (foo and bar)
=> nil
irb(main):023:0> x
=> nil
irb(main):024:0> (x = foo) && bar
=> nil
irb(main):025:0> x
=> :foo
```

Example-1: Ruby operators (||, or) precedence

```
irb(main):001:0> foo = :foo
```

```
=> :foo
irb(main):002:0> bar = nil
=> nil
irb(main):003:0> x = foo or bar
=> :foo
irb(main):004:0> x
=> :foo
irb(main):005:0> x = foo || bar
=> :foo
irb(main):006:0> x
=> :foo
irb(main):007:0> x = (foo or bar)
=> :foo
irb(main):008:0> x
=> :foo
irb(main):009:0> (x = foo) || bar
=> :foo
irb(main):010:0> x
=> :foo
```

3.5 TERNARY OPERATOR

Ternary operator logic uses "(condition) ? (true return value): (false return value)" statements to shorten your if/else structures. It first evaluates an expression for a true or false value and then execute one of the two given statements depending upon the result of the evaluation. Here is the syntax:

```
test-expression ? if-true-expression : if-false-expression
```

Advantages of Ternary Logic:

Makes coding simple if/else logic quicker

Makes code shorter

Makes maintaining code quicker, easier

```
# Example-1
var = 5;
var_is_greater_than_three = (var > 3 ? true : false);
```

```
puts var_is_greater_than_three
# Example-2
score= 50
result = score > 40 ? 'Pass' : 'Fail'
puts result

# Example-3
score = 10;
age = 22;
puts "Taking into account your age and score, you are : ",(age > 10 ? (score < 80 ? 'behind' : 'above average') : (score < 50 ? 'behind' : 'above average'));

# Example-4
score = 81
puts "Based on your score, you are a ", (score > 80 ? "genius" :  "Not genius")
```

Output:

```
true
Pass
Taking into account your age and score, you are :
behind
Based on your score, you are a
genius
```

Remember

Use Comparison Operators as the building blocks to construct your conditional statements. There are some simple ones that you should already be familiar with: ==, <, >, >=, and <= != is "not equal".

SUMMARY

- Conditional statements are also known by the name of conditional processing or conditional expressions. They are used to perform a certain set of instructions if a specified condition is met.
- In Ruby, 0 is considered as true whereas in other programming languages it is considered false.
- Sometimes you set flags in programming in order to tell a program to carry out a task. A flag usually just carries a value of true or false
- The unless and until statements are similar to if and while, except they are executed while their conditionals are false, whereas if and while statements are executed while their conditionals are true.
- Comparison operators take simple values (numbers or strings) as arguments and used to check for equality between two values.
- The standard logical operators and, or and not are supported by Ruby.
- Logical operators first convert their operands to Boolean values and then perform the respective comparison.

KNOWLEDGE CHECK

1. **What is the use of else statement?**
   a. When the if condition is false then the next else condition will get executed
   b. When the if condition is false then the elsif condition will get executed
   c. When the if condition is false and if else condition is true then only it will get executed
   d. None of the mentioned
2. **Is the following syntax correct?**

```
if conditional
   code...
   elsif conditional
   code..
   else
   code
   end
```

   a. True
   b. False
3. **What is the output of the given code?**

```
if 1>2
   puts “false”
   else
   puts “True”
```

   a. False
   b. True
   c. Syntax error
   d. None of the mentioned
4. **The expression specified by the when clause is evaluated as the left operand. If no when clauses match, case executes the code of the else clause.**
   a. True
   b. False
5. **What error does the if condition gives if not terminated with end statement?**
   a. Syntax error
   b. Unexpected end

c. Expecting keyword end

d. All of the mentioned

6. The following syntax is correct for if conditional statement.

```
if condition
  code
end
```

a. True

b. False

7. What is the output of the following?

```
if 1<2
  print "one is less than two"
  end
```

a. One is less than two

b. Syntax error

c. 1<2

d. None of the mentioned

REVIEW QUESTIONS

1. Why is case/when somewhat more convenient than if/elsif/else?
2. What is the ternary operator?
3. What is a statement modifier?
4. Why is upto or downto more convenient than a regular for loop?
5. An unless statement is the negated form of what other control structure?
6. What are the synonyms for && and ||?

Check Your Result

1. (c) 2. (a) 3. (c) 4. (a) 5. (d)
6. (a) 7. (a)

REFERENCES

1. A. Inc., "Amazon elastic compute cloud," Online, 2006, http://aws.amazon.com/ec2/, last access Jan 2015.
2. Atlassian, "Jira issue tracking product," Online, 2002, https://www.atlassian.com/software/jira, last access Jan 2015.
3. C. Olszowka, "Statement coverage for Ruby," Online, 2010, https://github.com/colszowka/simplecov, last access Jan 2015.
4. D. Chelimsky, "Behavior driven development for Rbuy," Online, 2009, https://github.com/rspec/rspec, last access Jan 2015.
5. G. Kaminski, P. Ammann, and J. Offutt, "Improving logic-based testing," Journal of Systems and Software, Elsevier, vol. 86, no. 8, pp. 2002–2012, August 2012.
6. J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in Proceedings of the 27th International Conference on Software Engineering, St. Louis, Missouri, May 2005.
7. L. Deng, J. Offutt, and N. Li, "Empirical evaluation of the statement deletion mutation operator," in 6th IEEE International Conference on Software Testing, Verification and Validation (ICST 2013), Luxembourg, March 2013.
8. M. E. Delamaro, L. Deng, V. Durelli, N. Li, and J. Offutt, "Experimental evaluation of sdl and one-op mutation for C," in Proceedings of the 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, ser. ICST '14, Cleveland, Ohio, USA, 2014.
9. M. Schirp, "Mutation testing tool for Ruby," Online, 2012, https://github.com/mbj/mutant, last access Jan 2015.
10. N. Li and J. Offutt, "An empirical analysis of test oracle strategies for model-based testing," in Proceedings of the 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, Cleveland, Ohio, USA, 2014.
11. N. Li, U. Praphamontripong, and J. Offutt, "An experimental comparison of four unit test criteria: Mutation, edge-pair, alluses and prime path coverage," in Fourth Workshop on Mutation Analysis, Denver, CO, April 2009.
12. P. R. Mateo and M. P. Usaola, "Parallel mutation testing," Software Testing, Verification, and Reliability, vol. 23, no. 4, pp. 315–350, June 2013.
13. R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in Proceedings of the 36th International Conference on Software Engineering, ser. ICSE 2014, Hyderabad, India, 2014, pp. 72–82.
14. W. Shelton, N. Li, P. Ammann, and J. Offutt, "Adding criteriabased tests to test-driven development," in Testing: Academic and Industrial Conference - Practice and Research Techniques, ser. TAIC PART 2012, Montreal, Quebec, April 2012.

CHAPTER

4

WORKING WITH LOOPS

"This is one of the reasons Lisp doesn't get anywhere. The trend to promote features so clever that you stop thinking about your problem and start thinking about the clever features. CL's loop is so powerful that people invented functional programming so that they'd never have to use it"

–G_Morgan in reddit

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Explain a simple loop and discuss how to control loop execution
2. Understand the while loops and until loops
3. Discuss the do/while loops and for loops
4. Define the conditionals within loops and iterators
5. Explain the recursion and ruby flip-flop

```ruby
step_goal = 7
spaces_needed = step_goal - 1

char_count = 0
loop do
  if char_count < spaces_needed
    putc " "
  elsif char_count < step_goal
    putc "#"
  end
end
```

INTRODUCTION

Looping in programming languages is a feature which clears the way for the execution of a set of instructions or functions repeatedly when some of the condition evaluates

to true or false. Ruby provides the different types of loop to handle the condition based situation in the program to make the programmers task simpler. The loops in Ruby are:

- while loop
- for loop
- do..while loop
- until loop

A loop is the repetitive execution of a piece of code for a given amount of repetitions or until a certain condition is met. We will cover while loops, do/while loops, and for loops.

Loops are used to execute set of statements repeatedly based on a condition. It is sometimes necessary to execute set of statements again and again. For example, checking whether number in an array are prime or not.

4.1 A SIMPLE LOOP

The simplest way to create a loop in Ruby is using the loop method. looptakes a block, which is denoted by { ... } or do ... end. A loop will execute any code within the block (again, that's just between the {} or do ... end) until you manually intervene with Ctrl + c or insert a break statement inside the block, which will force the loop to stop and the execution will continue after the loop.

Let's try an example of a loop by creating a file named loop_example.rb

loop_example.rb #

loop do

"puts "This will keep printing until you hit Ctrl + c

end

.Now we can run ruby loop_example.rb on the terminal and see what happens

You'll notice the same statement keeps printing on the terminal. You'll have to .interrupt with a Ctrl + c to stop it

This will keep printing until you hit Ctrl + c

This will keep printing until you hit Ctrl + c

This will keep printing until you hit Ctrl + c

This will keep printing until you hit Ctrl + c

:This will keep printing until you hit Ctrl + cInterrupt

'from (pry):2:in `puts

<(pry(main [2]

4.2 CONTROLLING LOOP EXECUTION

You'll hardly do something like this in a real program as it's not very useful and will result in an **infinite loop**. Eventually your system will crash.

Let's look at a more useful example with the break keyword by creating a file named useful_loop.rb:

```
# useful_loop.rb
i = 0
loop do
  i += 1
  puts i
  break          # this will cause execution to exit the loop
end
```

When you run useful_loop.rb in your terminal, the output should be:

```
$ ruby useful_loop.rb
1
```

The break keyword allows us to exit a loop at any point, so any code after a break will not be executed.

Next, let's look at adding conditions within a loop by printing all even numbers from 0 up to 10. Let's create a file named conditional_loop.rb

```
# conditional_loop.rb


i = 0
loop do
  i += 2
  puts i
  if i == 10
    break        # this will cause execution to exit the loop
  end
end
```

Here's the output when we run the file:

Keyword

Infinite loop is a sequence of instructions in a computer program which loops endlessly, either due to the loop having no terminating condition, having one that can never be met, or one that causes the loop to start over.

Remember

That break will not exit the program, but only exit the loop and execution will continue on from after the loop.

```
$ ruby conditional_loop.rb
2
4
6
8
10
```

You can see from the above that break was not executed during the first 4 iterations through the loop, but on the 5th iteration, the if statement evaluated to true and so the code within the if statement was executed, which is just break, and execution exited the loop.

We'll talk explicitly about using conditionals within a loop later. Similar to how we use break to exit a loop, we can use the keyword next to skip the rest of the current iteration and start executing the next iteration. We'll use the same example as before to demonstrate. This time we'll skip 4.

```
# next_loop.rb

i = 0
loop do
  i += 2
  if i == 4
    next        # skip rest of the code in this iteration
  end
  puts i
  if i == 10
    break
  end
end
```

And here's the output when we run the file.

```
$ ruby next_loop.rb
2
6
8
10
```

Notice that the above code did not print out 4, because that was skipped over. Execution continued to the next iteration of the loop.

break and next are important loop control concepts that can be used with loop or any other loop construct in Ruby, which we'll cover one by one below. When combined with conditionals, you can start to build simple programs with interesting behavior.

4.3 WHILE LOOPS

A while loop is given a parameter that evaluates to a boolean (remember, that's just true or false). Once that boolean expression becomes false, the while loop is not executed again, and the program continues after the while loop. Code within the while loop can contain any kind of logic that you would like to perform. Let's try an example of a while loop by creating a file named countdown.rb. We want this program to countdown from any number given by the user and print to the screen each number as it counts. Here we go!

```
# countdown.rb

x = gets.chomp.to_i

while x >= 0
  puts x
  x = x - 1
end

puts "Done!"
```

Keyword

Negative Number is a real number that is less than zero.

Now go to your terminal and run this program with ruby countdown.rb. You'll notice that the program initially waits for you to put in a number then executes the loop.

Initially the program evaluates the line x >= 0. This evaluates to true (unless you entered a **negative number**) and so the program enters the loop, executing puts x and the line after that, x = x - 1. Then the program returns to the top, now with the newly updated value of x and evaluates the x >= 0 again. This process repeats until the value of x is no longer greater than or equal to 0. It then exits the loop and continues with the rest of the program. You can see why it's called a loop. It loops over the logic within itself repeatedly.

We'd also like to take this opportunity to show you a small trick for refactoring this loop.

```
# countdown.rb

x = gets.chomp.to_i

while x >= 0
  puts x
  x -= 1 # <- refactored this line
end

puts "Done!"
```

We changed the line x = x - 1 to x -= 1. This is common to many programming languages and it's a nice succinct way to say the same thing with less typing. You can use it with any other operator as well (+, *, /, etc.).

You should also be aware that because we're using the x >= 0 expression as the test to see if we should execute the loop, the code within the loop *must* modify the variable x in some way. If it does not, then x >= 0 will always evaluate to true and cause an *infinite loop*. If you ever find your program unresponsive, it's possible that it is stuck in an infinite loop.

Example:

The following codes print the numbers 0 through 10. The condition a < 10 is checked before the loop is entered, then the body executes, then the condition is checked again. When the condition results in false the loop is terminated.

```
x =  1
y =  11
while x <  y  do
  print  x ,". Ruby while loop.\n"
  x +=1
  end
```

Copy

Output:

1. Ruby while loop.

2. Ruby while loop.
3. Ruby while loop.
4. Ruby while loop.
5. Ruby while loop.
6. Ruby while loop.
7. Ruby while loop.
8. Ruby while loop.
9. Ruby while loop.
10. Ruby while loop.

Within the while statement, the 'do' keyword is optional. The following loop is equivalent to the loop above:

```
x =  1
y =  11
while x <  y
  print  x ,". Ruby while loop.\n"
  x +=1
  end
```

Copy

Ruby while modifier:

Like if and unless, while can be used as modifiers.

```
x = 0
x += 1 while x < 10
p  x      # prints 10
```

Copy

You can use begin and end to create a while loop that runs the body once before the condition:

```
x = 0
begin
  x += 1
end while x <10
p  x  # prints 10
```

Did You Know?

Very early computers, such as Colossus, were programmed without the help of a stored program, by modifying their circuitry or setting banks of physical controls.

4.4 UNTIL LOOPS

We didn't mention the until loop in the introduction paragraph. We do, however, need to mention them briefly so that you know about them. The until loop is simply the opposite of the while loop. You can substitute it in order to phrase the problem in a different way. Let's look briefly at how it works.

```
# countdown.rb

x = gets.chomp.to_i

until x < 0
  puts x
  x -= 1
end

puts "Done!"
```

There are instances when using until will allow you to write code that is more readable and logical. Ruby has many features for making your code more expressive. The until loop is one of those features.

Example:

The following script prints the numbers 1 through 10. Like a while loop the condition x > 11 is checked when entering the loop and each time the loop body executes. If the condition is false the loop will continue to execute.

```
x =  1
y =  11
until x > y  do
  print  x ,". Ruby while loop.\n"
  x +=1
  end
```

Copy

Output:

1. Ruby while loop.
2. Ruby while loop.

3. Ruby while loop.
4. Ruby while loop.
5. Ruby while loop.
6. Ruby while loop.
7. Ruby while loop.
8. Ruby while loop.
9. Ruby while loop.
10. Ruby while loop.

4.5 DO/WHILE LOOPS

A **do/while loop** works in a similar way to a while loop; one important difference is that the code within the loop gets executed one time, prior to the conditional check to see if the code should be executed. In a "do/while" loop, the conditional check is placed at the end of the loop as opposed to the beginning. Let's write some code that asks if the user wants to perform an action again, but keep prompting if the user enters 'Y'. This is a classic use case for a "do/while", because we want to repeatedly perform an action based on some condition, but we want the action to be executed at least one time no matter what.

```
# perform_again.rb
loop do
  puts "Do you want to do that again?"
  answer = gets.chomp
  if answer != 'Y'
    break
  end
end
```

Notice that we're using a simple loop, except the condition to break out of the loop occurs at the end, therefore ensuring that the loop executes at least once. Try copying and pasting the above code into irb and playing around with it yourself. Compare this with a normal "while" loop.

Side note: there's also another construct in Ruby that supports "do/while" loops, like this:

```
begin
  puts "Do you want to do that again?"
  answer = gets.chomp
```

```
end while answer == 'Y'
```

While the above works, it's not recommended by Matz, the creator of Ruby.

4.6 FOR LOOPS

In Ruby, for loops are used to loop over a collection of elements. Unlike a while loop where if we're not careful we can cause an infinite loop, for loops have a definite end since it's looping over a finite number of elements. It begins with the for reserved word, followed by a variable, then the in reserved word, and then a collection of elements. We'll show this using an **array** and a range. A range is a special type in Ruby that captures a range of elements. For example 1..3 is a range that captures the integers 1, 2, and 3.

Keyword

Array is a systematic arrangement of similar objects, usually in rows and columns.

```
# countdown3.rb
x = gets.chomp.to_i
for i in 1..x do
  puts i
end
puts "Done!"
```

The odd thing about the for loop is that the loop returns the collection of elements after it executes, whereas the earlier while loop examples return nil. Let's look at another example using an array instead of a range.

```
# countdown4.rb

x = [1, 2, 3, 4, 5]

for i in x do
  puts i
end

puts "Done!"
```

You can see there are a lot of ways to loop through a collection of elements using Ruby. Let's talk about some more interesting ways you can use conditions to modify the

behavior of your loops. Most Rubyists forsake for loops and prefer using iterators instead.


```
#!/usr/bin/ruby

$i = 0

$num = 5

until $i > $num  do

puts("Inside the loop i = #$i" )

$i +=1;

end
```

4.7 CONDITIONALS WITHIN LOOPS

To make loops more effective and precise, we can add conditional flow control within them to alter their behavior. Let's use an if statement in a while loop to demonstrate.

conditional_while_loop.rb #

```
x = 0

while x <= 10
  if x.odd?
    puts x
  end
  x += 1
end
```

This loop uses the odd? method to decide if the current variable in the loop is odd. If it is, it prints to the screen. Next,x increments by one, and then the loop proceeds to the next iteration.

The reserved words next and break can be useful when looping as well.

If you place the next reserved word in a loop, it will jump from that line to the next loop iteration without executing the code beneath it. If you place the break reserved word in a loop, it will exit the loop immediately without executing any more code in the loop.

```
# conditional_while_loop_with_next.rb

x = 0

while x <= 10
  if x == 3
    x += 1
    next
  elsif x.odd?
    puts x
  end
  x += 1
end
```

We use the next reserved word here to avoid printing the number 3 in our loop. Let's try break as well.

```
# conditional_while_loop_with_break.rb

x = 0

while x <= 10
  if x == 7
    break
  elsif x.odd?
    puts x
  end
  x += 1
end
```

When you run this program you can see that the entire loop exits when the value of x reaches 7 in the loop. That is why the print out only goes to 5.

Loops are basic constructs in any **programming language**, .but most Rubyists, where possible, prefer iterators over loops

4.8 ITERATORS

Iterators are methods that naturally loop over a given set of data and allow you to operate on each element in the collection.

We said earlier that arrays are ordered lists. Let's say that you had an array of names and you wanted to print them to the screen. How could you do that? You could use the each method for arrays, like this:

```
# practice_each.rb
names = ['Bob', 'Joe', 'Steve', 'Janice', 'Susan', 'Helen']
names.each { |name| puts name }
```

Isn't that concise! We've got a lot of explaining to do with this one.

We have called each method using the dot operator (.) on our array. What this method does is loop through each element in our array, in order, starting from 'Bob'. Then it begins executing the code within the block. The block's starting and ending points are defined by the curly braces {}. Each time we iterate over the array, we need to assign the value of the element to a variable. In this example we have named the variable name and placed it in between two pipes |. After that, we write the logic that we want to use to operate on the variable, which represents the current array element. In this case it is simply printing to the screen using puts.

Run this program to see the output.

A **block** is just some lines of code ready to be executed. When working with blocks there are two styles you need to be aware of. By convention, we use the curly braces ({}) when everything can be contained in one line. We use the words do and end when we are performing multi-line operations. Let's add some functionality to our previous program to try out this do/end stuff.

```
# practice_each.rb
names = ['Bob', 'Joe', 'Steve', 'Janice', 'Susan', 'Helen']
x = 1
```

Keyword

Programming language is a formal language, which comprises a set of instructions used to produce various kinds of output. Programming languages are used to create programs that implement specific algorithms.

```
names.each do |name|
  puts "#{x}. #{name}"
  x += 1
end
```

We've added the counter x to add a number before each name, creating a numbered list output. The number x is incremented every time we go through the iteration.

Memorizing these small differences in syntax is one of the necessary tasks a Ruby programmer must go through. Ruby is a very expressive language. Part of what makes that possible is the ability to do things in more than one way.

There are many other iterator methods in Ruby, and over time, you'll get to use a lot of them. For now, know that most Rubyists prefer to use iterators, like each method, to loop over a collection of elements.

4.9 RECURSION

Recursion is another way to create a loop in Ruby. Recursion is the act of calling a method from within itself. That probably sounds confusing so let's look at some actual code to get a better idea.

A Simple Example

Let's say you wanted to know what the double of a number was, then the double of that number, etc. Let's say you wanted to double the number until the pre-doubled number is 10 or greater. You could create the following method:

```
def doubler(start)
  puts start * 2
end
```

And then you can use it like this:

```
irb(main):001:0> def doubler(start)
irb(main):002:1>   puts start * 2
irb(main):003:1> end
=> :doubler
irb(main):004:0> doubler(2)
4
=> nil
irb(main):005:0> doubler(4)
8
```

```
=> nil
irb(main):006:0> doubler(8)
16
=> nil
```

You can do this much more simply using recursion. Take a look at this version of the method:

```
def doubler(start)
  puts start
  if start < 10
    doubler(start * 2)
  end
end
```

This version of the method calls the doubler method again, passing it the doubled version of the value stored in the start variable. Once again, here is the declaration and use of the method using irb:

```
irb(main):001:0> def doubler(start)
irb(main):002:1>   puts start
irb(main):003:1>   if start < 10
irb(main):004:2>     doubler(start * 2)
irb(main):005:2>   end
irb(main):006:1> end
=> :doubler
irb(main):007:0> doubler(2)
2
4
8
16
=> nil
```

Another Example

We are using a method that uses recursion to calculate the nth number in the fibonacci sequence. You can learn more about the fibonacci sequence here. Basically, it is a sequence of numbers in which each number is the sum of the previous two numbers in the sequence.

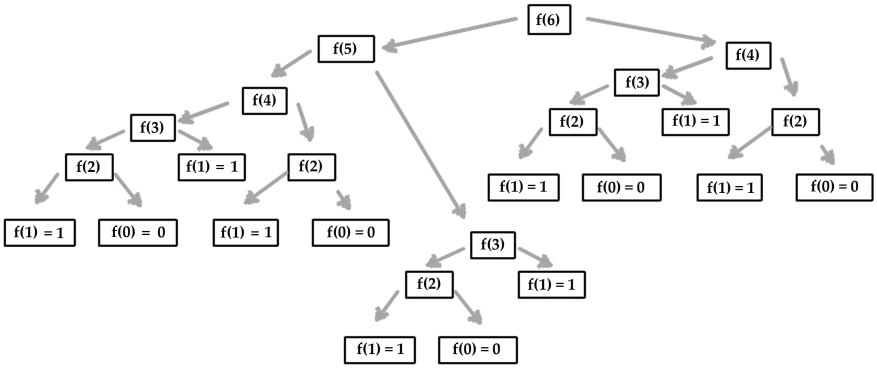
Note: This example may take a few reads to really grasp what's happening at every point in the program. That's normal. Just take your time, and you'll be fine. Also, be

excited! We are getting closer to reading more real-world examples!

Make the following file:

```
# fibonacci.rb

def fibonacci(number)
  if number < 2
    number
  else
    fibonacci(number - 1) + fibonacci(number - 2)
  end
end
puts fibonacci(6)
```

If you're panicking, don't be scared. Soon this will be simple to you. We just have to take it slow and understand everything that's going on, line-by-line. Recursion is a tricky subject for all programmers, so don't let this frustrate you any more than a healthy amount. When learning recursion, drawing diagrams can help. We can use a tree like structure to see what is happening. (We used fto abbreviate fibonacci to save space.)



Each time the code branches off again you are calling the fibonacci function from within itself two times. If you take all of those ones and zeros and add them together, you'll get the same answer you get when you run the code. You can see why computer programs are handy now. Think if you had to draw that diagram out every time you wanted to know the fibonacci respresentation of a number. Yikes!

The key concept with recursion is that there is some baseline condition that returns a value, which then "unwinds" the recursive calls. You can think of the successive

recursive calls building up, until some value is returned, and only then can the recursive calls be evaluated.

4.10 Ruby Flip-Flop

The flip-flop is used to process text from ruby one-line programs used with ruby -n or ruby -p. The form of the flip-flop is an expression that indicates when the flip-flop turns on, .. (or ...), then an expression that indicates when the flip-flop will turn off. While the flip-flop is on it will continue to evaluate to true, and false when off.

The flip-flop must be used inside a conditional such as if, while, unless, until etc.

Example

In the following example, the on condition is n==12. The flip-flop is initially off (false) for 10 and 11, but becomes on (true) for 12 and remains on through 18. After 18 it turns off and remains off for 19 and 20.

```
[] = selected
    10.upto 20 do |value|
     selected << value if value==12..value==18
    end
    p selected
    Copy
    Output:
    [12, 13, 14, 15, 16, 17, 18]
```

Remember

There are naming conventions governing variable names in Ruby. These conventions are not enforced, but you should stick by them if you want people to like you, since Ruby is case sensitive.

CASE STUDY

NESTED LOOPS

Composing computer programs to solve scientific problems is like writing poetry. You must choose every word with care and link it with the other words in perfect syntax. There is no place for verbosity or carelessness. To become fluent in a computer language demands almost the antithesis of modern loose thinking. It requires many interactive sessions, the hands-on use of the device. You do not learn a foreign language from a book, rather you have to live in the country for year to let the language become an automatic part of you, and the same is true for computer languages. "

SUMMARY

- The simplest way to create a loop in Ruby is using the loop method. looptakes a block, which is denoted by { ... } or do ... end.
- A while loop is given a parameter that evaluates to a boolean (remember, that's just true or false).
- We didn't mention the until loop in the introduction paragraph. We do, however, need to mention them briefly so that you know about them.
- A do/while loop works in a similar way to a while loop; one important difference is that the code within the loop gets executed one time, prior to the conditional check to see if the code should be executed.
- In Ruby, for loops are used to loop over a collection of elements. Unlike a while loop where if we're not careful we can cause an infinite loop, for loops have a definite end since it's looping over a finite number of elements.
- To make loops more effective and precise, we can add conditional flow control within them to alter their behavior.
- Iterators are methods that naturally loop over a given set of data and allow you to operate on each element in the collection.
- Recursion is another way to create a loop in Ruby. Recursion is the act of calling a method from within itself.
- The flip-flop is used to process text from ruby one-line programs used with ruby -n or ruby -p. The form of the flip-flop is an expression that indicates when the flip-flop turns on, .. (or ...), then an expression that indicates when the flip-flop will turn off.

KNOWLEDGE CHECK

1. What is the output of the given code?

```
for num in 1...5
  puts num
end
```

a. 1 2 3 4 5
b. 1 2 3 4
c. 2 3 4 5
d. None of the mentioned

2. What does the 1…10 indicate?

a. Inclusive range
b. Exclusive range
c. Both inclusive and exclusive range
d. None of the mentioned

3. What is the output of the given code?

```
for num in 1..3
  puts num
  for i in 1..2
  puts num*i
  end
end
```

a. 1 2 3 4 5
b. 1 1 2 2 2 4 3 3 6
c. 2 3 4 5
d. None of the mentioned

4. What is the output of the given code?

```
m= 0
loop do
    m += 1
    print m
    break if m == 10
end
```

a. 12345678910

b. 1 2 3 4

c. 2 3 4 5

d. None of the mentioned

5. What is the output of the given code?

```
for num in 1..5
  puts num*num
end
```

a. 12345678910

b. 1 2 3 4

c. 1 4 9 16 25

d. None of the mentioned

6. Which of the following is not a type of loop in ruby?

a. For Loop

b. Foreach Loop

c. Until Loop

d. While Loop

7. What is true about while loop?

a. Executes code while conditional is true

b. In while loop increment is not required

c. Executes code while conditional is false

d. None of the above

REVIEW QUESTIONS

1. What is a for loop?
2. What does the each method in the following program return after it is finished executing?

```
x = [1, 2, 3, 4, 5]
x.each do |a|
  a + 1
end
```

3. Write a while loop that takes input from the user, performs an action, and only stops when the user types "STOP". Each loop can get info from the user.

4. Use the each_with_index method to iterate through an array of your creation that prints each index and value of the array.
5. Write a method that counts down to zero using recursion.

Check Your Result

1. (b) 2. (b) 3. (b) 4. (a) 5. (c)
6. (b) 7. (a)

REFERENCES

1. A. Gurtovoy and D. Abrahams, "The Boost C++ metaprogramming library," Tech. Rep., Mar. 2002, http://www.boost.org/libs/mpl/doc/paper/mpl paper.pdf.
2. Andreu Carminati, Renan Augusto Starke, and Rômulo Silva de Oliveira. 2017. Combining loop unrolling strategies and code predication to reduce the worstcase execution time of real-time software. Applied Computing and Informatics 13, 2 (2017), 184–193.
3. B. Benatallah and al : Representing, Analysing and Managing Web Service Protocols. Data Knowledge Ingineering. 58 (3): 327-357, 2006.
4. Barhamgi, M., Benslimane, D., Medjahed, B. : A Query Rewriting Approach for Web Service Composition. IEEE Transactions Services Computing. 3, 206–222 (2010).
5. D. M. Beazley, "Automated scientific software scripting with SWIG," Future Generation Computer Systems, vol. 19, no. 5, pp. 599–609, July 2003.
6. im Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2015. The Java Virtual Machine Specification, Java SE 8 Edition. http://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf
7. J. Wedekind, B. Amavasai, and K. Dutton, "Steerable filters generated with the hypercomplex dual-tree wavelet transform," in IEEE International Conference on Signal Processing and Communications, 2007, pp. 1291–1294
8. Laurence Tratt and Roel Wuyts, "Guest editors' introduction: Dynamically typed languages," IEEE Software, vol. 24, no. 5, pp. 28–30, 2007.
9. Marcus Denker and Stephane Ducasse, "Software evolution from the field. an experience report from the Squeak maintainers," Electronic Notes in Theoretical Computer Science, vol. 166, pp. 81–91, 2007.
10. S. Baker and I. Matthew, "Lucas-kanade 20 years on: a unifying framework," International Journal of Computer Vision, vol. 56, no. 3, pp. 221–55, Feb. 2004.
11. Zaljko Obrenovic and Dragan Gasevic, "Open source software: All you do is put it together," IEEE Software, vol. 24, no. 5, pp. 86–95, 2007.
12. Zhou, L., Chen, H., Wang, H., Zhang,Y. : Semantic Web-Based Data Service Discovery and Composition. SKG. 213–219 (2008).

CHAPTER

5

WORKING WITH REGULAR EXPRESSIONS

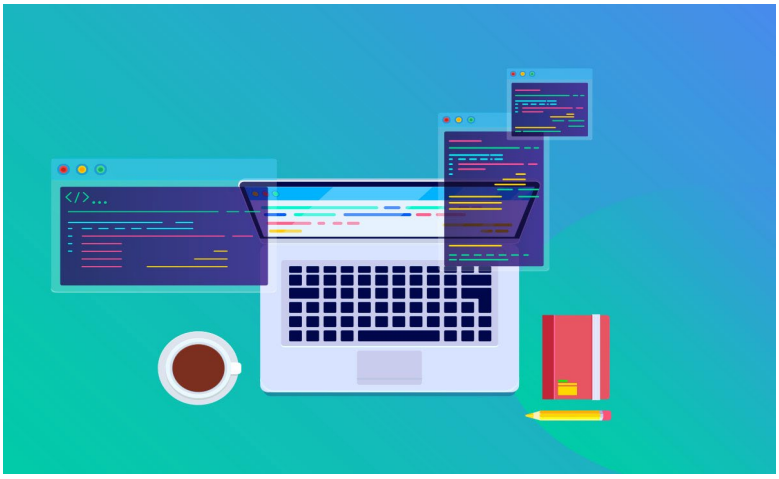
"Ruby on Rails is a breakthrough in lowering the barriers of entry to programming. Powerful web applications that formerly might have taken weeks or months to develop can be produced in a matter of days."

–Tim O'Reilly,

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Mastering the ruby regular expressions
2. Learn about the digging deeper techniques



INTRODUCTION

A regular expression is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings. Ruby regular expressions i.e. Ruby regex for short,

helps us to find particular patterns inside a string. Two uses of ruby regex are validation and parsing. Ruby regex can be used to validate an email address and an IP address too. Ruby regex expressions are declared between two forward slashes.

A Regexp holds a regular expression, used to match a pattern against strings. Regexps are created using the /.../ and %r{...} literals, and by the Regexp::new constructor. Regular expressions (regexps) are patterns which describe the contents of a string. They are used for testing whether a string contains a given pattern, or extracting the portions that match. They are created with the /pat/ and %r{pat} literals or the Regexp. new constructor.

5.1 MASTERING RUBY REGULAR EXPRESSIONS

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings using a specialized syntax held in a pattern.

A regular expression literal is a pattern between slashes or between arbitrary delimiters followed by %r as follows −

Syntax

```
/pattern/
/pattern/im     # option can be specified
%r!/usr/local! # general delimited regular expression
```

Example

```
#!/usr/bin/ruby

line1 = “Cats are smarter than dogs”;
line2 = “Dogs also like meat”;

if ( line1 =~ /Cats(.*)/ )
   puts “Line1 contains Cats”
end
if ( line2 =~ /Cats(.*)/ )
   puts “Line2 contains  Dogs”
    end
```

This will produce the following result −

```
Line1 contains Cats
```

5.1.1 Regular-Expression Modifiers

Regular expression literals may include an optional modifier to control various aspects of matching. The modifier is specified after the second slash character, it may be represented by one of these characters –

Sr.No.	Modifier & Description
1	**i** Ignores case when matching text.
2	**o** Performs #{} interpolations only once, the first time the regexp literal is evaluated.
3	**x** Ignores whitespace and allows comments in regular expressions.
4	**m** Matches multiple lines, recognizing newlines as normal characters.
5	**u,e,s,n** Interprets the regexp as Unicode (UTF-8), EUC, SJIS, or ASCII. If none of these modifiers is specified, the regular expression is assumed to use the source encoding.

Like string literals delimited with %Q, Ruby allows you to begin your regular expressions with %r followed by a delimiter of your choice. This is useful when the pattern you are describing contains a lot of forward slash characters that you don't want to escape –

```
# Following matches a single slash character, no escape required
%r|/|
# Flag characters are allowed with this syntax, too
r[</(.*)>]i%
```

Remember

The sub and gsub returns a new string, leaving the original unmodified where as sub! and gsub! modify the string on which they are called.

5.1.2 Search and Replace

Some of the most important String methods that use regular expressions are sub and gsub, and their in-place variants

sub! and gsub!. All of these methods perform a search-and-replace operation using a Regexp pattern. The sub & sub! replaces the first occurrence of the pattern and gsub& gsub! replaces all occurrences.

Following is the example –

```
#!/usr/bin/ruby

phone = "2004-959-559 #This is Phone Number"

# Delete Ruby-style comments
phone = phone.sub!(/#.*$/, "")
puts "Phone Num : #{phone}"

# Remove anything other than digits
phone = phone.gsub!(/\D/, "")
puts "Phone Num : #{phone}"
```

This will produce the following result –

```
Phone Num : 2004-959-559
Phone Num : 2004959559
```

Following is another example –

```
#!/usr/bin/ruby

text = "rails are rails, really good Ruby on Rails"

# Change "rails" to "Rails" throughout
text.gsub!("rails", "Rails")

# Capitalize the word "Rails" throughout
   text.gsub!(/\brails\b/, "Rails")

   puts "#{text}"
```

This will produce the following result –

```
Rails are Rails, really good Ruby on Rails
```

5.1.3 Regular-Expression Patterns

Except for control characters, **(+ ? . * ^ $ ( ) [ ] { } | \)**, all characters match themselves. You can escape a control character by preceding it with a backslash.

There are many ways of creating a regular expression pattern. By far the most common is to write it between forward slashes. Thus, the pattern /cat/ is a regular expression literal in the same way that "cat" is a string literal.

/cat/ is an example of a simple, but very common, pattern. It matches any string that contains the substring cat. In fact, inside a pattern, all characters except ., |, (,), [,], {, }, +, \, ^, $, *, and ? match themselves. So, at the risk of creating something that sounds like a logic puzzle, here are some patterns and examples of strings they match and don't match:

/cat/ Matches "dog and cat" and "catch" but not "Cat" or "c.a.t."

/123/ Matches "86512312" and "abc123" but not "1.23"

/t a b/ Matches "hit a ball" but not "table"

If you want to match one of the special characters literally in a pattern, precede it with a backslash, so /*/ is a pattern that matches a single asterisk, and /\/ /} is a pattern that matches a forward slash.

Pattern literals are like double-quoted strings. In particular, you can use #{...} expression substitutions in the pattern.

Matching Strings with Patterns

The Ruby operator =~ matches a string against a pattern. It returns the character offset into the string at which the match occurred:

```
/cat/ =~ "dog and cat" # => 8
/cat/ =~ "catch"       # => 0
/cat/ =~ "Cat"         # => nil
```

You can put the string first if you prefer:

```
"dog and cat" =~ /cat/ # => 8
"catch" =~ /cat/       # => 0
"Cat" =~ /cat/         # => nil
```

Keyword

Boolean is a subset of algebra used for creating true/false statements. Boolean expressions use the operators AND, OR, XOR, and NOT to compare values and return a true or false result.

Because pattern matching returns nil when it fails and because nil is equivalent to false in a **Boolean** context, you can use the result of a pattern match as a condition in statements such as if and while.

```
str = "cat and dog"

if str =~ /cat/
  puts "There's a cat here somewhere"
end
```

produces:

There's a cat here somewhere

The following code prints lines in testfile that have the string on in them:

```
File.foreach("testfile").with_index do |line, index|
  puts "#{index}: #{line}" if line =~ /on/
end
```

produces:

```
0: This is line one
3: And so on...
```

You can test to see whether a pattern does not match a string using !~:

```
File.foreach("testfile").with_index do |line, index|
  puts "#{index}: #{line}" if line !~ /on/
end
```

produces:

```
1: This is line two
2: This is line three
```

Changing Strings with Patterns

The sub method takes a pattern and some replacement text. If it finds a match for the pattern in the string, it replaces the matched substring with the replacement text.

str = "Dog and Cat"

new_str = str.sub(/Cat/, "Gerbil")

puts "Let's go to the #{new_str} for a pint."

produces:

Let's go to the Dog and Gerbil for a pint.

The sub method changes only the first match it finds. To replace all matches, use gsub. (The g stands for global.)

str = "Dog and Cat"

new_str1 = str.sub(/a/, "*")

new_str2 = str.gsub(/a/, "*")

puts "Using sub: #{new_str1}"

puts "Using gsub: #{new_str2}"

produces:

Using sub: Dog *nd Cat

Using gsub: Dog *nd C*t

Both sub and gsub return a new string. (If no substitutions are made, that new string will just be a copy of the original.)

If you want to modify the original string, use the sub! and gsub! forms:

```
str = "now is the time"
str.sub!(/i/, "*")
str.gsub!(/t/, "T")
puts str
```

produces:

now *s The Time

Unlike sub and gsub, sub! and gsub! return the string only if the pattern was matched. If no match for the pattern is found in the string, they return nil instead. This means it can make sense (depending on your need) to use the ! forms in conditions. So, at this point you know how to use patterns to look for text in a string and how to substitute different text for those matches. And, for many people, that's enough. So if you're itching to get on to other Ruby topics. At some point, you'll likely need to do something more complex with **regular expressions** (for example, matching a time by looking for two digits, a colon, and two more digits).

5.2 DIGGING DEEPER

Like most things in Ruby, regular expressions are just objects—they are instances of the class Regexp. This means you can assign them to variables, pass them to methods, and so on:

```
str = "dog and cat"
pattern = /nd/
pattern =~ str # => 5
str =~ pattern # => 5
```

You can also create regular expression objects by calling the Regexp class's new method or by using the %r{...} syntax. The %r syntax is particularly useful when creating patterns that contain forward slashes:

Keyword

Regular expression is a special text string for describing a search pattern. You can think of regular expressions as wildcards on steroids. You are probably familiar with wildcard notations such as *.txt to find all text files in a file manager. The regex equivalent is .*\.txt

```
/mm\/dd/ # => /mm\/dd/
Regexp.new("mm/dd") # => /mm\/dd/
%r{mm/dd} # => /mm\/dd/
```

5.2.1 Regular Expression Options

A regular expression may include one or more options that modify the way the pattern matches strings. If you're using literals to create the Regexp object, then the options are one or more characters placed immediately after the terminator. If you're using Regexp.new, the options are constants used as the second parameter of the constructor.

Remember

Another set of options allows you to set the language encoding of the regular expression. If none of these options is specified, the regular expression will have US-ASCII encoding if it contains only 7-bit characters. Otherwise, it will use the default encoding of the source file containing the literal: n: no encoding (ASCII), e: EUC, s: SJIS, and u: UTF-8.

- i Case insensitive. The pattern match will ignore the case of letters in the pattern and string. (The old technique of setting $= to make matches case insensitive no longer works.)
- o Substitute once. Any #{...} substitutions in a particular regular expression literal will be performed just once, the first time it is evaluated. Otherwise, the substitutions will be performed every time the literal generates a Regexp object.
- m Multiline mode. Normally, "." matches any character except a newline. With the /m option, "." matches any character.
- x Extended mode. Complex regular expressions can be difficult to read. The x option allows you to insert spaces and newlines in the pattern to make it more readable. You can also use # to introduce comments.

Matching Against Patterns

Once you have a regular expression object, you can match it against a string using the (Regexp#match(string) method or the match operators =~ (positive match) and !~ (negative match). The match operators are defined for both String and Regexp objects. One operand of the match operator must be a regular expression

```
name = "Fats Waller"
name =~ /a/                    # => 1
name =~ /z/                    # => nil
/a/ =~ name                    # => 1
/a/.match(name)                # => #<MatchData "a">
Regexp.new("all").match(name) # => #<MatchData "all">
```

The match operators return the character position at which the match occurred, while the match method returns a MatchData object. In all forms, if the match fails, nil is returned. After a successful match, Ruby sets a whole bunch of magic variables.

$& receives the part of the string that was matched by the pattern, $` receives the part of the string that preceded the match, and $' receives the string after the match. However, these particular variables are considered to be fairly ugly, so most Ruby programmers instead use the MatchData object returned from the match method, because it encapsulates all the information Ruby knows about the match.

Given a MatchData object, you can call pre_match to return the part of the string before the match, post_match for the string after the match, and index using [0] to get the matched portion. We can use these methods to write a method, show_regexp, that illustrates where a particular pattern matches:

```
def show_regexp(string, pattern)
match = pattern.match(string)
if match
"#{match.pre_match}->#{match[0]}<-#{match.post_match}"
else
"no match"
end
end
```

We could use this method like this:

```
show_regexp('very interesting', /t/) # => very in->t<-eresting
show_regexp('Fats Waller', /a/) # => F->a<-ts Waller
show_regexp('Fats Waller', /lle/) # => Fats Wa->lle<-r
show_regexp('Fats Waller', /z/) # => no match
```

5.2.2 Deeper Patterns

A pattern, all characters match themselves except ., |, (,), [,], {, }, +, \, ^, $, *, and ?. Let's dig a bit deeper into this. First, always remember that you need to escape any of these characters with a backslash if you want them to be treated as regular characters to match:

show_regexp('yes | no', /\|/) # => yes ->|<- no

show_regexp('yes (no)', /\(no\)/) # => yes ->(no)<-

show_regexp('are you sure?', /e\?/) # => are you sur->e?<-

Now let's see what some of these characters mean if you use them without escaping them.

5.2.3 Literal Characters

The most basic regular expression consists of a single literal character, e.g.: «a». It will match the first occurrence of that character in the string. If the string is "Jack is a boy", it will match the „a" after the "J". The fact that this "a" is in the middle of the word does not matter to the regex engine. If it matters to you, you will need to tell that to the regex engine by using word boundaries.

Sr.No.	Example & Description
1	/ruby/ Matches "ruby".
2	¥ Matches Yen sign. Multibyte characters are supported in Ruby 1.9 and Ruby 1.8.

5.2.4 Character Classes

With a "character class", also called "character set", you can tell the regex engine to match only one out of several characters. Simply place the characters you want to match between square brackets. If you want to match an a or an e, use «**[**ae]». You could use this in «gr[ae]y» to match either „gray" or „grey". Very useful if you do not know whether the document you are searching through is written in American or British English. A character class matches only a single character. «gr[ae]y» will not match "graay", "graey" or any such thing. The order of the characters inside a character class does not matter. The results are identical.

Sr.No.	Example & Description
1	/[Rr]uby/ Matches “Ruby” or “ruby”.
2	/rub[ye]/ Matches “ruby” or “rube”.
3	/[aeiou]/ Matches any one lowercase vowel.
4	/[0-9]/ Matches any digit; same as /[0123456789]/.
5	/[a-z]/ Matches any lowercase ASCII letter.
6	/[A-Z]/ Matches any uppercase ASCII letter.
7	/[a-zA-Z0-9]/ Matches any of the above.
8	/[^aeiou]/ Matches anything other than a lowercase vowel.
9	/[^0-9]/ Matches anything other than a digit.

A character class is a set of characters between brackets: [characters] matches any single character between the brackets, so [aeiou] matches a vowel, [,.:;!?] matches some punctuation, and so on. The significance of the special regular expression characters—.|(){+^$*?—is turned off inside the brackets. However, normal string substitution still occurs, so (for example) \b represents a backspace character, and \n represents a newline. In addition, you can use the **abbreviations** shown in Figure 1, on the following page, so that \s matches any whitespace character, not just a literal space:

```
show_regexp('Price $12.', /[aeiou]/) # => Pr->i<-ce $12.
show_regexp('Price $12.', /[\s]/) # => Price-> <-$12.
show_regexp('Price $12.', /[$.]/) # => Price ->$<-12.
```

Keyword

Abbreviation is a shortened form of a word or phrase. It consists of a group of letters taken from the word or phrase.

Within the brackets, the sequence c1-c2 represents all the characters from c1 to c2 in the current encoding:

a = 'see [The PickAxe-page 123]'

show_regexp(a, /[A-F]/) # => see [The Pick->A<-xe-page 123]

show_regexp(a, /[A-Fa-f]/) # => s->e<-e [The PickAxe-page 123]

show_regexp(a, /[0-9]/) # => see [The PickAxe-page ->1<-23]

show_regexp(a, /[0-9][0-9]/) # => see [The PickAxe-page ->12<-3]

You can negate a character class by putting an up arrow (^, sometimes called a :caret) immediately after the opening bracket

show_regexp('Price $12.', /[^A-Z]/) # => P->r<-ice $12.

show_regexp('Price $12.', /[^\w]/) # => Price-> <-$12.

show_regexp('Price $12.', /[a-z][^a-z]/) # => Pric->e <-$12.

The POSIX character classes, correspond to the ctype(3) macros of the same names. :They can also be negated by putting an up arrow (or caret) after the first colon

show_regexp('Price $12.', /[aeiou]/) # => Pr->i<-ce $12.

show_regexp('Price $12.', /[[:digit:]]/) # => Price $->1<-2.

show_regexp('Price $12.', /[[:space:]]/) # => Price-> <-$12.

show_regexp('Price $12.', /[[:^alpha:]]/) # => Price-> <-$12.

show_regexp('Price $12.', /[[:punct:]aeiou]/) # => Pr->i<-ce $12.

If you want to include the literal characters] and - within a character class, escape :\ them with

a = 'see [The PickAxe-page 123]'

show_regexp(a, /[\]]/) # => see [The PickAxe-page 123->]<-

show_regexp(a, /[0-9\]]/) # => see [The PickAxe-page ->1<-23]

show_regexp(a, /[\d\-]/) # => see [The PickAxe->-<-page 123]

Some character classes are used so frequently that Ruby provides abbreviations for them. These abbreviations are listed in Figure 1, on the following page—they may be used both within

brackets and in the body of a pattern.

show_regexp('It costs $12.', /\s/) # => It-> <-costs $12.

show_regexp('It costs $12.', /\d/) # => It costs $->1<-2.

Sequence	As [...]	Meaning
\d	[0-9]	ASCII decimal digit character
\D	[^0-9]	Any character except a digit
\h	[0-9a-fA-F]	Hexadecimal digit character
\H	[^0-9a-fA-F]	Any character except a hex digit
\s	[␣\t\r\n\f]	ASCII whitespace character
\S	[^␣\t\r\n\f]	Any character except whitespace
\w	[A-Za-z0-9_]	ASCII word character
\W	[^A-Za-z0-9_]	Any character except a word character

Figure 1: Character class abbreviations

You can create the intersection of character classes using &&. So, to match all lowercase ASCII letters that aren't vowels, you could use this:

str = "now is the time"

str.gsub(/[a-z&&[^aeiou]]/, '*') # => "*o* i* **e *i*e"

The \p construct is new with Ruby 1.9. It gives you an encoding-aware way of matching a 1.9 character with a particular Unicode property:

encoding: utf-8

string = "∂y/∂x = 2πx"

show_regexp(string, /\p{Alnum}/) # => ∂->y<-/∂x = 2πx

show_regexp(string, /\p{Digit}/) # => ∂y/∂x = ->2<-πx

show_regexp(string, /\p{Space}/) # => ∂y/∂x-> <-= 2πx

show_regexp(string, /\p{Greek}/) # => ∂y/∂x = 2->π<-x

show_regexp(string, /\p{Graph}/) # => ->∂<-y/∂x = 2πx

Finally, a period (.) appearing outside brackets represents any character except a newline (though in multiline mode it matches a newline, too):

a = 'It costs $12.'

show_regexp(a, /c.s/) # => It ->cos<-ts $12.

show_regexp(a, /./) # => ->I<-t costs $12.

show_regexp(a, /\./) # => It costs $12->.<-

5.2.5 Special Character Classes

Because we want to do more than simply search for literal pieces of text, we need to reserve certain characters for special use. In the regex flavors, there are 11 characters with special meanings: the opening square bracket «**[**», **the backslash** «\», **the caret** «^», **the dollar sign** «$», **the period or dot** «.», **the vertical bar or pipe symbol** «|», **the question mark** «?», **the asterisk or star** «*», **the plus sign** «+», **the opening round bracket** «(» **and the closing round bracket** «)». **These special characters are often called** "metacharacters". If you want to use any of these characters as a literal in a regex, you need to escape them with a backslash. If you want to match „1+1=2", the correct regex is «1\+1=2». Otherwise, the plus sign will have a special meaning. Note that «1+1=2», with the backslash omitted, is a valid regex. So you will not get an error message. But it will not match "1+1=2". It would match „111=2" in "123+111=234", due to the special meaning of the plus character.

Sr.No.	Example & Description
1	/./ Matches any character except newline.
2	/./m In multi-line mode, matches newline, too.
3	/\d/ Matches a digit: /[0-9]/.
4	/\D/ Matches a non-digit: /[^0-9]/.
5	/\s/ Matches a whitespace character: /[\t\r\n\f]/.
6	/\S/ Matches non-whitespace: /[^ \t\r\n\f]/.
7	/\w/ Matches a single word character: /[A-Za-z0-9_]/.
8	/\W/ Matches a non-word character: /[^A-Za-z0-9_]/.

5.2.6 Repetition Cases

When we specified the pattern that split the song list line, /\s*\|\s*/, we said we wanted to match a vertical bar surrounded by an arbitrary amount of whitespace. We now know that the \s sequences match a single whitespace character and \| means a literal vertical bar, so it seems likely that the asterisks somehow mean "an arbitrary amount." In fact, the asterisk is one of a number of modifiers that allow you to match multiple occurrences of a pattern.

Sr.No.	Example & Description
1	/ruby?/ Matches "rub" or "ruby": the y is optional.
2	/ruby*/ Matches "rub" plus 0 or more ys.
3	/ruby+/ Matches "rub" plus 1 or more ys.
4	/\d{3}/ Matches exactly 3 digits.
5	/\d{3,}/ Matches 3 or more digits.
6	/\d{3,5}/ Matches 3, 4, or 5 digits.

If r stands for the immediately preceding regular expression within a pattern, then

r* Matches zero or more occurrences of r

r+ Matches one or more occurrences of r

r? Matches zero or one occurrence of r

r{m,n} Matches at least m and at most n occurrences of r

r{m,} Matches at least m occurrences of r

r{,n} Matches at most n occurrences of r

r{m} Matches exactly m occurrences of r

These repetition constructs have a high precedence—they bind only to the immediately preceding matching construct in the pattern. /ab+/ matches an a followed by one or more b's, not a sequence of ab's. These patterns are called greedy, because

Keyword

Suffixes are a letter or group of letters added to the ending of words to change their meaning or function.

by default they will match as much of the string as they can. You can alter this behavior and have them match the minimum by adding a question mark **suffix**. The repetition is then called lazy—it stops once it has done the minimum amount of work required.

```
a = "The moon is made of cheese"
show_regexp(a, /\w+/) # => ->The<- moon is made of cheese
show_regexp(a, /\s.*\s/) # => The-> moon is made of <-cheese
show_regexp(a, /\s.*?\s/) # => The-> moon <-is made of cheese
show_regexp(a, /[aeiou]{2,99}/) # => The m->oo<-n is made of cheese
show_regexp(a, /mo?o/) # => The ->moo<-n is made of cheese
# here's the lazy version
show_regexp(a, /mo??o/) # => The ->mo<-on is made of cheese
```

(There's an additional modifier, +, that makes them greedy and also stops backtracking.) Be very careful when using the * modifier. It matches zero or more occurrences. We often forget about the zero part. In particular, a pattern that contains just a * repetition will always match, whatever string you pass it. For example, the pattern /a*/ will always match, because every string contains zero or more a's.

```
a = "The moon is made of cheese"
# both of these match an empty substring at the start of the string
show_regexp(a, /m*/) # => -><-The moon is made of cheese
show_regexp(a, /Z*/) # => -><-The moon is made of cheese
```

Non-greedy Repetition: This matches the smallest number of repetitions −

Sr.No.	Example & Description
1	/<.*>/ Greedy repetition: matches “<ruby>perl>”.
2	/<.*?>/ Non-greedy: matches “<ruby>” in “<ruby>perl>”.

5.2.7 Grouping with Parentheses

You can use parentheses to group terms within a regular expression. Everything within the group is treated as a single regular expression

```
# This matches an ‘a’ followed by one or more ‘n’s
show_regexp(‘banana’, /an+/) # => b->an<-ana
# This matches the sequence ‘an’ one or more times
show_regexp(‘banana’, /(an)+/) # => b->anan<-a
a = ‘red ball blue sky’
show_regexp(a, /blue|red/) # => ->red<- ball blue sky
show_regexp(a, /(blue|red) \w+/) # => ->red ball<- blue sky
show_regexp(a, /(red|blue) \w+/) # => ->red ball<- blue sky
show_regexp(a, /red|blue \w+/) # => ->red<- ball blue sky
show_regexp(a, /red (ball|angry) sky/) # => no match
a = ‘the red angry sky’
show_regexp(a, /red (ball|angry) sky/) # => the ->red angry sky<-
```

Sr.No.	Example & Description
1	/\D\d+/ No group: + repeats \d
2	/(\D\d)+/ Grouped: + repeats \D\d pair
3	/([Rr]uby(,)?)+/ Match “Ruby”, “Ruby, ruby, ruby”, etc.

Parentheses also collect the results of pattern matching. Ruby counts opening parentheses and for each stores the result of the partial match between it and the corresponding closing parenthesis. You can use this partial match both within the rest

of the pattern and in your Ruby program. Within the pattern, the sequence \1 refers to the match of the first group, \2 the second group, and so on. Outside the pattern, the special variables $1, $2, and so on, serve the same purpose.

```
/(\d\d):(\d\d)(..)/ =~ "12:50am" # => 0
"Hour is #$1, minute #$2" # => "Hour is 12, minute 50"
/((\d\d):(\d\d))(..)/ =~ "12:50am" # => 0
"Time is #$1" # => "Time is 12:50"
"Hour is #$2, minute #$3" # => "Hour is 12, minute 50"
"AM/PM is #$4" # => "AM/PM is am"
```

If you're using the MatchData object returned by the match method, you can index into it to get the corresponding subpatterns:

```
md = /(\d\d):(\d\d)(..)/.match("12:50am")
"Hour is #{md[1]}, minute #{md[2]}" # => "Hour is 12, minute 50"
md = /((\d\d):(\d\d))(..)/.match("12:50am")
"Time is #{md[1]}" # => "Time is 12:50"
"Hour is #{md[2]}, minute #{md[3]}" # => "Hour is 12, minute 50"
"AM/PM is #{md[4]}" # => "AM/PM is am"
```

The ability to use part of the current match later in that match allows you to look for various forms of repetition:

```
# match duplicated letter
show_regexp('He said "Hello"', /(\w)\1/) # => He said "He->ll<-o"
# match duplicated substrings
show_regexp('Mississippi', /(\w+)\1/) # => M->ississ<-ippi
```

Rather than use numbers, you can also use names to refer to matched content. You 1.9 give a group a name by placing ? immediately after the opening parenthesis. You can subsequently refer to this named group using \k (or \k'name').

```
# match duplicated letter
str = 'He said "Hello"'
show_regexp(str, /(?<char>\w)\k<char>/) # => He said "He->ll<-o"
# match duplicated adjacent substrings
str = 'Mississippi'
show_regexp(str, /(?<seq>\w+)\k<seq>/) # => M->ississ<-ippi
```

The named matches in a regular expression are also available as local variables:

```
/(?<hour>\d\d):(?<min>\d\d)(..)/ =~ "12:50am" # => 0
```

"Hour is #{hour}, minute #{min}" # => "Hour is 12, minute 50"

Once you use named matches in a particular regular expression, Ruby no longer bothers to capture unnamed groups.

5.2.8 Alternatives

We know that the vertical bar is special, because our line-splitting pattern had to escape it with a backslash. That's because an unescaped vertical bar, as in |, matches either the construct that precedes it or the construct that follows it:

a = "red ball blue sky"

show_regexp(a, /d|e/) # => r->e<-d ball blue sky

show_regexp(a, /al|lu/) # => red b->al<-l blue sky

show_regexp(a, /red ball|angry sky/) # => ->red ball<- blue sky

There's a trap for the unwary here, because | has a very low precedence. The last example in the previous lines matches red ball or angry sky, not red ball sky or red angry sky. To match red ball sky or red angry sky, you'd need to override the default precedence using grouping.

Sr.No.	Example & Description
1	/ruby|rube/ Matches "ruby" or "rube".
2	/rub(y|le))/ Matches "ruby" or "ruble".
3	/ruby(!+|\?)/ "ruby" followed by one or more ! or one ?

Keyword

String is a data type used in programming, such as an integer and floating point unit, but is used to represent text rather than numbers. It is comprised of a set of characters that can also contain spaces and numbers.

5.2.9 Anchors

It needs to specify match position. By default, a regular expression will try to find the first match for the pattern in a string. Match /iss/ against the **string** "Mississippi," and it will find the substring "iss" starting at position 1 (the second character in the string). But what if you want to force a pattern to match only at the start or end of a string?

Sr.No.	Example & Description
1	/^Ruby/ Matches “Ruby” at the start of a string or internal line.
2	/Ruby$/ Matches “Ruby” at the end of a string or line.
3	/\ARuby/ Matches “Ruby” at the start of a string.
4	/Ruby\Z/ Matches “Ruby” at the end of a string.
5	/\bRuby\b/ Matches “Ruby” at a word boundary.
6	/\brub\B/ \B is non-word boundary: matches “rub” in “rube” and “ruby” but not alone.
7	/Ruby(?=!)/ Matches “Ruby”, if followed by an exclamation point.
8	/Ruby(?!!)/ Matches “Ruby”, if not followed by an exclamation point.

The patterns ^ and $ match the beginning and end of a line, respectively. These are often used to anchor a pattern match; for example, /^option/ matches the word option only if it appears at the start of a line. Similarly, the sequence \A matches the beginning of a string, and \z and \Z match the end of a string. (Actually, \Z matches the end of a string unless the string ends with \n, in which case it matches just before the \n.)

```
str = “this is\nthe time”
show_regexp(str, /^the/) # => this is\n->the<- time
show_regexp(str, /is$/) # => this ->is<-\nthe time
show_regexp(str, /\Athis/) # => ->this<- is\nthe time
show_regexp(str, /\Athe/) # => no match
```

Similarly, the patterns \b and \B match word boundaries and nonword boundaries, respectively. Word characters are ASCII letters, numbers, and underscores:

show_regexp("this is\nthe time", /\bis/) # => this ->is<-\nthe time

show_regexp("this is\nthe time", /\Bis/) # => th->is<- is\nthe time

5.2.10 Pattern-Based Substitution

We've already seen how sub and gsub replace the matched part of a string with other text. In those previous examples, the pattern was always fixed text, but the substitution methods work equally well if the pattern contains repetition, alternation, and grouping.

```
a = "quick brown fox"
a.sub(/[aeiou]/, '*') # => "q*ick brown fox"
a.gsub(/[aeiou]/, '*') # => "q**ck br*wn f*x"
a.sub(/\s\S+/, '') # => "quick fox"
a.gsub(/\s\S+/, '') # => "quick"
```

The substitution methods can take a string or a block. If a block is used, it is passed the matching substring, and the block's value is substituted into the original string.

```
a = "quick brown fox"
a.sub(/^./) {|match| match.upcase } # => "Quick brown fox"
a.gsub(/[aeiou]/) {|vowel| vowel.upcase } # => "qUIck brOwn fOx"
```

Maybe we want to normalize names entered by users into a web application. They may enter DAVE THOMAS, dave thomas, or dAvE tHoMas, and we'd like to store it as Dave Thomas. The following method is a simple first **iteration**. The pattern that matches the first character of a word is \b\w—look for a word boundary followed by a word character.

Combine this with gsub, and we can hack the names:

```
def mixed_case(name)
name.downcase.gsub(/\b\w/) {|first| first.upcase }
end
mixed_case("DAVE THOMAS") # => "Dave Thomas"
mixed_case("dave thomas") # => "Dave Thomas"
mixed_case("dAvE tHoMas") # => "Dave Thomas"
```

Keyword

Iteration is the act of repeating a process, to generate a sequence of outcomes, with the aim of approaching a desired goal, target or result.

Remember

Methods like String#scan, String#split, Enumerable#grep, and the "sub" family of String methods use regular expressions and pattern matching as a way of determining how their actions should be applied. Gaining knowledge about regular expressions gives you access not only to relatively simple matching methods but also to a suite of string-handling tools that otherwise would not be usable.

There's an idiomatic way to write the substitution in Ruby, the symbol to_proc Trick, why it works

```
def mixed_case(name)
name.downcase.gsub(/\b\w/, &:upcase)
end
mixed_case("dAvE tHoMas") # => "Dave Thomas"
```

You can also give sub and gsub a hash as the replacement parameter, in which case they will look up matched groups and use the corresponding values as replacement text:

```
replacement = { "cat" => "feline", "dog" => "canine" }
replacement.default = "unknown"
"cat and dog".gsub(/\w+/, replacement) # => "feline unknown canine"
```

Did You Know?

Ruby 1.9 was released on Christmas Day in 2007. Effective with Ruby 1.9.3, released October 31, 2011, Ruby switched from being dual-licensed under the Ruby License and the GPL to being dual-licensed under the Ruby License and the two-clause BSD license. Adoption of 1.9 was slowed by changes from 1.8 that required many popular third party gems to be rewritten.

5.2.11 Backslash Sequences in the Substitution

Earlier we noted that the sequences \1, \2, and so on, are available in the pattern, standing for the nth group matched so far. The same sequences can be used in the second argument of sub and gsub.

```
puts "fred:smith".sub(/(\w+):(\w+)/, '\2, \1')
puts "nercpyitno".gsub(/(.)(.)/, '\2\1')
```

produces:

```
smith, fred
encryption
```

You can also reference named groups:

```
puts "fred:smith".sub(/(?<first>\w+):(?<last>\w+)/, '\k<last>, \k<first>')
puts "nercpyitno".gsub(/(?<c1>.)(?<c2>.)/, '\k<c2>\k<c1>')
```

produces:

```
smith, fred
encryption
```

Additional backslash sequences work in substitution strings: \& (last match), \+ (last matched group), \` (string prior to match), \' (string after match), and \\ (a literal backslash).

It gets confusing if you want to include a literal backslash in a substitution. The obvious thing is to write this:

```
str.gsub(/\\/, '\\\\')
```

Clearly, this code is trying to replace each backslash in str with two. The programmer doubled up the backslashes in the replacement text, knowing that they'd be converted to \\ in syntax analysis. However, when the substitution occurs, the regular expression engine performs another pass through the string, converting \\ to \, so the net effect is to replace each single backslash with another single backslash. You need to write !('\\\\\\\\\' ,/\\/)gsub

```
str = 'a\b\c' # => "a\b\c"
str.gsub(/\\/, '\\\\\\\\') # => "a\\b\\c"
```

However, using the fact that \& is replaced by the matched string, you could also write this:

```
str = 'a\b\c' # => "a\b\c"
str.gsub(/\\/, '\&\&') # => "a\\b\\c"
```

If you use the block form of gsub, the string for substitution is analyzed only once (during the syntax pass), and the result is what you intended:

```
str = 'a\b\c' # => "a\b\c"
str.gsub(/\\/) { '\\\\' } # => "a\\b\\c"
```

SUMMARY

- A regular expression is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings.
- Ruby regular expressions i.e. Ruby regex for short, helps us to find particular patterns inside a string. Two uses of ruby regex are validation and parsing.
- There are many ways of creating a regular expression pattern. By far the most common is to write it between forward slashes.
- Because pattern matching returns nil when it fails and because nil is equivalent to false in a Boolean context, you can use the result of a pattern match as a condition in statements such as if and while.
- The most basic regular expression consists of a single literal character, e.g.: «a». It will match the first occurrence of that character in the string.
- You can use parentheses to group terms within a regular expression. Everything within the group is treated as a single regular expression.
- A character class is a set of characters between brackets: [characters] matches any single character between the brackets, so [aeiou] matches a vowel, [,.:;!?] matches some punctuation, and so on.
- Some of the most important String methods that use regular expressions are sub and gsub, and their in-place variants sub! and gsub!.

KNOWLEDGE CHECK

1. **Regular expressions are used to represent which language**
   a. Recursive language
   b. Context free language
   c. Regular language
   d. All of these
2. **Which of the following operation can be applied on regular expressions?**
   a. Union
   b. Concatenation
   c. Closure
   d. All of these
3. **Which of the following identity is wrong?**
   a. R + R = R
   b. (R*)* = R*
   c. εR = Rε = R
   d. ØR = RØ = RR*
4. **Which of the following statement is true?**
   a. Every language that is defined by regular expression can also be defined by finite automata
   b. Every language defined by finite automata can also be defined by regular expression
   c. We can convert regular expressions into finite automata
   d. All of these
5. **L is a regular Language if and only If the set of _____ classes of IL is finite.**
   a. Equivalence
   b. Reflexive
   c. Myhill
   d. Nerode
6. **Regular expression {0,1} is equivalent to**
   a. 0 U 1
   b. 0/1

c. 0+1

d. Al of the above

7. Which of the following languages have built in regexps support?

a. Ruby

b. Java

c. Python

d. C++

REVIEW QUESTIONS

1. Discuss about the regular-expression modifiers.
2. What are regular-expression patterns?
3. Define the regular expression options.
4. Discuss on the role of character classes.
5. What do you understand by the special character classes?
6. How to group with parentheses in ruby?
7. Determine the pattern-based substitution.

Check Your Result

1. (c) 2. (d) 3. (d) 4. (d) 5. (a)
6. (d) 7. (a)

REFERENCES

1. Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman. Compilers - Principles, Techniques and Tools. Addison Wesley, second edition, 2007.
2. B. Chess and G. McGraw. Static analysis for security. Security & Privacy, IEEE, 2(6):76–79, 2004.
3. Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In ACM PLDI. ACM Press, 2000.
4. Gavin Bierman, Matthew Parkinson, and Andrew Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical report, Cambridge University, 2003.
5. http://web.cse.ohio-state.edu/~joseph.97/courses/3901/lectures/lecture09.pdf
6. http://www.rubyguides.com/2015/06/ruby-regex/
7. https://bitcetera.com/page_attachments/0000/0030/regex_in_a_nutshell.pdf
8. https://dgrisham.github.io/slides/ruby/3-regex/slides.pdf
9. https://doc.lagout.org/programmation/Regular%20Expressions/Regular%20Expressions%20Cookbook_%20Detailed%20Solutions%20in%20Eight%20Programming%20Languages%20%282nd%20ed.%29%20%5BGoyvaerts%20%26%20Levithan%202012-09-06%5D.pdf
10. https://media.pragprog.com/titles/ruby3/ruby3_extract_regular_expressions.pdf
11. https://www.geos.ed.ac.uk/~bmg/software/Perl%20Books/RegExp_perl_python_java_etc.pdf
12. https://www.princeton.edu/~mlovett/reference/Regular-Expressions.pdf
13. https://www.tutorialspoint.com/ruby/ruby_regular_expressions.htm
14. J. Berdine, B. Cook, D. Distefano, and P. OHearn. Automatic termination proofs for programs with shape-shifting heaps. In Computer Aided Verification, pages 386–400. Springer, 2006.
15. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. Java(TM) Language Specification, Third Edition. Addison-Wesley, 2005.
16. Russ Cox. Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, Php, Python, Ruby, ...). Available at http://swtch.com/~rsc/regexp/regexp1.html, January 2007.
17. Russ Cox. Regular expression matching: the virtual machine approach. Available at http://swtch.com/~rsc/regexp/regexp2.html, December 2009.
18. Wei-Ngan Chin, Florin Craciun, Shengchao Qin, and Martin Rinard. Region Inference for an Object-Oriented Language. In ACM PLDI, Washington, 2004.

CHAPTER

6

RUBY: OBJECT-ORIENTED PROGRAMMING

"This is great stuff! Your descriptions are so vibrant and vivid that I'm rediscovering the truth buried in OO principles that are otherwise so internalized that I forget to explore them. Your thoughts on design and knowing the future are especially eloquent."

–Ian McFarland

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Focus on ruby class
2. Discuss about class inheritance

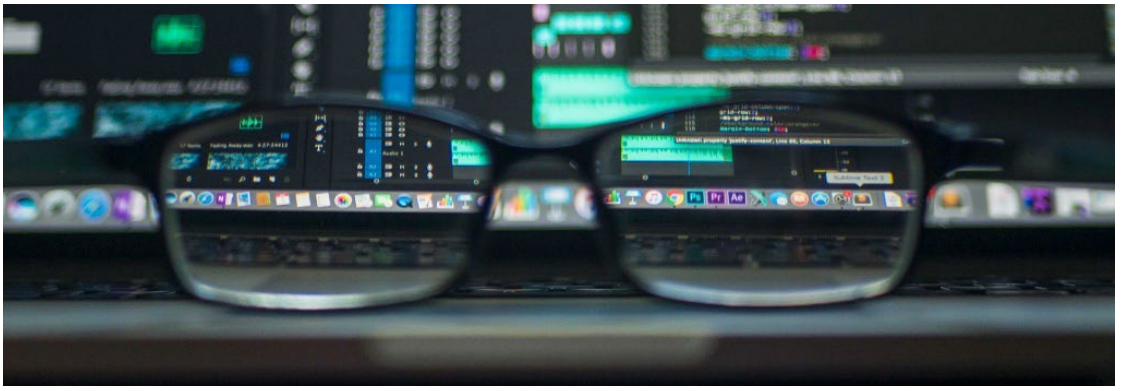
INTRODUCTION

Ruby is an object-oriented programming language (OOP) that uses classes as blueprints for objects. Objects are the basic building-blocks of Ruby code (everything in Ruby is

an object), and have two main properties: states and behaviors. Ruby classes are the blueprints that establish what attributes (also known as states) and behaviors (known in Ruby as methods) that an object should have.

Ruby is a true object oriented language which can be embedded into hypertext markup language. Everything in Ruby is an object. All the numbers, strings or even class is an object. The whole Ruby language is basically built on the concepts of object and data.

OOPs is a programming concept that uses objects and their interactions to design applications and computer programs.



6.1 DEFINITION OF RUBY CLASS

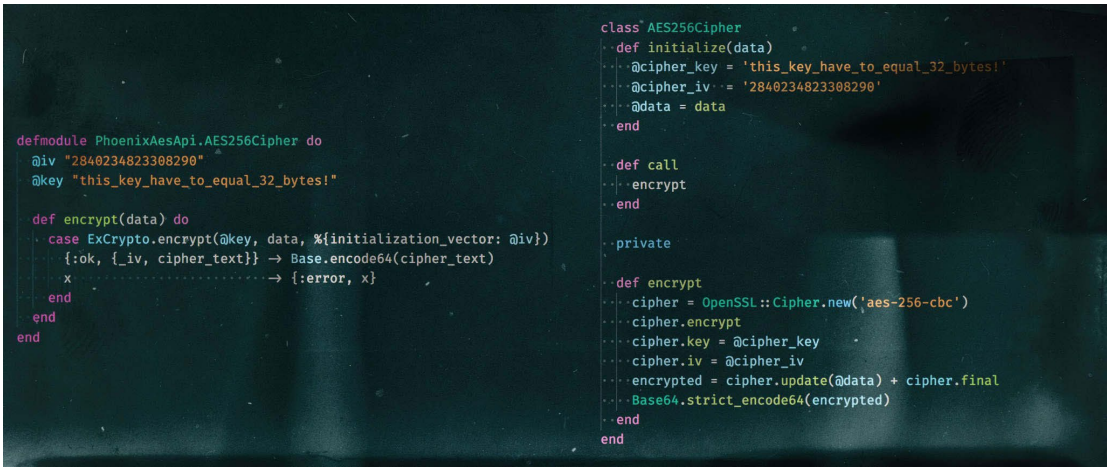
When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword **class** followed by the **class name** and is delimited with an **end**.

For example, we defined the Box class using the keyword class as follows –

```
class Box
   code
end
```

The name must begin with a capital letter and by convention names that contain more than one word are run together with each word capitalized and no separating characters (CamelCase).



6.1.1 Define Ruby Objects

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class using **new** keyword. Following statements declare two objects of class Box −

box1 = Box.new

box2 = Box.new

The initialize Method

The initialize method is a standard Ruby class method and works almost same way as **constructor** works in other object oriented programming languages. The initialize

method is useful when you want to initialize some class variables at the time of object creation. This method may take a list of parameters and like any other ruby method it would be preceded by **def** keyword as shown below −

```
class Box
   def initialize(w,h)
      @width, @height = w, h
   end
end
```

Keyword

A constructor is a special method that is used to initialize a newly created object and is called just after the memory is allocated for the object.

The instance Variables

The instance variables are kind of class attributes and they become properties of objects once objects are created using the class. Every object›s attributes are assigned individually and share no value with other objects. They are accessed using the @ operator within the class but to access them outside of the class we use **public** methods, which are called **accessor methods**. If we take the above defined class **Box** then @width and @height are instance variables for the class Box.

```
class Box
   def initialize(w,h)
      # assign instance variables
      @width, @height = w, h
   end
end
```

6.1.2 The accessor & setter Methods

To make the variables available from outside the class, they must be defined within **accessor methods**, these accessor methods are also known as a getter methods. Following example shows the usage of accessor methods −

```
#!/usr/bin/ruby -w
# define a class
class Box
   # constructor method
   def initialize(w,h)
      @width, @height = w, h
   end
   # accessor methods
   def printWidth
      @width
   end

   def printHeight
      @height
   end
end

# create an object
box = Box.new(10, 20)

# use accessor methods
x = box.printWidth()
y = box.printHeight()

puts “Width of the box is : #{x}”
puts “Height of the box is : #{y}”
```

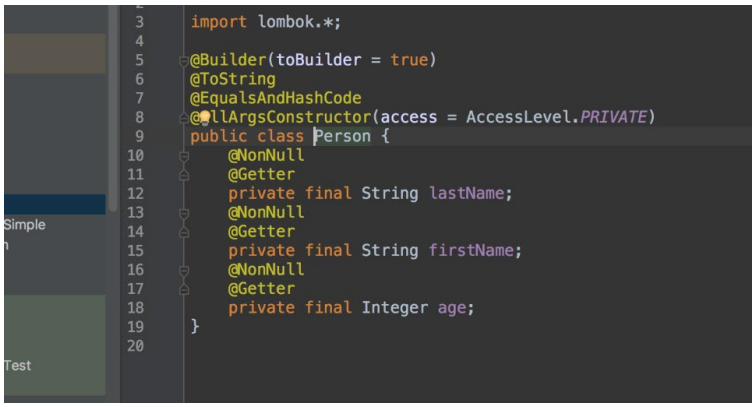
When the above code is executed, it produces the following result −

```
Width of the box is : 10
Height of the box is: 20
```

Similar to accessor methods, which are used to access the value of the variables, Ruby provides a way to set the values of those variables from outside of the class using **setter methods**, which are defined as below #!/usr/bin/ruby -w

Keyword

Provide public **setter** and getter **methods** to modify and view the variables values.




```
# define a class
class Box
   # constructor method
   def initialize(w,h)
      @width, @height = w, h
   end

   # accessor methods
   def getWidth
      @width
   end
   def getHeight
      @height
   end

   # setter methods
   def setWidth=(value)
      @width = value
   end
```

```
   def setHeight=(value)
      @height = value
   end
end

# create an object
box = Box.new(10, 20)

# use setter methods
box.setWidth = 30
box.setHeight = 50

# use accessor methods
x = box.getWidth()
y = box.getHeight()

puts “Width of the box is : #{x}”
puts “Height of the box is : #{y}”
```

When the above code is executed, it produces the following result −

```
Width of the box is : 30
Height of the box is : 50
```

The instance Methods

The instance methods are also defined in the same way as we define any other method using **def** keyword and they can be used using a class instance only as shown below. Their functionality is not limited to access the instance variables, but also they can do a lot more as per your requirement.

```
#!/usr/bin/ruby -w

# define a class
class Box
   # constructor method
   def initialize(w,h)
      @width, @height = w, h
```

```
    end
    # instance method
    def getArea
       @width * @height
    end
end

# create an object
box = Box.new(10, 20)

# call instance methods
a = box.getArea()
puts "Area of the box is : #{a}"
```

When the above code is executed, it produces the following result −

```
Area of the box is : 200
```

Keyword

A **delimiter** is a sequence of one or more characters used to specify the boundary between separate, independent regions in plain text or other data streams.

6.1.3 The class Methods and Variables

The **class variables** is a variable, which is shared between all instances of a class. In other words, there is one instance of the variable and it is accessed by object instances. Class variables are prefixed with two @ characters (@@). A class variable must be initialized within the class definition as shown below. A class method is defined using **def self.methodname()**, which ends with end **delimiter** and would be called using the class name as **classname.methodname** as shown in the following example −

```
#!/usr/bin/ruby -w
```

```
class Box
   # Initialize our class variables
   @@count = 0
   def initialize(w,h)
      # assign instance avriables
      @width, @height = w, h

      @@count += 1
   end

   def self.printCount()
      puts "Box count is : #@@count"
   end
end
# create two object
box1 = Box.new(10, 20)
box2 = Box.new(30, 100)

# call class method to print box count
Box.printCount()
```

When the above code is executed, it produces the following result −

```
Box count is: 2
```

6.1.4 The to_s Method

Any class you define should have a to_s instance method to return a string representation of the object. Following is a simple example to represent a Box object in terms of width and height −

```
#!/usr/bin/ruby -w

class Box
   # constructor method
   def initialize(w,h)
```

Did You Know?

In 2001, Laing and Coleman examined several NASA Goddard Space Flight Center applications (rocket science) with the express intention of finding "a way to produce cheaper and higher quality software."

```
    @width, @height = w, h
  end
  # define to_s method
  def to_s
    "(w:#@width,h:#@height)"  # string formatting of the object.
  end
end

# create an object
box = Box.new(10, 20)

# to_s method will be called in reference of string automatically.
puts "String representation of box is : #{box}"
```

When the above code is executed, it produces the following result –

String representation of box is: (w:10,h: 20)

```
class Document

attr_reader :name

def initialize (arg={})
@name=arg[:yourname]
end

def to_s
"name is #{name}"
end
end

doc=Document.new(yourname:'tom')

puts doc
```

```
#<Document:0x00000002e515f8>
[Finished in 0.3s]
```

6.1.5 Access Control

Ruby gives you three levels of protection at instance methods level, which may be **public, private, or protected**. Ruby does not apply any access control over instance and class variables.

- **Public Methods**: Public methods can be called by anyone. Methods are public by default except for initialize, which is always private.

- **Private Methods**: Private methods cannot be accessed, or even viewed from outside the class. Only the class methods can access private members.
- **Protected Methods**: A protected method can be invoked only by objects of the defining class and its subclasses. Access is kept within the family.

Following is a simple example to show the syntax of all the three access modifiers −

```
#!/usr/bin/ruby -w
# define a class
class Box
   # constructor method
   def initialize(w,h)
      @width, @height = w, h
   end
   # instance method by default it is public
   def getArea
      getWidth() * getHeight
   end
   # define private accessor methods
   def getWidth
      @width
   end
   def getHeight
      @height
   end
   # make them private
   private :getWidth, :getHeight
   # instance method to print area
   def printArea
      @area = getWidth() * getHeight
      puts "Big box area is : #@area"
   end
   # make it protected
```

Remember

If Agile is correct, two other things are also true. First, there is absolutely no point in doing a Big Up Front Design (BUFD) (because it cannot possibly be correct), and second, no one can predict when the application will be done (because you don't know in advance what it will eventually do).

```
    protected :printArea
end
# create an object
box = Box.new(10, 20)
# call instance methods
a = box.getArea()
puts "Area of the box is : #{a}"

# try to call protected or methods
box.printArea()
```

When the above code is executed, it produces the following result. Here, first method is called successfully but second method gave a problem.

```
Area of the box is : 200
test.rb:42: protected method `printArea' called for #
<Box:0xb7f11280 @height=20, @width=10> (NoMethodError)
```

Did You Know?

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class or superclass, and the new class is referred to as the derived class or sub-class.

6.2 CLASS INHERITANCE

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. Inheritance also provides an opportunity to reuse the code functionality and fast implementation time but unfortunately Ruby does not support multiple levels of inheritances but Ruby supports **mixins**. A mixin is like a specialized implementation of multiple inheritance in which only the interface portion is inherited.

Ruby also supports the concept of sub classing, i.e., inheritance and following example explains the concept. The syntax for extending a class is simple. Just add a < character and the name of the superclass to your class statement. For example, following define a class *BigBox* as a subclass of *Box* –

```
class ComicBookCharacter {
    constructor(
        public alias: string, public health: number, public strength: number,
        protected secretIdentity: string
    ) {}
}

class SuperHero extends ComicBookCharacter {
    traits = ["empathy", "strong moral code"];
    getSecretId() { console.log(this.secretIdentity); }
}

class SuperVillain extends ComicBookCharacter {
    flaws = ["hubris", "always explains evil plan"];

    constructor() {
        Missed superclass's constructor invocation{this.alias} eats kittens!!!`);
    }
}

let jubilee = new SuperHero("Jubilee", 23, 233, "Jubilation Lee");
let scarletWitch = new SuperVillain("Scarlet Witch", 233, 4444, "Wanda Maximoff");

console.log(jubilee.getSecretId());
// console.log("////");
```

```
#!/usr/bin/ruby -w
# define a class
class Box
   # constructor method
   def initialize(w,h)
      @width, @height = w, h
   end
   # instance method
   def getArea
      @width * @height
   end
end
# define a subclass
class BigBox < Box

   # add a new instance method
   def printArea
      @area = @width * @height
      puts “Big box area is : #@area”
   end
end
# create an object
box = BigBox.new(10, 20)
```

```
# print the area
box.printArea()
```

When the above code is executed, it produces the following result –

```
Big box area is: 200
```

Keyword

Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.

6.2.1 Methods Overriding

Though you can add new functionality in a derived class, but sometimes you would like to change the behavior of already defined method in a parent class. You can do so simply by keeping the method name same and **overriding** the functionality of the method as shown below in the example –

```
#!/usr/bin/ruby -w
# define a class
class Box
   # constructor method
   def initialize(w,h)
      @width, @height = w, h
   end
   # instance method
   def getArea
      @width * @height
   end
end

# define a subclass
class BigBox < Box

   # change existing getArea method as follows
   def getArea
      @area = @width * @height
      puts “Big box area is : #@area”
   end
end
```

```
# create an object
box = BigBox.new(10, 20)

# print the area using overriden method.
box.getArea()
```

6.2.2 Operator Overloading

We'd like the + operator to perform vector addition of two Box objects using +, the * operator to multiply a Box width and height by a scalar, and the unary - operator to do negate the width and height of the Box. Here is a version of the Box class with mathematical operators defined −

```
class Box
   def initialize(w,h)      # Initialize the width and height
      @width,@height = w, h
   end

   def +(other)         # Define + to do vector addition
      Box.new(@width + other.width, @height + other.height)
   end

   def -@               # Define unary minus to negate width and height
      Box.new(-@width, -@height)
   end

   def *(scalar)              # To perform scalar multiplication
      Box.new(@width*scalar, @height*scalar)
   end
end
```

```ruby
require "http/server"

server = HTTP::Server.new(8080) do |context|
  context.response.content_type = "text/plain"
  context.response.print "Hello world, got #{context.request.path}!"
end

puts "Listening on http://127.0.0.1:8080"
server.listen
```

6.2.3 Freezing Objects

Sometimes, we want to prevent an object from being changed. The freeze method in Object allows us to do this, effectively turning an object into a constant. Any object can be frozen by invoking **Object.freeze**. A frozen object may not be modified: you can't change its instance variables.

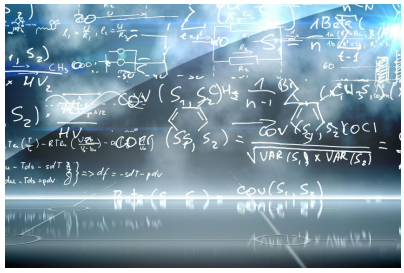
You can check if a given object is already frozen or not using **Object.frozen?**method, which returns true in case the object is frozen otherwise a false value is return. Following example clears the concept –

```ruby
#!/usr/bin/ruby -w

# define a class
class Box
   # constructor method
   def initialize(w,h)
      @width, @height = w, h
   end

   # accessor methods
   def getWidth
      @width
   end
   def getHeight
      @height
   end

   # setter methods
   def setWidth=(value)
```

```
      @width = value
   end
   def setHeight=(value)
      @height = value
   end
end

# create an object
box = Box.new(10, 20)

# let us freez this object
box.freeze
if( box.frozen? )
   puts "Box object is frozen object"
else
   puts "Box object is normal object"
end

# now try using setter methods
box.setWidth = 30
box.setHeight = 50

# use accessor methods
x = box.getWidth()
y = box.getHeight()

puts "Width of the box is : #{x}"
puts "Height of the box is : #{y}"
```

When the above code is executed, it produces the following result −

```
Box object is frozen object
test.rb:20:in `setWidth=': can't modify frozen object (TypeError)
   from test.rb:39
```



Keyword

A convention, in the sense of a meeting, is a gathering of individuals who meet at an arranged place and time in order to discuss or engage in some common interest.

6.2.4 Class Constants

You can define a constant inside a class by assigning a direct numeric or string value to a variable, which is defined without using either @ or @@. By **convention**, we keep constant names in upper case. Once a constant is defined, you cannot change its value but you can access a constant directly inside a class much like a variable but if you want to access a constant outside of the class then you would have to use class name::constant as shown in the below example.

```
#!/usr/bin/ruby -w
# define a class
class Box
   BOX_COMPANY = “TATA Inc”
   BOXWEIGHT = 10
   # constructor method
   def initialize(w,h)
      @width, @height = w, h
   end
   # instance method
   def getArea
      @width * @height
   end
end

# create an object
box = Box.new(10, 20)
```

```
# call instance methods
a = box.getArea()
puts "Area of the box is : #{a}"
puts Box::BOX_COMPANY
puts "Box weight is: #{Box::BOXWEIGHT}"
```

When the above code is executed, it produces the following result –

```
Area of the box is : 200
TATA Inc
Box weight is: 10
```

Class constants are inherited and can be overridden like instance methods.

6.2.5 Create Object Using Allocate

There may be a situation when you want to create an object without calling its constructor initialize i.e. using new method, in such case you can call *allocate*, which will create an uninitialized object for you as in the following example –

```
#!/usr/bin/ruby -w

# define a class
class Box
   attr_accessor :width, :height

   # constructor method
   def initialize(w,h)
      @width, @height = w, h
   end

   # instance method
   def getArea
      @width * @height
   end
end
```

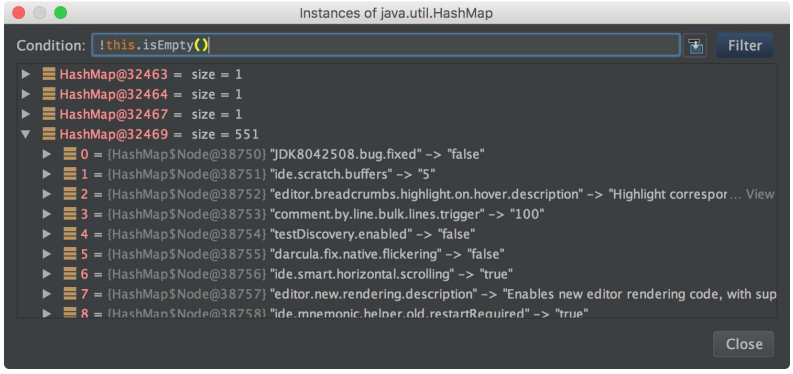
Remember

Classes are defined using the class keyword followed by the end keyword and must be given a name by which they can be referenced. This name is a constant so must begin with a capital letter.

```
# create an object using new
box1 = Box.new(10, 20)

# create another object using allocate
box2 = Box.allocate

# call instance method using box1
a = box1.getArea()
puts “Area of the box is : #{a}”

# call instance method using box2
a = box2.getArea()
puts “Area of the box is : #{a}”
```

When the above code is executed, it produces the following result –

```
Area of the box is : 200
test.rb:14: warning: instance variable @width not initialized
test.rb:14: warning: instance variable @height not initialized
test.rb:14:in `getArea’: undefined method `*’
   for nil:NilClass (NoMethodError) from test.rb:29
```



6.2.6 Class Information

If class definitions are executable code, this implies that they execute in the context of some object: self must reference something. Let’s find out what it is.

```
#!/usr/bin/ruby -w

class Box
   # print class information
   puts "Type of self = #{self.type}"
   puts "Name of self = #{self.name}"
end
```

When the above code is executed, it produces the following result –

```
Type of self = Class
Name of self = Box
```

This means that a class definition is executed with that class as the current object. This means that methods in the met class and its super classes will be available during the execution of the method definition.

SUMMARY

- Ruby is an object-oriented programming language (OOP) that uses classes as blueprints for objects.
- Ruby classes are the blueprints that establish what attributes (also known as states) and behaviors (known in Ruby as methods) that an object should have.
- A class definition starts with the keyword class followed by the class name and is delimited with an end.
- A class provides the blueprints for objects, so basically an object is created from a class.
- The initialize method is useful when you want to initialize some class variables at the time of object creation.
- The instance variables are kind of class attributes and they become properties of objects once objects are created using the class.
- The class variables is a variable, which is shared between all instances of a class.
- Ruby gives you three levels of protection at instance methods level, which may be public, private, or protected.
- Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application.
- A mixin is like a specialized implementation of multiple inheritance in which only the interface portion is inherited.

KNOWLEDGE CHECK

1. **In a class, member variables are often called its _____, and its member functions are sometimes referred to as its behaviour, or _____.**
   a. attributes, methods
   b. none of these
   c. values, morals
   d. data, activities
   e. attributes, activities
2. **Which of these keywords are access specifiers?**
   a. near and far
   b. opened and closed
   c. table and row
   d. none of these
   e. private and public
3. **True/False: An Object can be declared prior to the class definition.**
   a. True
   b. False
4. **Use of _____ protects data from inadvertent modifications.**
   a. protect() member function
   b. private access specifier
   c. class protection operator, @
   d. none of these
   e. public access specifier
5. **You can redefine the way _____ work when used with objects.**
   a. none of these
   b. white space characters
   c. standard operators
   d. pre-processor directives
   e. undefined variables
6. **Which of the following is supported by Ruby?**
   a. Multiple Programming Paradigms
   b. Dynamic Type System

c. Automatic Memory Management

d. All of the above

7. What is the extension used for saving the ruby file?

a. .ruby extension

b. .rb extension

c. .rrb extension

d. None of the mentioned

REVIEW QUESTIONS

1. Define Ruby Objects.
2. Focus on class methods and variables in Ruby.
3. Ruby does not apply any access control over instance and class variables, Explain.
4. Discuss about freezing objects.
5. How can you say that initialize method is a standard Ruby class method? Explain.

Check Your Result

1. (a) 2. (e) 3. (b) 4. (b) 5. (c)
6. (d) 7. (b)

REFERENCES

1. Bista, R., Bajracharya, L., & Dongol, D. (2015). A New Approach To Enhance Efficiency of Object Oriented Programming. Technia, 8(1), 1058.
2. Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B. J., Conallen, J., & Houston, K. A. (2007). Object-Oriented Analysis and Design with Applications (3rd ed.). Boston, MA: AddisonWesley Professional.
3. Booch, G., Rumbaugh, J., & Jacobson, I. (2005, May). The Unified Modeling Language User Guide. Addison-Wesley.
4. D. Beckett. The design and implementation of the Redland RDF application framework. Computer Networks, 39(5):577–588, 2002.
5. David Barnes and Michael Kölling. Objects First with Java: A practical introduction using BlueJ (Prentice Hall, 2004)
6. Deborah J. Armstrong. 2006. The quarks of object-oriented development. Commun. ACM 49, 2 (February 2006), 123-128. DOI: http://doi.acm.org/10.1145/1113034.1113040
7. Deitel, P., & Deitel, H. (2012). Java How to Program (9th ed.). Pearson Education Limited.
8. Dennis, A., Wixom, B. H., & Tegarden, D. (2015, April). System Analysis & Design: An Object-Oriented Approach with UML (5th ed.). Academic Pres.
9. Erdebilli (B.D.Rouyendegh) B. (2011), Selecting the high - performing departments within universities applying the fuzzy MADM methods, Scientific Research and Essays (SRE) , 6, 2646-2654
10. Fong-Gong Wu, F.G., & Ying-Jye Lee, Y.J., & Ming-Chyuan Lin, C.M. (2004), Using the Fuzzy Analytic Hierarchy Process on Optimum Spatial Allocation, International Journal of Industrial Ergonomics, 33, 553-569.
11. Harel, D., Marron, A., & Weiss, G. (2010, June). Programming coordinated behavior in java. In European Conference on Object-Oriented Programming (pp. 250-274). Springer Berlin Heidelberg.
12. Jens Bennedsen and Michael E. Caspersen. 2004. Programming in Context – A Model-First Approach to CS1, In Proceedings of the 35th SIGCSE technical symposium on Computer science education (SIGCSE '04). ACM, New York, NY, USA, 477-481. DOI: http://doi.acm.org/10.1145/971300.971461, 477-481.
13. Sandy Garner, Patricia Haden, and Anthony Robins. 2005. My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In Proceedings of the 7th Australasian conference on Computing education - Volume 42 (ACE '05), Alison Young and Denise Tolhurst (Eds.), Vol. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 173-180.
14. T. Berners-Lee. Weaving the Web – The Past, Present and Future of the World Wide Web by its Inventor. Texere, 2000.

CHAPTER

7 DEBUGGER

"Debugging itu adalah sebuah metode yang dilakukan oleh para programmer untuk mencari jarum di tumpukan jerami"

–Harly Umboh

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Explain the Ruby debugger
2. Understand the logger
3. Discuss how to debug with the bye bug gem

INTRODUCTION

A debugger or debugging tool is a computer program used to test and debug other programs (the "target" program). The main use of a debugger is to run the target program

under controlled conditions that permit the programmer to track its operations in progress and monitor changes in computer resources (most often memory areas used by the target program or the computer's operating system) that may indicate malfunctioning code. Typical debugging facilities include the ability to run or halt the target program at specific points, display the contents of memory, CPU registers or storage devices (such as disk drives), and modify memory or register contents in order to enter selected test data that might be a cause of faulty program execution.

The code to be examined might alternatively be running on an instruction set simulator (ISS), a technique that allows great power in its ability to halt when specific conditions are encountered, but which will typically be somewhat slower than executing the code directly on the appropriate (or the same) processor. Some debuggers offer two modes of operation, full or partial simulation, to limit this impact.

Typically, debuggers offer a query processor, a symbol resolver, an expression interpreter, and a debug support interface at its top level. Debuggers also offer more sophisticated functions such as running a program step by step (single-stepping or program animation), stopping (breaking) (pausing the program to examine the current state) at some event or specified instruction by means of a breakpoint, and tracking the values of variables. Some debuggers have the ability to modify program state while it is running. It may also be possible to continue execution at a different location in the program to bypass a crash or logical error.

The same functionality which makes a debugger useful for correcting bugs allows it to be used as a software cracking tool to evade copy protection, digital rights management, and other software protection features. It often also makes it useful as a general verification tool, fault coverage, and performance analyzer, especially if instruction path lengths are shown. Early microcomputers with disk-based storage often benefitted from the ability to diagnose and recover corrupted directory or registry data records, to "undelete" files marked as deleted, or to crack file password protection.

Most mainstream debugging engines, such as gdb and dbx, provide console-based command line interfaces. Debugger front-ends are popular extensions to debugger engines that provide IDE integration, program animation, and visualization features.

Typically, when you compile a program and there's an error, it does not give any meaningful explanation about the error. In such cases, it can be hard to locate and resolve the problem.

- Syntax error
- Logical error

A syntax error does not let you compile the code until you fix the syntax. With a logical error, the code compiles without any issues, but when you execute the program, it gives unexpected results.

A debugger helps us locate and fix logical errors efficiently that, in some cases, would be a nightmare to fix without a debugger.

7.1 RUBY – DEBUGGER

It doesn't matter how easy a language is to use, it usually contains some bugs if it is more than a few lines long. To help deal with bugs, the standard distribution of Ruby includes a debugger.

In order to start the Ruby debugger, load the debug library using the command-line option -r debug. The debugger stops before the first line of executable code and asks for the input of user commands.

Keyword

Command is a directive to a computer program acting as an interpreter of some kind to perform a specific task.

7.1.1 Usage Syntax

Here is the usage syntax to use ruby debugger –

$ ruby -r debug filename [, ...]

7.1.2 Ruby Debugger Commands

Here is a complete list of **commands**, which you can use while debugging your program. Here, it is not necessary to use complete keyword to give a command, part given inside [...] is option.

Sr.No.	Command & Description
1	**b[reak] [< file| class>:]< line| method>** Sets breakpoint to some position. Breakpoint is a place where you want to pause program execution for debugging purpose.
2	**wat[ch] expression** Sets watchpoints.
3	**cat[ch] (exception|off)** Sets catchpoint to an exception.
4	**b[reak]** Displays breakpoints and watchpoints.
5	**del[ete] [n]** Deletes breakpoints.
6	**disp[lay]** ***expression*** Displays value of *expression*.

7	**undisp[lay] [n]** Removes display of n
8	c[ont] Continues execution.
9	**s[tep] [n]** Executes next n lines stepping into methods.
10	**n[ext] [n]** Executes next n lines stepping over methods.
11	**w[here]** Displays stack frame
12	**f[rame]** Synonym for where.
13	**l[ist][<-| n- m>]** Displays source lines from n to m.
14	**up [n]** Moves up n levels in the stack frame.
15	**down [n]** Moves down n levels in the stack frame.
16	**fin[ish]** Finishes execution of the current method.
17	**tr[ace] [on|off]** Toggles trace mode on and off.
18	**q[uit]** Exits debugger.
19	**v[ar] g[lobal]** Displays global variables.
20	**v[ar] l[ocal]** Displays local variables.
21	**v[ar] i[instance]** ***object*** Displays instance variables of *object*
22	**v[ar] c[onst]** ***object*** Displays constants of *object*.
23	**m[ethod] i[instance]** ***object*** Displays instance methods of *object*.

24	**m[ethod]** ***class*** **|** ***module*** Displays instance methods of the *class or module*.
25	**th[read] l[ist]** Displays threads.
26	**th[read] c[ur[rent]]** Displays current thread.
27	**th[read] n** Stops specified thread.
28	th[read] stop > Synonym for th[read] n.
29	**th[read] c[ur[rent]] n>** Synonym for th[read] n
30	**th[read] resume >** Resumes thread n
31	**p** ***expression*** Evaluates the *expression*
32	**h[elp]** Displays help message
33	**everything else** Evaluates.

Example

Consider the following file *hello.rb*, which needs to be debugged −

```
#!/usr/bin/env ruby
class Hello
   def initialize( hello )
      @hello = hello
   end
   def hello
      @hello
   end
end
salute = Hello.new( “Hello, Mac!” )
puts salute.hello
```

Here is one interactive session captured. Given commands are written in bold −

```
[root@ruby]# ruby -r debug hello.rb
Debug.rb
Emacs support available.

hello.rb:3:class Hello
(rdb:1) v l
   salute => nil
(rdb:1) b 10
Set breakpoint 1 at hello.rb:10
(rdb:1) c
Hello, Mac!
[root@ruby]#
```

7.2 THE LOGGER

It can also be useful to save information to log files at runtime. Rails maintains a separate log file for each runtime environment.

7.2.1 What is the Logger?

Rails makes use of the ActiveSupport::Logger class to write log information. Other loggers, such as Log4r, may also be substituted.

You can specify an alternative logger in config/application.rb or any other environment file, for example:

config.logger = Logger.new(STDOUT)

config.logger = Log4r::Logger.new("Application Log")

Or in the Initializer section, add any of the following

Rails.logger = Logger.new(STDOUT)

Rails.logger = Log4r::Logger.new("Application Log")

By default, each log is created under Rails.root/log/ and the log file is named after the environment in which the application is running.

If your program makes use of a framework like Rails or Sinatra your stacktrace might be quite convoluted. It might be useful to filter the stack trace by your project name.

puts caller.select { |line| line.include? 'project_name' }

or

puts caller.select { |line| line['project_name'] }

7.2.2 Log Levels

When something is logged, it's printed into the corresponding log if the log level of the message is equal to or higher than the configured log level. If you want to know the current log level, you can call the Rails.logger.level method.

The available log levels are: :debug, :info, :warn, :error, :fatal, and :unknown, corresponding to the log level numbers from 0 up to 5, respectively. To change the default log level, use

config.log_level = :warn # In any environment initializer, or

Rails.logger.level = 0 # at any time

This is useful when you want to log under development or staging without flooding your production log with unnecessary information.

The default Rails log level is debug in all environments.

Controller is a chip, an expansion card, or a stand-alone device that interfaces with a peripheral device.

7.2.3 Sending Messages

To write in the current log use the logger.(debug|info|warn|error|fatal) method from within a **controller,** model or mailer:

logger.debug "Person attributes hash: #{@person.attributes.inspect}"

logger.info "Processing the request..."

logger.fatal "Terminating application, raised unrecoverable error!!!"

Here's an example of a method instrumented with extra logging:

```
class ArticlesController < ApplicationController
  # ...
  def create
```

```
    @article = Article.new(article_params)
    logger.debug "New article: #{@article.attributes.inspect}"
    logger.debug "Article should be valid: #{@article.valid?}"

    if @article.save
    logger.debug "The article was saved and now the user is going to be redirected..."
      redirect_to @article, notice: 'Article was successfully created.'
    else
      render :new
    end
  end

  # ...

  private
    def article_params
      params.require(:article).permit(:title, :body, :published)
    end
end
```

Here's an example of the log generated when this controller action is executed:

Started POST "/articles" for 127.0.0.1 at 2017-08-20 20:53:10 +0900

Processing by ArticlesController#create as HTML

Parameters: {"utf8"=>"✓", "authenticity_token"=>"xhuIbSBFytHCE1agHgvrlKnSVIOGD6jltW2tO+P6a/ACjQ3igjpV4OdbsZjIhC98QizWH9YdKokrqxBCJrtoqQ==", "article"=>{"title"=>"Debugging Rails", "body"=>"I'm learning how to print in logs!!!", "published"=>"0"}, "commit"=>"Create Article"}

New article: {"id"=>nil, "title"=>"Debugging Rails", "body"=>"I'm learning how to print in logs!!!", "published"=>false, "created_at"=>nil, "updated_at"=>nil}

Article should be valid: true

(0.1ms) BEGIN

SQL (0.4ms) INSERT INTO "articles" ("title", "body", "published", "created_at", "updated_at") VALUES ($1, $2, $3, $4, $5) RETURNING "id" [["title", "Debugging Rails"], ["body", "I'm learning how to print in logs!!!"], ["published", "f"], ["created_at", "2017-08-20 11:53:10.010435"], ["updated_at", "2017-08-20 11:53:10.010435"]]

(0.3ms) COMMIT

The article was saved and now the user is going to be redirected...

Redirected to http://localhost:3000/articles/1

Completed 302 Found in 4ms (ActiveRecord: 0.8ms)

Adding extra logging like this makes it easy to search for unexpected or unusual behavior in your logs. If you add extra logging, be sure to make sensible use of log levels to avoid filling your production logs with useless trivia.

7.2.4 Tagged Logging

When running multi-user, multi-account applications, it's often useful to be able to filter the logs using some custom rules. TaggedLogging in Active Support helps you do exactly that by stamping log lines with subdomains, request ids, and anything else to aid debugging such applications.

```
logger = ActiveSupport::TaggedLogging.new(Logger.new(STDOUT))
logger.tagged("BCX") { logger.info "Stuff" }                    # Logs "[BCX] Stuff"
logger.tagged("BCX", "Jason") { logger.info "Stuff" }           # Logs "[BCX] [Jason] Stuff"
logger.tagged("BCX") { logger.tagged("Jason") { logger.info "Stuff" } } # Logs "[BCX] [Jason] Stuff"
```

7.2.5 Impact of Logs on Performance

Logging will always have a small impact on the performance of your Rails app, particularly when logging to disk. Additionally, there are a few subtleties:

Using the: debug level will have a greater performance penalty than: fatal, as a far greater number of strings are being evaluated and written to the log output (e.g. disk).

Another potential pitfall is too many calls to Logger in your code:

```
logger.debug "Person attributes hash: #{@person.attributes.inspect}"
```

In the example, there will be a performance impact even if the allowed output level doesn't include debug. The reason is that Ruby has to evaluate these strings, which includes instantiating the somewhat heavy String object and interpolating the variables. Therefore, it's recommended to pass blocks to the logger methods, as these are only evaluated if the output level is the same as — or included in — the allowed level (i.e. lazy loading). The same code rewritten would be:

```
logger.debug {"Person attributes hash: #{@person.attributes.inspect}"}
```

The contents of the block, and therefore the string interpolation, are only evaluated if debug is enabled. This performance savings are only really noticeable with large amounts of logging, but it's a good practice to employ.

7.3 DEBUGGING WITH THE BYE BUG GEM

Keyword

Console is the text entry and display device for system administration messages, particularly those from the BIOS or boot loader, the kernel, from the init system and from the system logger.

When your code is behaving in unexpected ways, you can try printing to logs or the **console** to diagnose the problem. Unfortunately, there are times when this sort of error tracking is not effective in finding the root cause of a problem. When you actually need to journey into your running source code, .the debugger is your best companion

7.3.1 Setup

You can use the byebug gem to set breakpoints and step through live code in Rails. To install it, just run:

```
$ gem install byebug
```

Inside any Rails application you can then invoke the debugger by calling the byebug method.

Here's an example:

```
class PeopleController < ApplicationController
  def new
    byebug
    @person = Person.new
  end
end
```

7.3.2 The Shell

As soon as your application calls the byebug method, the debugger will be started in a debugger shell inside the terminal window where you launched your application server, and you will be placed at the debugger's prompt (byebug). Before the prompt, the code around the line that is about to be run will be displayed and the current line will be marked by '=>', like this:

```
[1, 10] in /PathTo/project/app/controllers/articles_controller.
rb
     3:
     4:   # GET /articles
```

```
    5:   # GET /articles.json
    6:   def index
    7:     byebug
=>  8:     @articles = Article.find_recent
    9:
   10:     respond_to do |format|
   11:       format.html # index.html.erb
   12:       format.json { render json: @articles }
(byebug)
```

If you got there by a browser request, the browser tab containing the request will be hung until the debugger has finished and the trace has finished processing the entire request.

For example

```
=> Booting Puma
    => Rails 5.1.0 application starting in development on http://0.0.0.0:3000
    => Run `rails server -h` for more startup options
    Puma starting in single mode...
    * Version 3.4.0 (ruby 2.3.1-p112), codename: Owl Bowl Brawl
    * Min threads: 5, max threads: 5
    * Environment: development
    * Listening on tcp://localhost:3000
    Use Ctrl-C to stop
    Started GET "/" for 127.0.0.1 at 2014-04-11 13:11:48 +0200
      ActiveRecord::SchemaMigration Load (0.2ms)  SELECT "schema_migrations".* FROM "schema_migrations"
    Processing by ArticlesController#index as HTML
    [3, 12] in /PathTo/project/app/controllers/articles_controller.rb
        3:
        4:   # GET /articles
        5:   # GET /articles.json
```

Keyword

HTML is the standard markup language for creating web pages and web applications.

```
    6:   def index
    7:     byebug
=>  8:     @articles = Article.find_recent
    9:
   10:     respond_to do |format|
   11:       format.html # index.html.erb
   12:       format.json { render json: @articles }
(byebug)
```

Now it's time to explore your application. A good place to start is by asking the debugger for help. Type: help

```
(byebug) help
  break      -- Sets breakpoints in the source code
  catch      -- Handles exception catchpoints
  condition  -- Sets conditions on breakpoints
  continue   -- Runs until program ends, hits a breakpoint or reaches a line
  debug      -- Spawns a subdebugger
  delete     -- Deletes breakpoints
  disable    -- Disables breakpoints or displays
  display    -- Evaluates expressions every time the debugger stops
  down       -- Moves to a lower frame in the stack trace
  edit       -- Edits source files
  enable     -- Enables breakpoints or displays
  finish     -- Runs the program until frame returns
  frame      -- Moves to a frame in the call stack
  help       -- Helps you using byebug
  history    -- Shows byebug's history of commands
  info       -- Shows several informations about the program being debugged
  interrupt -- Interrupts the program
  irb        -- Starts an IRB session
  kill       -- Sends a signal to the current process
  list       -- Lists lines of source code
  method     -- Shows methods of an object, class or module
  next       -- Runs one or more lines of code
```

```
pry         -- Starts a Pry session
quit        -- Exits byebug
restart     -- Restarts the debugged program
save        -- Saves current byebug session to a file
set         -- Modifies byebug settings
show        -- Shows byebug settings
source      -- Restores a previously saved byebug session
step        -- Steps into blocks or methods one or more times
thread      -- Commands to manipulate threads
tracevar    -- Enables tracing of a global variable
undisplay   -- Stops displaying all or some expressions when program stops
untracevar -- Stops tracing a global variable
up          -- Moves to a higher frame in the stack trace
var         -- Shows variables and its values
where       -- Displays the backtrace
```

(byebug)

To see the previous ten lines you should type list- (or l-).

```
(byebug) l-
[1, 10] in /PathTo/project/app/controllers/articles_controller.rb
   1  class ArticlesController < ApplicationController
   2    before_action :set_article, only: [:show, :edit, :update, :destroy]
   3
   4    # GET /articles
   5    # GET /articles.json
   6    def index
   7      byebug
   8      @articles = Article.find_recent
   9
   10     respond_to do |format|
```

This way you can move inside the file and see the code above the line where you added the byebug call. Finally, to see where you are in the code again you can type list==byebug) list)

```
in /PathTo/project/app/controllers/articles_controller.rb [12 ,3]
     3:
     4:   # GET /articles
     5:   # GET /articles.json
     6:   def index
     7:     byebug
=>   8:     @articles = Article.find_recent
     9:
    10:     respond_to do |format|
    11:       format.html # index.html.erb
    12:       format.json { render json: @articles }
(byebug)
```

7.3.3 The Context

When you start debugging your application, you will be placed in different contexts as you go through the different parts of the stack.

The debugger creates a context when a stopping point or an event is reached. The context has information about the suspended program which enables the debugger to inspect the frame stack, evaluate variables from the perspective of the debugged program, and know the place where the debugged program is stopped

At any time you can call the backtrace command (or its alias where) to print the backtrace of the application. This can be very helpful to know how you got where you are. If you ever wondered about how you got somewhere in your code, then backtrace will supply the answer.

```
(byebug) where
--> #0  ArticlesController.index
        at /PathToProject/app/controllers/articles_controller.rb:8
    #1  ActionController::BasicImplicitRender.send_action(method#String, *args#Array)
        at /PathToGems/actionpack-5.1.0/lib/action_controller/metal/basic_implicit_
render.rb:4
    #2  AbstractController::Base.process_action(action#NilClass, *args#Array)
        at /PathToGems/actionpack-5.1.0/lib/abstract_controller/base.rb:181
    #3  ActionController::Rendering.process_action(action, *args)
        at /PathToGems/actionpack-5.1.0/lib/action_controller/metal/rendering.rb:30
```

...

The current frame is marked with -->. You can move anywhere you want in this trace (thus changing the context) by using the frame n command, where n is the specified frame number. If you do that, byebug will display your new context.

```
(byebug) frame 2
[176, 185] in /PathToGems/actionpack-5.1.0/lib/abstract_controller/base.rb
   176:       # is the intended way to override action dispatching.
   177:       #
   178:       # Notice that the first argument is the method to be dispatched
   179:       # which is *not* necessarily the same as the action name.
   180:       def process_action(method_name, *args)
=> 181:         send_action(method_name, *args)
   182:       end
   183:
   184:       # Actually call the method associated with the action. Override
   185:       # this method if you wish to change how action methods are called,
(byebug)
```

The available variables are the same as if you were running the code line by line. After all, that's what debugging is.

You can also use up [n] and down [n] commands in order to change the context n frames up or down the stack respectively. n defaults to one. Up in this case is towards higher-numbered stack frames, and down is towards lower-numbered stack frames.

7.3.4 Threads

The debugger can list, stop, resume and switch between running threads by using the thread command (or the abbreviated th). This command has a handful of options:

thread: shows the current thread.

thread list: is used to list all threads and their statuses. The current thread is marked with a plus (+) sign.

thread stop n: stops thread n.

thread resume n: resumes thread n.

thread switch n: switches the current thread context to n.

This command is very helpful when you are debugging concurrent threads and need to verify that there are no race conditions in your code.

7.3.5 Inspecting Variables

Any expression can be evaluated in the current context. To evaluate an expression, just type it!

This example shows how you can print the instance variables defined within the current context:

```
[3, 12] in /PathTo/project/app/controllers/articles_controller.rb
    3:
    4:   # GET /articles
    5:   # GET /articles.json
    6:   def index
    7:     byebug
=>  8:     @articles = Article.find_recent
    9:
   10:     respond_to do |format|
   11:       format.html # index.html.erb
   12:       format.json { render json: @articles
(byebug) instance_variables
[:@_action_has_layout, :@_routes, :@_request, :@_response, :@_lookup_context,
 :@_action_name, :@_response_body, :@marked_for_same_origin_verification,
 :@_config]
```

As you may have figured out, all of the variables that you can access from a controller are displayed. This list is dynamically updated as you execute code. For example, run the next line using next.

```
(byebug) next
[5, 14] in /PathTo/project/app/controllers/articles_controller.rb
   5     # GET /articles.json
   6     def index
   7       byebug
   8       @articles = Article.find_recent
   9
=> 10      respond_to do |format|
   11       format.html # index.html.erb
```

```
12          format.json { render json: @articles }
13        end
14      end
15
```

(byebug)

And then ask again for the instance_variables:

(byebug) instance_variables

[:@_action_has_layout, :@_routes, :@_request, :@_response, :@_lookup_context,
:@_action_name, :@_response_body, :@marked_for_same_origin_verification,
:@_config, :@articles]

Now @articles is included in the instance variables, because the line defining it was executed.

You can also step into irb mode with the command irb (of course!). This will start an irb session within the context you invoked it.

The var method is the most convenient way to show variables and their values. Let's have byebug help us with it.

(byebug) help var

```
[v]ar <subcommand>
Shows variables and its values
var all      -- Shows local, global and instance variables of self.
var args     -- Information about arguments of the current scope
var const    -- Shows constants of an object.
var global   -- Shows global variables.
var instance -- Shows instance variables of self or a specific object.
var local    -- Shows local variables in current scope.
```

This is a great way to inspect the values of the current context variables. For example, to check that we have no local variables currently defined:

(byebug) var local

(byebug)

You can also inspect for an object method this way:

(byebug) var instance Article.new

@_start_transaction_state = {}

@aggregation_cache = {}

@association_cache = {}

@attributes = #<ActiveRecord::AttributeSet:0x007fd0682a9b18 @attributes={"id"=>#<ActiveRecord::Attribute::FromDatabase: 0x007fd0682a9a00 @ name="id", @value_be...

@destroyed = false

@destroyed_by_association = nil

@marked_for_destruction = false

@new_record = true

@readonly = false

@transaction_state = nil

You can also use display to start watching variables. This is a good way of tracking the values of a variable while the execution goes on.

(byebug) display @articles

1: @articles = nil

The variables inside the displayed list will be printed with their values after you move in the stack. To stop displaying a variable use undisplay n where n is the variable number (1 in the last example).

Did You Know?

Prior to the development of alphanumeric CRT system consoles, some computers such as the IBM 1620 had console typewriters and front panels while the very first programmable computer, the Manchester Baby, used a combination of electro-mechanical switches and a CRT to provide console functions—the CRT displaying memory contents in binary by mirroring the machine's Williams-Kilburn tube CRT-based RAM.

7.3.6 Step by Step

Now you should know where you are in the running trace and be able to print the available variables. But let's continue and move on with the application execution.

Use step (abbreviated s) to continue running your program until the next logical stopping point and return control to the debugger. next is similar to step, but while step stops at the next line of code executed, doing just a single step, next moves to the next line without descending inside methods.

For example, consider the following situation:

Started GET "/" for 127.0.0.1 at 2014-04-11 13:39:23 +0200

Processing by ArticlesController#index as HTML

[1, 6] in /PathToProject/app/models/article.rb

```
    1: class Article < ApplicationRecord
    2:   def self.find_recent(limit = 10)
    3:     byebug
=> 4:     where('created_at > ?', 1.week.ago).limit(limit)
```

```
    5:   end
    6: end
(byebug)
```

If we use next, we won't go deep inside method calls. Instead, byebug will go to the next line within the same context. In this case, it is the last line of the current method, so byebug will return to the next line of the caller method.

```
(byebug) next
[4, 13] in /PathToProject/app/controllers/articles_controller.rb
    4:   # GET /articles
    5:   # GET /articles.json
    6:   def index
    7:     @articles = Article.find_recent
    8:
=>  9:     respond_to do |format|
   10:       format.html # index.html.erb
   11:       format.json { render json: @articles }
   12:     end
   13:   end
(byebug)
```

If we use step in the same situation, byebug will literally go to the next Ruby instruction to be executed -- in this case, Active Support's week method.

```
(byebug) step
[49, 58] in /PathToGems/activesupport-5.1.0/lib/active_support/core_ext/numeric/
time.rb
   49:
   50:   # Returns a Duration instance matching the number of weeks provided.
   51:   #
   52:   #   2.weeks # => 14 days
   53:   def weeks
=> 54:     ActiveSupport::Duration.weeks(self)
   55:   end
   56:   alias: week: weeks
   57:
```

58: # Returns a Duration instance matching the number of fortnights provided.

(byebug)

This is one of the best ways to find bugs in your code.

You can also use step n or next n to move forward n steps at once.

Remember

If you don't want to run the program manually in debug mode, you can press the "Resume Button" on the tools panel to run to the next breakpoint, or finish the debug if there are no more breakpoints.

7.3.7 Breakpoints

A breakpoint makes your application stop whenever a certain point in the program is reached. The debugger shell is invoked in that line

You can add breakpoints dynamically with the command break (or just b). There are 3 possible ways of adding breakpoints manually:

break n: set breakpoint in line number n in the current source file.

break file: n [if expression]: set breakpoint in line number n inside file named file. If an expression is given it must have evaluated to true to fire up the debugger.

break class (.|\#)method [if expression]: set breakpoint in method (. and # for class and instance method respectively) defined in class. The expression works the same way as with file: n.

For example, in the previous situation

[4, 13] in /PathToProject/app/controllers/articles_controller.rb

```
    4:   # GET /articles
    5:   # GET /articles.json
    6:   def index
    7:     @articles = Article.find_recent
    8:
=>  9:     respond_to do |format|
   10:       format.html # index.html.erb
   11:       format.json { render json: @articles }
   12:     end
   13:   end
```

(byebug) break 11

Successfully created breakpoint with id 1

Use info breakpoints to list breakpoints. If you supply a number, it lists that breakpoint. Otherwise it lists all breakpoints.

(byebug) info breakpoints

Num Enb What

1 y at /PathToProject/app/controllers/articles_controller.rb:11

To delete breakpoints: use the command delete n to remove the breakpoint number n. If no number is specified, it deletes all breakpoints that are currently active.

(byebug) delete 1

(byebug) info breakpoints

No breakpoints.

You can also enable or disable breakpoints:

enable breakpoints [n [m [...]]]: allows a specific breakpoint list or all breakpoints to stop your program. This is the default state when you create a breakpoint.

disable breakpoints [n [m [...]]]: make certain (or all) breakpoints have no effect on your program.

Remember

Proper use of the debugger is essential to finding semantic (logical) errors in how your program behaves. The debugger should be considered your best friend while programming (that is, unless you can perfectly visualize how your program will run in your head).

7.3.8 Catching Exceptions

The command catch exception-name (or just cat exception-name) can be used to intercept an exception of type exception-name when there would otherwise be no handler for it.

To list all active catch points use catch.

7.3.9 Resuming Execution

There are two ways to resume execution of an application that is stopped in the debugger:

continue [n]: resumes program execution at the address where your script last stopped; any breakpoints set at that address are bypassed. The optional argument n allows you to specify a line number to set a one-time breakpoint which is deleted when that breakpoint is reached.

finish [n]: execute until the selected stack frame returns. If no frame number is given, the application will run until the currently selected frame returns. The currently selected frame starts out the most-recent frame or 0 if no frame positioning (e.g up, down or frame) has been performed. If a frame number is given it will run until the specified frame returns.

7.3.10 Editing

Two commands allow you to open code from the debugger into an editor:

edit [file:n]: edit file named file using the editor specified by the EDITOR environment variable. A specific line n can also be given.

7.3.11 Quitting

To exit the debugger, use the quit command (abbreviated to q). Or, type q! to bypass the Really quit? (y/n) prompt and exit unconditionally.

A simple quit tries to terminate all threads in effect. Therefore your server will be stopped and you will have to start it again.

7.3.12 Settings

byebug has a few available options to tweak its behavior:

```
(byebug) help se
  set <setting> <value>
  Modifies byebug settings
  Boolean values take “on”, “off”, “true”, “false”, “1” or “0”. If you
  don’t specify a value, the boolean setting will be enabled. Conversely,
  you can use “set no<setting>” to disable them.
  You can see these environment settings with the “show” command.
  List of supported settings:
  autosave      -- Automatically save command history record on exit
  autolist      -- Invoke list command on every stop
  width         -- Number of characters per line in byebug’s output
  autoirb       -- Invoke IRB on every stop
  basename      -- <file>:<line> information after every stop uses short paths
  linetrace     -- Enable line execution tracing
  autopry       -- Invoke Pry on every stop
```

stack_on_error -- Display stack trace when `eval` raises an exception

fullpath -- Display full file names in backtraces

histfile -- File where cmd history is saved to. Default: ./.byebug_history

listsize -- Set number of source lines to list by default

post_mortem -- Enable/disable post-mortem mode

callstyle -- Set how you want method call parameters to be displayed

histsize -- Maximum number of commands that can be stored in byebug history

savefile -- File where settings are saved to. Default: ~/.byebug_save

You can save these settings in an .byebugrc file in your home directory. The debugger reads these global settings when it starts. For example:

```
set callstyle short
set listsize 25
```

SUMMARY

- A debugger or debugging tool is a computer program used to test and debug other programs (the "target" program).
- The main use of a debugger is to run the target program under controlled conditions that permit the programmer to track its operations in progress and monitor changes in computer resources (most often memory areas used by the target program or the computer's operating system) that may indicate malfunctioning code.
- Typical debugging facilities include the ability to run or halt the target program at specific points, display the contents of memory, CPU registers or storage devices (such as disk drives), and modify memory or register contents in order to enter selected test data that might be a cause of faulty program execution.
- A debugger helps us locate and fix logical errors efficiently that, in some cases, would be a nightmare to fix without a debugger.
- TaggedLogging in Active Support helps you do exactly that by stamping log lines with subdomains, request ids, and anything else to aid debugging such applications.
- The debugger creates a context when a stopping point or an event is reached.
- The context has information about the suspended program which enables the debugger to inspect the frame stack, evaluate variables from the perspective of the debugged program, and know the place where the debugged program is stopped
- The optional argument n allows you to specify a line number to set a one-time breakpoint which is deleted when that breakpoint is reached.
- To stop displaying a variable use undisplay n where n is the variable number (1 in the last example).

KNOWLEDGE CHECK

1. **Which of the following is supported by Ruby?**
   a. Multiple Programming Paradigms
   b. Dynamic Type System
   c. Automatic Memory Management
   d. All of the Mentioned
2. **Which of the following features does the 2.0 version of ruby supports?**
   a. Method keyword arguments
   b. New literals
   c. Security fixes
   d. All of the mentioned
3. **Which of the following languages syntax matches with the Ruby's syntax?**
   a. Perl
   b. PHP
   c. Java
   d. Jquery
4. **What is the extension used for saving the ruby file?**
   a. .ruby extension
   b. .rb extension
   c. .rrb extension
   d. None of the mentioned
5. **Which of the following are valid floating point literal?**
   a. .5
   b. 2
   c. 0.5
   d. None of the mentioned
6. **A step by step instruction used to solve a problem is known as ……..?**
   a. Sequential structure
   b. A List
   c. An Algorithm
   d. A plan

7. **Some incorrect word sequence in a program would generate**
   a. Semantics error
   b. Syntax error
   c. Runtime error
   d. Logical error

REVIEW QUESTIONS

1. Start the server without --debug, then call debugger in the code, and observe the output.
2. Start the server with --debug and add a breakpoint to a controller method. Trigger that breakpoint and experiment with each of these commands:
   - eval
   - list
   - next
   - step
   - continue
3. Debugger is just a method. Try combining it with a conditional branch to only execute on a certain pathway through your code (like a nil input, for example).
4. What you're looking for is a debugger. With ruby 1.8, the ruby-debug gem provides the canonical one, and in ruby 1.9, the debugger gem provides a version of its successor that is well-maintained.
5. Start your program with the gem loaded, for example ruby -rdebugger yourfile.rb, and you'll have access to the debugger.

Check Your Result

1. (d) 2. (d) 3. (a) 4. (b) 5. (c)
6. (c) 7. (b)

REFERENCES

1. Alfred V.Aho,Monica S.Lam, Ravi Sethi and Jeffrey D.Ullman, Compilers (Principles, Techniques and Tools), Second Edition,2006.
2. Aspect oriented programming (article series). Commun. ACM, 44(10), Oct. 2001.
3. Dave Thomas, Chad Fowler and Andy Hunt, Programming Ruby The Pragmatic Programmer's Guide, Second Edition, 2004.
4. Dave Thomas, David Hansson, Leon Breedt, Mike Clark, James Duncan Davidson, Justin Gehtland and Andreas Schwarz, Agile Web Development With Rails, 2nd Edition, 2006.
5. G. H. Cooper and S. Krishnamurthi. FrTime: Functional reactive programming in PLT Scheme. Technical Report cs03-20, Brown University, 2003.
6. K. Anderson, T. J. Hickey, and P. Norvig. Silk: A playful combination of Scheme and Java. In Proceedings of the Workshop on Scheme and Functional Programming, pages 13–22, 2000.
7. KO, Andrew J.; MYERS, Brad A. Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior. In. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Vienna, Austria — April 24 - 29, 2004. New York, NY, USA: ACM Press, 2004, pp. 151–158. ISBN 1-58113-702-8
8. KO, Andrew J.; MYERS, Brad A.; COBLENZ, Michael J.; AUNG, Htet Htet. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks. IEEE Trans. Softw. Eng. 2006, vol. 32, no. 12, pp. 971–987. Available also from WWW: hhttp://dx.doi.org/10.1109/TSE.2006. 116i. ISSN 0098-5589
9. LATOZA, Thomas D.; MYERS, Brad A. Developers Ask Reachability Questions. In. 2010 ACM IEEE 32nd International Conference on Software Engineering. New York, NY, USA: ACM Press, 2010.
10. Linda Dailey Paulson, Developers Shift to dynamic langauges, IEEE Computer Society February 2007.
11. M. Auguston, C. Jeffery, and S. Underwood. A framework for automatic debugging. In Automated Software Engineering, 2002.
12. M. de Sousa Dias and D. J. Richardson. Issues on software monitoring. Technical report, ICS, 2002.
13. R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. Journal of Functional Programming, 12(2):159–182, 2002.

CHAPTER

8

REFLECTION AND METAPROGRAMMING

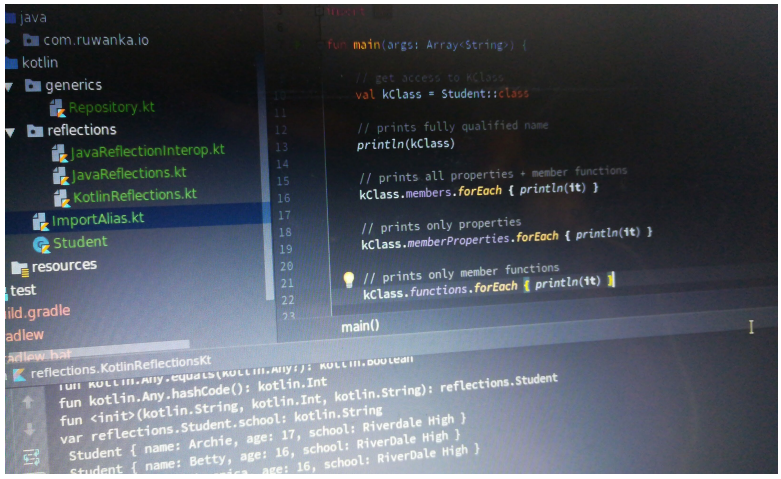
"Most programming languages are decidedly inferior to mathematical notation and are little used as tools of thought in ways that would be considered significant by, say, an applied mathematician. "

–Kenneth E. Iverson

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Discuss the types, classes, and modules
2. Evaluate strings and blocks
3. Define variables and constants
4. Explain the methods for listing, querying, and invoking
5. Describe callback methods or hooks
6. Discuss how to trace the execution of a program
7. Deal with objectspace module
8. Describe custom control structures
9. Focus on missing methods and missing constants
10. Explain dynamically creating methods and alias chaining
11. Define domain-specific languages



INTRODUCTION

Ruby is a very dynamic language; you can insert new methods into classes at runtime, create aliases for existing methods, and even define methods on individual objects.

In addition, it has a rich API for reflection. Reflection, also called introspection, simply means that a program can examine its state and its structure. A Ruby program can, for example, obtain the list of methods defined by the Hash class, query the value of a named instance variable within a specified object, or iterate through all Regexp objects currently defined by the interpreter. The reflection API actually goes further and allows a program to alter its state and structure. A Ruby program can dynamically set named variables, invoke named methods, and even define new classes and new methods.

```ruby
class DunderMifflin
  attr_accessor :name, :department

  def initialize(n, d)
          @name = n
    @department = d
  end

  define_method(:staff_intro) do |*arg|
    puts "#{self.name.capitalize} works in #{self.department} at #
  {self.class}."
  end
end
```

Ruby's reflection API—along with its generally dynamic nature, its blocks-and-iterators control structures, and its parentheses-optional syntax—makes it an ideal language for metaprogramming. Loosely defined, metaprogramming is writing programs (or frameworks) that help you write programs. To put it another way, metaprogramming is a set of techniques for extending Ruby's syntax in ways that make programming easier. Metaprogramming is closely tied to the idea of writing domain specific languages, or DSLs. DSLs in Ruby typically use method invocations and blocks as if they were keywords in a task-specific extension to the language.

We introduce Ruby's reflection API. This API is surprisingly rich and consists of quite a few methods. These methods are defined, for the most part, by Kernel, Object, and Module.

Keep in mind that reflection is not, by itself, metaprogramming. Metaprogramming typically extends the syntax or the behavior of Ruby in some way, and often involves more than one kind of reflection.

8.1 TYPES, CLASSES, AND MODULES

The most commonly used reflective methods are those for determining the type of an object—what class it is an instance of and what methods it responds to.

o.class

Returns the class of an object o.

c.superclass

Returns the superclass of a class c.

o.instance_of? C

Determines whether the object o.class == c.

o.is_a? c

Determines whether o is an instance of c, or of any of its **subclasses**. If c is a module,

this method tests whether o.class (or any of its ancestors) includes the module.

o.kind_of? c

kind_of? is a synonym for is_a?.

c === o

For any class or module c, determines if o.is_a?(c).

o.respond_to? Name

Determines whether the object o has a public or protected method with the specified name. Passes true as the second argument to check private methods as well.

Keyword

Subclasses are classes that can be derived from a parent class by adding some functionality, such as new object variables or new methods.

8.1.1 Ancestry and Modules

In addition to these methods, there are a few related reflective methods for determining the ancestors of a class or module and for determining which modules are included by a class or module. These methods are easy to understand when demonstrated:

```
module A; end                 # Empty module
module B; include A; end;     # Module B includes A
class C; include B; end;      # Class C includes module B
```

```
C < B                          # => true: C includes B
B < A                          # => true: B includes A
C < A                          # => true
Fixnum < Integer               # => true: all fixnums are integers
Integer <Comparable            # => true: integers are comparable
Integer < Fixnum               # => false: not all integers are fixnums
String < Numeric               # => nil: strings are not numbers

A.ancestors                    # => [A]
B.ancestors                    # => [B, A]
C.ancestors                    # => [C, B, A, Object, Kernel]
String.ancestors               # => [String, Enumerable, Comparable, Object, Kernel]
# Note: in Ruby 1.9 String is no longer Enumerable

C.include?(B)                  # => true
C.include?(A)                  # => true
B.include?(A)                  # => true
A.include?(A)                  # => false
A.include?(B)                  # => false


A.included_modules             # => []
B.included_modules             # => [A]
C.included_modules             # => [B, A, Kernel]
```

This code demonstrates include?, which is a public instance method defined by the Module class. But it also features two invocations of the include method (without the question mark), which is a private instance method of Module. As a private method, it can only be invoked implicitly on self, which restricts its usage to the body of a class or module definition. This use of the method include as if it were a keyword is a **metaprogramming** example in Ruby's core syntax.

A method related to the private include method is the public Object.extend. This method extends an object by making the instance methods of each of the specified modules into singleton methods of the object:

```
module Greeter; def hi; "hello"; end; end        # A silly module
s = "string object"
s.extend(Greeter)                                # Add hi as a singleton method to s
s.hi                                             # => "hello"
String.extend(Greeter)                           # Add hi as a class method of String
String.hi                                        # => "hello"
```

Keyword

Metaprogramming is a programming technique in which computer programs have the ability to treat other programs as their data.

The class method Module.nesting is not related to module inclusion or ancestry; instead, it returns an array that specifies the nesting of modules at the current location. Module.nesting[0] is the current class or module, Module.nesting[1] is the containing class or module, and so on:

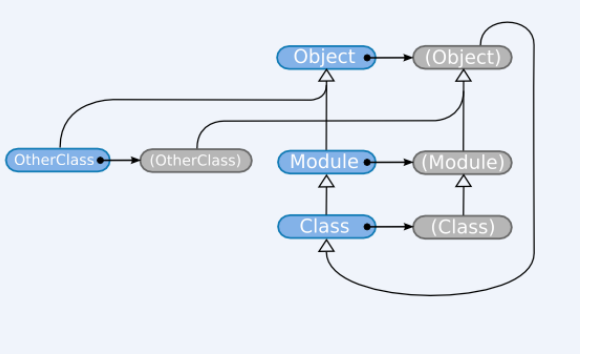
```
module M
 class C
Module.nesting # => [M::C, M]
 end
end
```

8.1.2 Defining Classes and Modules

Classes and modules are instances of the Class and Module classes. As such, you can create them dynamically:

```
M = Module.new                   # Define a new module M
C = Class.new                    # Define a new class C
D = Class.new(C) {               # Define a subclass of C
 include M                       # that includes module M
    }
D.to_s # => "D": class gets constant name by magic
```

One nice feature of Ruby is that when a dynamically created anonymous module or class is assigned to a constant, the name of that constant is used as the name of the module or class (and is returned by its name and to_s methods).



8.2 EVALUATING STRINGS AND BLOCKS

One of the most powerful and straightforward reflective features of Ruby is its eval method. If your Ruby program can generate a string of valid Ruby code, the Kernel.eval method can evaluate that code:

```
x = 1
eval "x + 1" # => 2
```

eval is a very powerful function, but unless you are actually writing a shell program (like irb) that executes lines of Ruby code entered by a user you are unlikely to really need it. (And in a networked context, it is almost never safe to call eval on text received from a user, as it could contain **malicious code**.) Inexperienced programmers sometimes end up using eval as a crutch. If you find yourself using it in your code, see if there is not a way to avoid it. Having said that, there are some more useful ways to use eval and eval-like methods.

Keyword

Malicious code is unwanted files or programs that can cause harm to a computer or compromise data stored on a computer

8.2.1 Bindings and eval

A Binding object represents the state of Ruby's variable bindings at some moment. The Kernel.binding object returns the bindings in effect at the location of the call. You may pass a Binding object as the second argument to eval, and the string you specify will be evaluated in the context of those bindings. If, for example, we define an instance method that returns a Binding object that represents the variable bindings inside an object, then we can use those bindings to query and set the instance variables of that object.

We might accomplish this as follows:

```
class Object                          # Open Object to add a new method
 def bindings                         # Note plural on this method
 binding # This is the predefined Kernel method
 end
end

class Test                  # A simple class with an instance variable
 def initialize(x); @x = x; end
end

t = Test.new(10)                  # Create a test object
eval("@x", t.bindings)            # => 10: We have peeked inside t
```

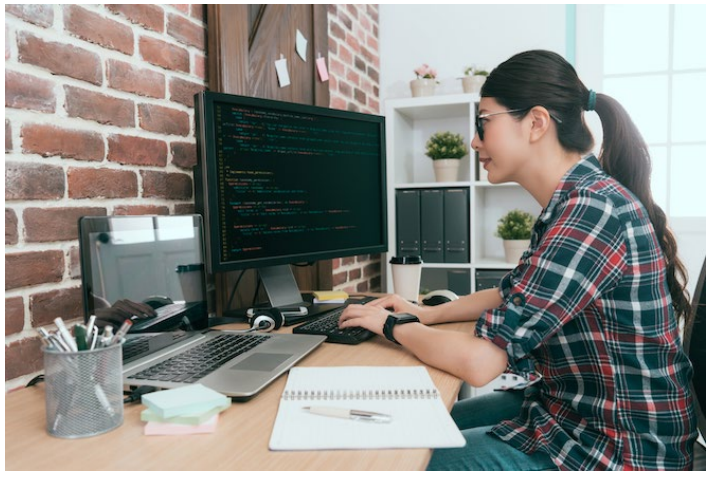
Note that it is not actually necessary to define an Object.bindings method of this sort to peek at the instance variables of an object. Several other methods described shortly offer easier ways to query (and set) the value of the instance variables of an object.

The Proc object defines a public binding method that returns a Binding object representing the variable bindings in effect for the body of that Proc. Furthermore, the eval method allows you to pass a Proc object instead of a Binding object as the second argument.

Ruby 1.9 defines an eval method on Binding objects, so instead of passing a Binding as the second argument to the global eval, you can instead invoke the eval method on a Binding. Which one you choose is purely a stylistic matter; the two techniques are equivalent.

8.2.2 instance_eval and class_eval

The **Object class** defines a method named instance_eval, and the Module class defines a method named class_eval. (module_eval is a synonym for class_eval.) Both of these methods evaluate Ruby code, like eval does, but there are two important differences. The first difference is that they evaluate the code in the context of the specified object or in the context of the specified module—the object or module is the value of self while the code is being evaluated.



Keyword

Object class is the parent class of all the classes in java by default.

Here are some examples:

```
o.instance_eval("@x")    # Return the value of o's instance variable @x

# Define an instance method len of String to return string length
String.class_eval("def len; size; end")


# Here's another way to do that
# The quoted code behaves just as if it was inside "class String" and "end"
String.class_eval("alias len size")


# Use instance_eval to define class method String.empty
# Note that quotes within quotes get a little tricky...
```

String.instance_eval("def empty; ''; end")

Note the subtle but crucial difference between instance_eval and class_eval when the code being evaluated contains a method definition. instance_eval defines singleton methods of the object (and this results in class methods when it is called on a class object). class_eval defines regular instance methods.

The second important difference between these two methods and the global eval is that instance_eval and class_eval can accept a block of code to evaluate. When passed a block instead of a string, the code in the block is executed in the appropriate context. Here, therefore, are alternatives to the previously shown invocations:

```
o.instance_eval { @x }
String.class_eval {
 def len
 size
 end
}
String.class_eval { alias len size }
String.instance_eval { def empty; ""; end }
```

8.2.3 instance_exec and class_exec

Ruby 1.9 defines two more evaluation methods: instance_exec and class_exec (and its alias, module_exec). These methods evaluate a block (but not a string) of code in the context of the receiver object, as instance_eval and class_eval do. The difference is that the exec methods accept arguments and pass them to the block. Thus, the block of code is evaluated in the context of the specified object, with parameters whose values come from outside the object.

8.3 VARIABLES AND CONSTANTS

Kernel, Object, and Module define reflective methods for listing the names (as strings) of all defined global variables, currently defined local variables, all instance variables of an object, all class variables of a class or module, and all constants of a class or module:

```
global_variables                    # => ["$DEBUG", "$SAFE", ...]
x = 1 # Define a local variable
local_variables # => ["x"]
```

```
# Define a simple class
class Point
 def initialize(x,y); @x,@y = x,y; end   # Define instance variables
 @@classvar = 1                          # Define a class variable
 ORIGIN = Point.new(0,0)                 # Define a constant
end
Point::ORIGIN.instance_variables         # => ["@y", "@x"]
Point.class_variables                    # => ["@@classvar"]
Point.constants                          # => ["ORIGIN"]
```

The global_variables, instance_variables, class_variables, and constants methods return arrays of strings in Ruby 1.8 and arrays of symbols in Ruby 1.9. The local_variables method returns an array of strings in both versions of the language.

8.3.1 Querying, Setting, and Testing Variables

In addition to listing defined variables and constants, Ruby Object and Module also define reflective methods for querying, setting, and removing instance variables, class variables, and constants. There are no special purpose methods for querying or setting local variables or global variables, but you can use the eval method for this purpose:

Keyword

eval is a function which evaluates a string as though it were an expression and returns a result; in others, it executes multiple lines of code as though they had been included instead of the line including the eval.

```
x = 1
varname = "x"
eval(varname)                 # => 1
eval("varname = '$g'")        # Set varname to "$g"
eval("#{varname} = x")        # Set $g to 1
eval(varname) # => 1
```

Note that eval evaluates its code in a temporary scope. **eval** can alter the value of instance variables that already exist. But any new instance variables it defines are local to the invocation of eval and cease to exist when it returns. (It is as if the evaluated code is run in the body of a block—variables local to a block do not exist outside the block.)

You can query, set, and test the existence of instance variables on any object and of class variables and constants on any class or module:

```
o = Object.new
o.instance_variable_set(:@x, 0)          # Note required @ prefix
o.instance_variable_get(:@x)             # => 0

o.instance_variable_defined?(:@x)        # => true
Object.class_variable_set(:@@x, 1)       # Private in Ruby 1.8
Object.class_variable_get(:@@x)          # Private in Ruby 1.8
Object.class_variable_defined?(:@@x)     # => true; Ruby 1.9 and later

Math.const_set(:EPI, Math::E*Math::PI)
Math.const_get(:EPI)          # => 8.53973422267357
Math.const_defined? :EPI      # => true
```

In Ruby 1.9, you can pass false as the second argument to const_get and const_defined? to specify that these methods should only look at the current class or module and should not consider inherited constants.

The methods for querying and setting class variables are private in Ruby 1.8. In that version, you can invoke them with class_eval:

```
String.class_eval { class_variable_set(:@@x, 1) }      # Set @@x in String
String.class_eval { class_variable_get(:@@x) }         # => 1
```

Object and Module define private methods for undefining instance variables, class variables, and constants. They all return the value of the removed variable or constant. Because these methods are private, you cannot invoke them directly on an object, class, or module, and you must use an eval method or the send method:

```
o.instance_eval { remove_instance_variable :@x }
String.class_eval { remove_class_variable(:@@x) }
Math.send :remove_const, :EPI # Use send to invoke private method
```

The const_missing method of a module is invoked, if there is one, when a reference is made to an undefined constant. You can define this method to return the value of the named constant. (This feature can be used, for example, to implement an autoload facility in which classes or modules are loaded on demand.) Here is a simpler example:

```
def Symbol.const_missing(name)
 name # Return the constant name as a symbol
end
Symbol::Test # => :Test: undefined constant evaluates to a Symbol
```

Did You Know?

Metaprogramming was popular in the 1970s and 1980s using list processing languages such as LISP. LISP hardware machines were popular in the 1980s and enabled applications that could process code. They were frequently used for artificial intelligence applications.

8.4 METHODS

The Object and Module classes define a number of methods for listing, querying, invoking, and defining methods. We will consider each category in turn.

8.4.1 Listing and Testing

For Methods Object defines methods for listing the names of methods defined on the object. These methods return arrays of methods names. Those name are strings in Ruby 1.8 and symbols in Ruby 1.9:

```
o = "a string"
o.methods                    # => [ names of all public methods ]
o.public_methods             # => the same thing
o.public_methods(false)       # Exclude inherited methods
o.protected_methods  # => []: there aren't any
o.private_methods            # => array of all private methods
o.private_methods(false)     # Exclude inherited private methods
def o.single; 1; end         # Define a singleton method
o.singleton_methods          # => ["single"] (or [:single] in 1.9)
```

```
gemfiles/Gemfile.3.2.mysql2

.....................................

Finished in 0.11225 seconds (files took 0.35324 seconds to load)
36 examples, 0 failures


gemfiles/Gemfile.5.0.pg

.....................................

Finished in 0.09381 seconds (files took 0.35269 seconds to load)
36 examples, 0 failures


Summary

- gemfiles/Gemfile.3.2.mysql2  Ruby 2.1.8  Skipped
- gemfiles/Gemfile.4.2.mysql2  Ruby 2.1.8  Skipped
- gemfiles/Gemfile.4.2.pg      Ruby 2.1.8  Skipped
- gemfiles/Gemfile.3.2.mysql2  Ruby 2.2.4  Success
- gemfiles/Gemfile.4.2.mysql2  Ruby 2.2.4  Success
- gemfiles/Gemfile.4.2.pg      Ruby 2.2.4  Success
- gemfiles/Gemfile.5.0.mysql2  Ruby 2.2.4  Success
- gemfiles/Gemfile.5.0.pg      Ruby 2.2.4  Success
- gemfiles/Gemfile.3.2.mysql2  Ruby 2.3.1  Skipped
- gemfiles/Gemfile.4.2.mysql2  Ruby 2.3.1  Skipped
- gemfiles/Gemfile.4.2.pg      Ruby 2.3.1  Skipped
- gemfiles/Gemfile.5.0.mysql2  Ruby 2.3.1  Skipped
- gemfiles/Gemfile.5.0.pg      Ruby 2.3.1  Skipped

All gemfiles succeeded for Ruby 2.2.4.
```

It is also possible to query a class for the methods it defines rather than querying an instance of the class. The following methods are defined by Module. Like the Object methods, they return arrays of strings in Ruby 1.8 and arrays of symbols in 1.9:

```
String.instance_methods == "s".public_methods # => true
String.instance_methods(false) == "s".public_methods(false)        # => true
String.public_instance_methods == String.instance_methods          # => true
String.protected_instance_methods                                  # => []
String.private_instance_methods(false)                  # => ["initialize_copy",
 # "initialize"]
```

Recall that the class methods of a class or module are singleton methods of the Class or Module object. So to list class methods, use Object.singleton_methods:

```
Math.singleton_methods # => ["acos", "log10", "atan2", ... ]
```

In addition to these listing methods, the Module class defines some predicates for testing whether a specified class or module defines a named instance method:

```
String.public_method_defined? :reverse        # => true
String.protected_method_defined? :reverse     # => false
String.private_method_defined? :initialize    # => true
String.method_defined? :upcase!               # => true
```

Module.method_defined? checks whether the named method is defined as a public or protected method. It serves essentially the same purpose as Object.respond_to?. In Ruby 1.9, you can pass false as the second argument to specify that inherited methods should not be considered.

8.4.2 Obtaining Method Objects

Keyword

Proc object is an encapsulation of a block of code, which can be stored in a local variable, passed to a method or another Proc, and can be called. Proc is an essential concept in Ruby and a core of its functional programming features.

To query a specific named method, call method on any object or instance_method on any module. The former returns a callable Method object bound to the receiver, and the latter returns an UnboundMethod. In Ruby 1.9, you can limit your search to public methods by calling public_method and public_instance_method.

"s".method(:reverse) # => Method object String.instance_method(:reverse) # => UnboundMethod object

8.4.3 Invoking Method

You can use the method method of any object to obtain a Method object that represents a named method of that object. Method objects have a call method just like **Proc objects** do; you can use it to invoke the method.

Usually, it is simpler to invoke a named method of a specified object with send:

"hello".send :upcase # => "HELLO": invoke an instance method

Math.send(:sin, Math::PI/2) # => 1.0: invoke a class method

send invokes on its receiver the method named by its first argument, passing any remaining arguments to that method. The name "send" derives from the objectoriented idiom in

which invoking a method is called "sending a message" to an object.

send can invoke any named method of an object, including private and protected methods. We saw send used earlier to invoke the private method remove_const of a Module object. Because global functions are really private methods of Object, we can use send to invoke these methods on any object (though this is not anything that we'd ever actually want to do):

```
"hello".send :puts, "world" # prints "world"
```

Ruby 1.9 defines public_send as an alternative to send. This method works like send, but will only invoke public methods, not private or protected methods:

```
"hello".public_send :puts, "world" # raises NoMethodError
```

send is a very fundamental method of Object, but it has a common name that might be overridden in subclasses. Therefore, Ruby defines __send__ as a synonym, and issues a warning if you attempt to delete or redefine __send__.

8.4.4 Defining, Undefining, and Aliasing Methods

If you want to define a new instance method of a class or module, use define_method. This instance method of Module takes the name of the new method (as a Symbol) as its first argument. The body of the method is provided either by a Method object passed as the second argument or by a block. It is important to understand that define_method is private. You must be inside the class or module you want to use it on in order to call it:

```
# Add an instance method named m to class c with body b
def add_method(c, m, &b)
 c.class_eval {
 define_method(m, &b)
 }
end

add_method(String, :greet) { "Hello, " + self }
"world".greet # => "Hello, world"
```

To define a class method (or any singleton method) with define_method, invoke it on the eigenclass:

```
def add_class_method(c, m, &b)
```

```
 eigenclass = class << c; self; end
 eigenclass.class_eval {
 define_method(m, &b)
 }
end
add_class_method(String, :greet) {|name| "Hello, " + name }
String.greet("world")              # => "Hello, world"
```

In Ruby 1.9, you can more easily use define_singleton_method, which is a method of Object

```
String.define_singleton_method(:greet) {|name| "Hello, " + name }
```

One shortcoming of define_method is that it does not allow you to specify a method body that expects a block. If you need to dynamically create a method that accepts a block, you will need to use the def statement with class_eval. And if the method you are creating is sufficiently dynamic, you may not be able to pass a block to class_eval and will instead have to specify the method definition as a string to be evaluated.

To create a synonym or an alias for an existing method, you can normally use the alias statement:

```
alias plus + # Make "plus" a synonym for the + operator
```

When programming dynamically, however, you sometimes need to use alias_method instead. Like define_method, alias_method is a private method of Module. As a method, it can accept two arbitrary expressions as its arguments, rather than requiring two identifiers to be hardcoded in your source code. (As a method, it also requires a comma between its arguments.) alias_method is often used for alias chaining existing methods.

```
# Create an alias for the method m in the class (or module) c
def backup(c, m, prefix="original")
 n = :"#{prefix}_#{m}"    # Compute the alias
 c.class_eval {           # Because alias_method is private
alias_method n, m         # Make n an alias for m
 }
end
backup(String, :reverse)
"test".original_reverse   # => "tset"
```

You can use the undef statement to undefine a method. This works only if you can express the name of a method as a hardcoded identifier in your program. If you need to dynamically delete a method whose name has been computed by your program, you have two choices: remove_method or undef_method. Both are private methods of Module. remove_method removes the definition of the method from the current class. If there is a version defined by a superclass, that version will now be inherited. undef_method is more severe; it prevents any invocation of the specified method through an instance of the class, even if there is an inherited version of that method.

If you define a class and want to prevent any dynamic alterations to it, simply invoke the freeze method of the class. Once frozen, a class cannot be altered.

8.4.5 Handling Undefined Methods

When the method name resolution algorithm fails to find a method, it looks up a method named method_missing instead. When this method is invoked, the first argument is a symbol that names the method that could not be found. This symbol is followed by all the arguments that were to be passed to the original method. If there is a block associated with the method invocation, that block is passed to method_missing as well.

The default implementation of method_missing, in the Kernel module, simply raises a NoMethodError. This exception, if uncaught, causes the program to exit with an error message, which is what you would normally expect to happen when you try to invoke a method that does not exist.

Defining your own method_missing method for a class allows you an opportunity to handle any kind of invocation on instances of the class. The method_missing hook is one of the most powerful of Ruby's dynamic capabilities, and one of the most commonly used metaprogramming techniques. For now, the following example code adds a method_missing method to the Hash class. It allows us to query or set the value of any named key as if the key were the name of a method:

```
class Hash
 # Allow hash values to be queried and set as if they were attributes.
 # We simulate attribute getters and setters for any key.
 def method_missing(key, *args)
 text = key.to_s
 if text[-1,1] == "="                    # If key ends with = set a value

self[text.chop.to_sym] = args[0]         # Strip = from key
 else                                    # Otherwise...
 self[key]                               # ...just return the key value
```

```
 end
 end
end
h = {}                          # Create an empty hash object
h.one = 1                       # Same as h[:one] = 1
puts h.one                      # Prints 1. Same as puts h[:one]
```

8.4.6 Setting Method Visibility

We introduced public, protected, and private. These look like language keywords but are actually private instance methods defined by Module. These methods are usually used as a static part of a class definition. But, with class_eval, they can also be used dynamically:

```
String.class_eval { private :reverse }
"hello".reverse # NoMethodError: private method 'reverse'
```

private_class_method and public_class_method are similar, except that they operate on class methods and are themselves public:

```
# Make all Math methods private
# Now we have to include Math in order to invoke its methods
Math.private_class_method *Math.singleton_methods
```

8.5 HOOKS

Module, Class, and Object implement several callback methods, or hooks. These methods are not defined by default, but if you define them for a module, class, or object, then they will be invoked when certain events occur. This gives you an opportunity to extend Ruby's behavior when classes are subclassed, when modules are included, or when methods are defined. Hook methods have names that end in "ed."

When a new class is defined, Ruby invokes the class method inherited on the superclass of the new class, passing the new class object as the argument. This allows classes to add behavior to or enforce constraints on their descendants. Recall that class methods are inherited, so that the an inherited method will be invoked if it is defined by any of the ancestors of the new class. Define Object.inherited to receive notification of all new classes that are defined:

```
def Object.inherited(c)
```

```
 puts "class #{c} < #{self}"
end
```

When a module is included into a class or into another module, the included class method of the included module is invoked with the class or module object into which it was included as an argument. This gives the included module an opportunity to augment or alter the class in whatever way it wants—it effectively allows a module to define its own meaning for include. In addition to adding methods to the class into which it is included, a module with an included method might also alter the existing methods of that class, for example:

```
module Final              # A class that includes Final can't be subclassed
 def self.included(c)     # When included in class c
 c.instance_eval do       # Define a class method of c
 def inherited(sub)       # To detect subclasses
 raise Exception,         # And abort with an exception
 "Attempt to create subclass #{sub} of Final class #{self}"
 end
 end
 end
end
```

Similarly, if a module defines a class method named extended, that method will be invoked any time the module is used to extend an object (with Object.extend). The argument to the extended method will be the object that was extended, of course, and the extended method can take whatever actions it wants on that object.

In addition to hooks for tracking classes and the modules they include, there are also hooks for tracking the methods of classes and modules and the singleton methods of arbitrary objects. Define a class method named method_added for any class or module and it will be invoked when an instance method is defined for that class or module:

```
def String.method_added(name)
 puts "New instance method #{name} added to String"
end
```

Note that the method_added class method is inherited by subclasses of the class on which it is defined. But no class argument is passed to the hook, so there is no way to tell whether the named method was added to the class that defines method_added or whether it was added to a subclass of that class. A workaround for this problem is to define an inherited hook on any class that defines a method_added hook. The inherited method can then define a method_added method for each subclass.

When a singleton method is defined for any object, the method singleton_method_added is invoked on that object, passing the name of the new method. Remember that for classes, singleton methods are class methods:

```
def String.singleton_method_added(name)
 puts "New class method #{name} added to String"
end
```

Interestingly, Ruby invokes this singleton_method_added hook when the hook method itself is first defined. Here is another use of the hook. In this case, singleton_method_added is defined as an instance method of any class that includes a module. It is notified of any singleton methods added to instances of that class:

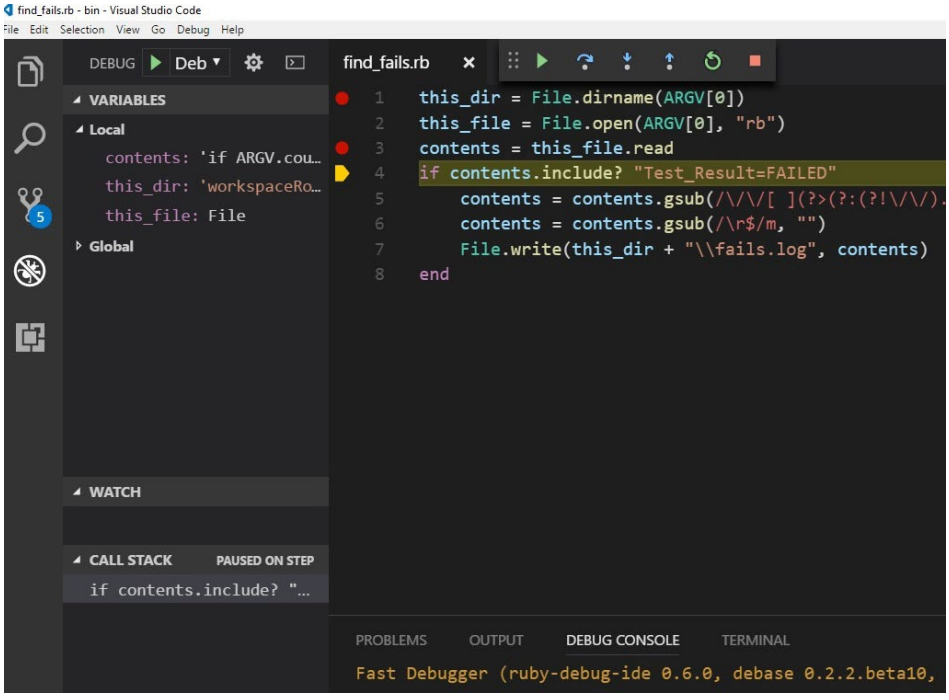
```
# Including this module in a class prevents instances of that class
# from having singleton methods added to them. Any singleton methods added
# are immediately removed again.
module Strict
 def singleton_method_added(name)
 STDERR.puts "Warning: singleton #{name} added to a Strict object"
 eigenclass = class << self; self; end
 eigenclass.class_eval { remove_method name }
 end
end
```

In addition to method_added and singleton_method_added, there are hooks for tracking when instance methods and singleton methods are removed or undefined. When an instance method is removed or undefined on a class or module, the class methods method_removed and method_undefined are invoked on that module. When a singleton method is removed or undefined on an object, the methods singleton_method_removed and singleton_method_undefined are invoked on that object.

8.6 TRACING

Ruby defines a number of features for tracing the execution of a program. These are mainly useful for debugging code and printing informative error messages. Two of the simplest features are actual language keywords: __FILE__ and __LINE__. These keyword expressions always evaluate to the name of the file and the line number within that file on which they appear, and they allow an error message to specify the exact location at which it was generated:

```
STDERR.puts "#{__FILE__}:#{__LINE__}: invalid data"
```



As an aside, note that the methods Kernel.eval, Object.instance_eval, and Module.class_eval all accept a filename (or other string) and a line number as their final two arguments. If you are evaluating code that you have extracted from a file of some sort, you can use these arguments to specify the values of __FILE__ and __LINE__ for the evaluation.

You have undoubtedly noticed that when an exception is raised and not handled, the error message printed to the console contains filename and line number information. This information is based on __FILE__ and __LINE__, of course. Every Exception object has a backtrace associated with it that shows exactly where it was raised, where the method that raised the exception was invoked, where that method was invoked, and

so on. The Exception.backtrace method returns an array of strings containing this information. The first element of this array is the location at which the exception occurred, and each subsequent element is one stack frame higher. You need not raise an exception to obtain a current stack trace, however. The Kernel.caller method returns the current state of the call stack in the same form as

Stack traces returned by Exception.backtrace and Kernel.caller also include method names. Prior to Ruby 1.9, you must parse the stack trace strings to extract method names. In Ruby 1.9, however, you can obtain the name (as a symbol) of the currently executing method with Kernel.__method__ or its synonym, Kernel.__callee__. __method__ is useful in conjunction with __FILE__ and __LINE__:

```
raise "Assertion failed in #{__method__} at #{__FILE__}:#{__LINE__}"
```

Note that __method__ returns the name by which a method was originally defined, even if the method was invoked through an alias.

Instead of simply printing the filename and number at which an error occurs, you can take it one step further and display the actual line of code. If your program defines a global constant named SCRIPT_LINES__ and sets it equal to a hash, then the require and load methods add an entry to this hash for each file they load. The hash keys are filenames and the values associated with those keys are arrays that contain the lines of those files. If you want to include the main file (rather than just the files it requires) in the hash, initialize it like this:

```
SCRIPT_LINES__ = {__FILE__ => File.readlines(__FILE__)}
```

If you do this, then you can obtain the current line of source code anywhere in your program with this expression:

```
SCRIPT_LINES__[__FILE__][__LINE__-1]
```

Ruby allows you to trace assignments to global variables with Kernel.trace_var. Pass this method a symbol that names a global variable and a string or block of code. When the value of the named variable changes, the string will be evaluated or the block will be invoked. When a block is specified, the new value of the variable is passed as an argument.

To stop tracing the variable, call Kernel.untrace_var. In the following, note the use of caller[1] to determine the program location at which the variable tracing block was invoked:


```
# Print a message every time $SAFE changes

trace_var(:$SAFE) {|v|

 puts "$SAFE set to #{v} at #{caller[1]}"

}
```

The final tracing method is Kernel.set_trace_func, which registers a Proc to be invoked after every line of a Ruby program. set_trace_func is useful if you want to write a debugger module that allows line-by-line stepping through a program, but we will not cover it in any detail here.

8.7 OBJECTSPACE AND GC

The ObjectSpace module defines a handful of low-level methods that can be occasionally useful for debugging or metaprogramming. The most notable method is each_object, an iterator that can yield every object (or every instance of a specified class) that the interpreter knows about:

```
# Print out a list of all known classes
ObjectSpace.each_object(Class) {|c| puts c }
```

ObjectSpace._id2ref is the inverse of Object.object_id: it takes an object ID as its argument and returns the corresponding object, or raises a RangeError if there is no object with that ID.

ObjectSpace.define_finalizer allows the registration of a Proc or a block of code to be invoked when a specified object is garbage collected. You must be careful when registering such a finalizer, however, as the finalizer block is not allowed to use the garbage collected object.

The final ObjectSpace method is ObjectSpace.garbage_collect, which forces Ruby's garbage collector to run. Garbage collection functionality is also available through the GC module. GC.start is a synonym for ObjectSpace.garbage_collect. Garbage collection can be temporarily disabled with GC.disable, and it can be enabled again with GC.enable.

Any values required to finalize the object must be captured in the scope of the finalizer block, so that they are available without dereferencing the object. Use ObjectSpace.undefine_finalizer to delete all finalizer blocks registered for an object.

The combination of the _id2ref and define_finalizer methods allows the definition of "weak reference" objects, which hold a reference to a value without preventing the value from being garbage collected if they become otherwise unreachable. See the WeakRef class in the standard library (in lib/weakref.rb) for an example.

8.8 CUSTOM CONTROL STRUCTURES

Ruby's use of blocks, coupled with its parentheses-optional syntax, make it very easy to define iterator methods that look like and behave like control structures. The loop method of Kernel is a simple example.

8.8.1 Delaying and Repeating Execution: after and every

Example 1 defines global methods named after and every. Each takes a numeric argument that represents a number of seconds and should have a block associated with it after creates a new thread and returns the Thread object immediately. The newly created thread sleeps for the specified number of seconds and then calls (with no arguments) the block you provided. Every is similar, but it calls the block repeatedly, sleeping the specified number of seconds between calls. The second argument to every is a value to pass to the first invocation of the block. The return value of each invocation becomes the value passed for the next invocation. The block associated with every can use break to prevent any future invocations.

Here is some example code that uses after and every:

```
require 'afterevery'
1.upto(5) {|i| after i { puts i} }          # Slowly print the numbers 1 to 5
sleep(5)                                    # Wait five seconds
every 1, 6 do |count|                       # Now slowly print 6 to 10
 puts count
 break if count == 10
 count + 1                                  # The next value of count
end
sleep(6) 1                                  # Give the above time to run
```

Example 1. The after and every methods

```
#
# Define Kernel methods after and every for deferring blocks of code.
# Examples:
```

```
#
# after 1 { puts “done” }
# every 60 { redraw_clock }
#
# Both methods return Thread objects. Call kill on the returned objects
# to cancel the execution of the code.
#
# Note that this is a very naive implementation. A more robust
# implementation would use a single global timer thread for all tasks,
# would allow a way to retrieve the value of a deferred block, and would
# provide a way to wait for all pending tasks to complete.
#
# Execute block after sleeping the specified number of seconds.
def after(seconds, &block)
 Thread.new do # In a new thread...
 sleep(seconds) # First sleep
 block.call # Then call the block

end                # Return the Thread object right away
end
# Repeatedly sleep and then execute the block.
# Pass value to the block on the first invocation.
# On subsequent invocations, pass the value of the previous invocation.
def every(seconds, value=nil, &block)
 Thread.new do          # In a new thread...
 loop do                # Loop forever (or until break in block)
 sleep(seconds)                 # Sleep
 value = block.call(value)      # And invoke block
 end                            # Then repeat..
 end                            # every returns the Thread
end
```

8.8.2 Thread Safety with Synchronized Blocks

When writing programs that use multiple threads, it is important that two threads do not attempt to modify the same object at the same time. One way to do this is to place the code that must be made thread-safe in a block associated with a call to the synchronize method of a Mutex object. In Example 2 we take this a step further, and emulate Java's synchronized keyword with a global method named synchronized. This synchronized method expects a single object argument and a block. It obtains a Mutex associated with the object, and uses Mutex.synchronize to invoke the block. The tricky part is that Ruby's object, unlike Java's objects, do not have a Mutex associated with them. So Example 2 also defines an instance method named mutex in Object. Interestingly, the implementation of this mutex method uses synchronized in its new keyword-style form!

Example 2. Simple synchronized blocks

```
require 'thread'          # Ruby 1.8 keeps Mutex in this library
# Obtain the Mutex associated with the object o, and then evaluate
# the block under the protection of that Mutex.
# This works like the synchronized keyword of Java.
def synchronized(o)
 o.mutex.synchronize { yield }
end

# Object.mutex does not actually exist. We've got to define it.
# This method returns a unique Mutex for every object, and
# always returns the same Mutex for any particular object.
# It creates Mutexes lazily, which requires synchronization for
# thread safety.
class Object
 # Return the Mutex for this object, creating it if necessary.
 # The tricky part is making sure that two threads don't call
 # this at the same time and end up creating two different mutexes.
 def mutex
 # If this object already has a mutex, just return it
 return @__mutex if @__mutex

# Otherwise, we've got to create a mutex for the object.
```

```
 # To do this safely we've got to synchronize on our class object.
 synchronized(self.class) {
 # Check again: by the time we enter this synchronized block,
 # some other thread might have already created the mutex.
 @__mutex = @__mutex || Mutex.new
 }
 # The return value is @__mutex
 end
end

# The Object.mutex method defined above needs to lock the class
# if the object doesn't have a Mutex yet. If the class doesn't have
# its own Mutex yet, then the class of the class (the Class object)
# will be locked. In order to prevent infinite recursion, we must
# ensure that the Class object has a mutex.
Class.instance_eval { @__mutex = Mutex.new }
```

8.9 MISSING METHODS AND MISSING CONSTANTS

The method_missing method is a key part of Ruby's method lookup algorithm and provides a powerful way to catch and handle arbitrary invocations on an object. The const_missing method of Module performs a similar function for the constant lookup algorithm and allows us to compute or lazily initialize constants on the fly. The examples that follow demonstrate both of these methods.

8.9.1 Unicode Codepoint Constants with const_missing

Example 3 defines a Unicode module that appears to define a constant (a UTF-8 encoded string) for every Unicode codepoint from U+0000 to U+10FFFF. The only practical way to support this many constants is to use the const_missing method. The code makes the assumption that if a constant is referenced once, it is likely to be used again, so the const_missing method calls Module.const_set to define a real constant to refer to each value it computes.

Example 3. Unicode codepoint constants with const_missing

```
# This module provides constants that define the UTF-8 strings for
# all Unicode codepoints. It uses const_missing to define them lazily.
```

```
# Examples:
# copyright = Unicode::U00A9
# euro = Unicode::U20AC
# infinity = Unicode::U221E
module Unicode
 # This method allows us to define Unicode codepoint constants lazily.
 def self.const_missing(name) # Undefined constant passed as a symbol
 # Check that the constant name is of the right form.
 # Capital U followed by a hex number between 0000 and 10FFFF.
 if name.to_s =~ /^U([0-9a-fA-F]{4,5}|10[0-9a-fA-F]{4})$/
 # $1 is the matched hexadecimal number. Convert to an integer.

codepoint = $1.to_i(16)
 # Convert the number to a UTF-8 string with the magic of Array.pack.
 utf8 = [codepoint].pack("U")
 # Make the UTF-8 string immutable.
 utf8.freeze
 # Define a real constant for faster lookup next time, and return
 # the UTF-8 text for this time.
 const_set(name, utf8)
 else
 # Raise an error for constants of the wrong form.
 raise NameError, "Uninitialized constant: Unicode::#{name}"
 end
 end
end
```

8.9.2 Tracing Method Invocations with method_missing

We demonstrated an extension to the Hash class using method_missing. Now, in Example 4, we demonstrate the use of method_missing to delegate arbitrary calls on one object to another object. In this example, we do this in order to output tracing messages for the object.

Example 4 defines an Object.trace instance method and a TracedObject class. The trace method returns an instance of TracedObject that uses method_missing to catch invocations, trace them, and delegate them to the object being traced. You might use it like this:

```
a = [1,2,3].trace("a")
a.reverse
puts a[2]
puts a.fetch(3)
```

This produces the following tracing output:

```
Invoking: a.reverse() at trace1.rb:66
Returning: [3, 2, 1] from a.reverse to trace1.rb:66
Invoking: a.fetch(3) at trace1.rb:67
Raising: IndexError:index 3 out of array from a.fetch
```

Notice that in addition to demonstrating method_missing, Example 4 also demonstrates Module.instance_methods, Module.undef_method, and Kernel.caller.

Example 4. Tracing method invocations with method_missing

```
# Call the trace method of any object to obtain a new object that
# behaves just like the original, but which traces all method calls
# on that object. If tracing more than one object, specify a name to
# appear in the output. By default, messages will be sent to STDERR,
# but you can specify any stream (or any object that accepts strings # as arguments to <<).
class Object
 def trace(name="", stream=STDERR)
 # Return a TracedObject that traces and delegates everything else to us.
 TracedObject.new(self, name, stream)
end
end

# This class uses method_missing to trace method invocations and
# then delegate them to some other object. It deletes most of its own
```

```
# instance methods so that they don't get in the way of method_missing.
# Note that only methods invoked through the TracedObject will be traced.
# If the delegate object calls methods on itself, those invocations
# will not be traced.

class TracedObject
 # Undefine all of our noncritical public instance methods.
 # Note the use of Module.instance_methods and Module.undef_method.
 instance_methods.each do |m|
 m = m.to_sym # Ruby 1.8 returns strings, instead of symbols
 next if m == :object_id || m == :__id__ || m == :__send__
 undef_method m
 end

# Initialize this TracedObject instance.
 def initialize(o, name, stream)
 @o = o # The object we delegate to
 @n = name # The object name to appear in tracing messages
 @trace = stream # Where those tracing messages are sent
 End

# This is the key method of TracedObject. It is invoked for just
 # about any method invocation on a TracedObject.
 def method_missing(*args, &block)
 m = args.shift # First arg is the name of the method
 begin
 # Trace the invocation of the method.
 arglist = args.map {|a| a.inspect}.join(', ')
 @trace << "Invoking: #{@n}.#{m}(#{arglist}) at #{caller[0]}\n"
 # Invoke the method on our delegate object and get the return value.
 r = @o.send m, *args, &block
 # Trace a normal return of the method.
 @trace << "Returning: #{r.inspect} from #{@n}.#{m} to #{caller[0]}\n"
```

```
 # Return whatever value the delegate object returned.
 r

rescue Exception => e
 # Trace an abnormal return from the method.
 @trace << "Raising: #{e.class}:#{e} from #{@n}.#{m}\n"
 # And re-raise whatever exception the delegate object raised.
 raise
 end
 end

# Return the object we delegate to.
 def __delegate
 @o
 end
end
```

8.9.3 Synchronized Objects by Delegation

In Example 2, we saw a global method synchronized, which accepts an object and executes a block under the protection of the Mutex associated with that object. Most of the example consisted of the implementation of the Object.mutex method. The synchronized method was trivial:

```
def synchronized(o)
 o.mutex.synchronize { yield }
end
```

Example 5 modifies this method so that, when invoked without a block, it returns a SynchronizedObject wrapper around the object. SynchronizedObject is a delegating wrapper class based on method_missing. It is much like the TracedObject class of Example 4, but it is written as a subclass of Ruby 1.9's BasicObject, so there is no need to explicitly delete the instance methods of Object. Note that the code in this example does not stand alone; it requires the Object.mutex method defined earlier.

Example 5. Synchronizing methods with method_missing

```
def synchronized(o)
 if block_given?
 o.mutex.synchronize { yield }
```

```
 else
 SynchronizedObject.new(o)
 end
end

# A delegating wrapper class using method_missing for thread safety
# Instead of extending Object and deleting our methods we just extend
# BasicObject, which is defined in Ruby 1.9. BasicObject does not
# inherit from Object or Kernel, so the methods of a BasicObject cannot
# invoke any top-level methods: they are just not there.
class SynchronizedObject < BasicObject
 def initialize(o); @delegate = o; end
 def __delegate; @delegate; end
 def method_missing(*args, &block)
 @delegate.mutex.synchronize {
 @delegate.send *args, &block
 }
 end
end
```

8.10 DYNAMICALLY CREATING METHODS

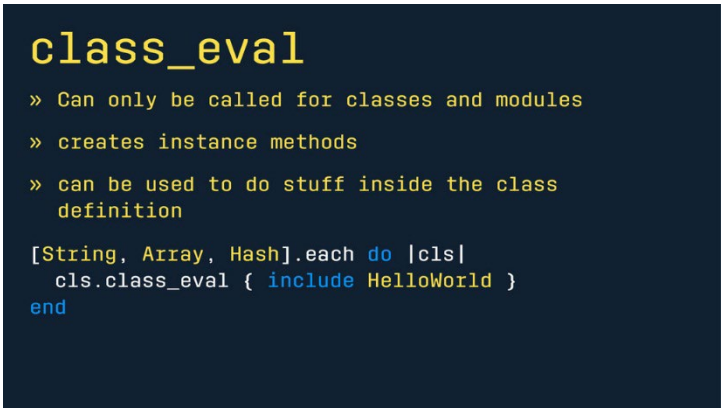
One important metaprogramming technique is the use of methods that create methods. The attr_reader and attr_accessor methods are examples. These private instance methods of Module are used like keywords within class definitions. They accept attribute names as their arguments, and dynamically create methods with those names.

The examples that follow are variants on these attribute accessor creation methods and demonstrate two different ways to dynamically create methods like this.

8.10.1 Defining Methods with class_eval

Example 6 defines private instance methods of Module named readonly and readwrite. These methods work like attr_reader and attr_accessor do, and they are here to demonstrate how those methods are implemented. The implementation is actually quite simple: readonly and readwrite first build a string of Ruby code containing the def statements required to define appropriate accessor methods. Next, they evaluate that

string of code using class_eval. Using class_eval like this incurs the slight overhead of parsing the string of code. The benefit, however, is that the methods we define need not use any reflective APIs themselves; they can query or set the value of an instance variable directly.



Example 6. Attribute methods with class_eval

```
class Module
 private # The methods that follow are both private
 # This method works like attr_reader, but has a shorter name
 def readonly(*syms)
 return if syms.size == 0 # If no arguments, do nothing
 code = “” # Start with an empty string of code
 # Generate a string of Ruby code to define attribute reader methods.
 # Notice how the symbol is interpolated into the string of code.
 syms.each do |s| # For each symbol
 code << “def #{s}; @#{s}; end\n” # The method definition
end

 # Finally, class_eval the generated code to create instance methods.
 class_eval code
 end

# This method works like attr_accessor, but has a shorter name.
 def readwrite(*syms)
 return if syms.size == 0
```

```
  code = ""
  syms.each do |s|
  code << "def #{s}; @#{s} end\n"
  code << "def #{s}=(value); @#{s} = value; end\n"
  end
  class_eval code
  end
end
```

8.10.2 Defining Methods with define_method

Example 7 is a different take on attribute accessors. The attributes method is something like the readwrite method defined in Example 6. Instead of taking any number of attribute names as arguments, it expects a single hash object. This hash should have attribute names as its keys, and should map those attribute names to the default values for the attributes. The class_attrs method works like attributes, but defines class attributes rather than instance attributes.

Remember that Ruby allows the curly braces to be omitted around hash literals when they are the final argument in a method invocation. So the attributes method might be invoked with code like this:

```
class Point
 attributes :x => 0, :y => 0
end
```

In Ruby 1.9, we can use the more succinct hash syntax:

```
class Point
 attributes x:0, y:0
end
```

This is another example that leverages Ruby's flexible syntax to create methods that behave like language keywords.

The implementation of the attributes method in Example 7 is quite a bit different than that of the readwrite method in Example 6. Instead of defining a string of Ruby code and evaluating it with class_eval, the attributes method defines the body of the attribute accessors in a block and defines the methods using define_method. Because

this method definition technique does not allow us to interpolate identifiers directly into the method body, we must rely on reflective methods such as instance_variable_get. Because of this, the accessors defined with attributes are likely to be less efficient than those defined with readwrite.

An interesting point about the attributes method is that it does not explicitly store the default values for the attributes in a class variable of any kind. Instead, the default value for each attribute is captured by the scope of the block used to define the method.

The class_attrs method defines class attributes very simply: it invokes attributes on the eigenclass of the class. This means that the resulting methods use class instance variables instead of regular class variables.

Example 7. Attribute methods with define_method

```
class Module
 # This method defines attribute reader and writer methods for named
 # attributes, but expects a hash argument mapping attribute names to
 # default values. The generated attribute reader methods return the
 # default value if the instance variable has not yet been defined.
 def attributes(hash)
 hash.each_pair do |symbol, default|          # For each attribute/default pair
 getter = symbol                              # Name of the getter method
 setter = :"#{symbol}="                       # Name of the setter method
 variable = :"@#{symbol}"                     # Name of the instance variable
 define_method getter do                      # Define the getter method
 if instance_variable_defined? variable

instance_variable_get variable                # Return variable, if defined
 else
 default                                      # Otherwise return default
 end
 end
 define_method setter do |value|              # Define setter method
 instance_variable_set variable,              # Set the instance variable
 value                                        # To the argument value
end
```

```
 end
 end

# This method works like attributes, but defines class methods instead
 # by invoking attributes on the eigenclass instead of on self.
 # Note that the defined methods use class instance variables
 # instead of regular class variables.
 def class_attrs(hash)
 eigenclass = class << self; self; end
 eigenclass.class_eval { attributes(hash) }
 end
 # Both methods are private
 private :attributes, :class_attrs
end
```

8.11 ALIAS CHAINING

Metaprogramming in Ruby often involves the dynamic definition of methods. Just as common is the dynamic modification of methods. Methods are modified with a technique we'll call alias chaining.* It works like this:

- First, create an alias for the method to be modified. This alias provides a name for the unmodified version of the method.
- Next, define a new version of the method. This new version should call the unmodified version through the alias, but it can add whatever functionality is needed before and after it does that.

```
"baseUrl": "./src",
  "paths": {
      "Screens/*": ["screens/*"],
      "Common/*": ["common/*"],
      "Components/*": ["components/*"],
      "Assets/*": ["assets/*"],
      "Services/*": ["services/*"],
  }
```

Note that these steps can be applied repeatedly (as long as a different alias is used

each time), creating a chain of methods and aliases. We include three alias chaining examples. The first performs the alias chaining statically; i.e., using regular alias and def statements. The second and third examples are more dynamic; they alias chain arbitrarily named methods using alias_method, define_method, and class_eval.

8.11.1 Tracing Files Loaded and Classes Defined

Example 8 is code that keeps track of all files loaded and all classes defined in a program. When the program exits, it prints a report. You can use this code to "instrument" an existing program so that you better understand what it is doing. One way to use this code is to insert this line at the beginning of the program:

```
require 'classtrace'
```

An easier solution, however, is to use the -r option to your Ruby interpreter:

```
ruby -rclasstrace my_program.rb --traceout /tmp/trace
```

The -r option loads the specified library before it starts running the program.

Example 8 uses static alias chaining to trace all invocations of the Kernel.require and Kernel.load methods. It defines an Object.inherited hook to track definitions of new classes. And it uses Kernel.at_exit to execute a block of code when the program terminates. Besides alias chaining require and load and defining Object.inherited, the only modification to the global namespace made by this code is the definition of a module named ClassTrace. All state required for tracing is stored in constants within this module, so that we do not pollute the namespace with global variables.

Example 8. Tracing files loaded and classes defined

```
# We define this module to hold the global state we require, so that
# we don't alter the global namespace any more than necessary.
module ClassTrace
 # This array holds our list of files loaded and classes defined.
 # Each element is a subarray holding the class defined or the
 # file loaded and the stack frame where it was defined or loaded.
 T = [] # Array to hold the files loaded

# Now define the constant OUT to specify where tracing output goes.
 # This defaults to STDERR, but can also come from command-line arguments
```

```
 if x = ARGV.index("--traceout") # If argument exists
 OUT = File.open(ARGV[x+1], "w") # Open the specified file
 ARGV[x,2] = nil # And remove the arguments
 else
 OUT = STDERR # Otherwise default to STDERR
 end
end

# Alias chaining step 1: define aliases for the original methods
alias original_require require
alias original_load load
# Alias chaining step 2: define new versions of the methods
def require(file)
 ClassTrace::T << [file,caller[0]]          # Remember what was loaded where
 original_require(file)                     # Invoke the original method
end


def load(*args)
 ClassTrace::T << [args[0],caller[0]]       # Remember what was loaded where
 original_load(*args)                       # Invoke the original method
end

# This hook method is invoked each time a new class is defined
def Object.inherited(c)
 ClassTrace::T << [c,caller[0]]             # Remember what was defined where
end

# Kernel.at_exit registers a block to be run when the program exits
# We use it to report the file and class data we collected
at_exit {
 o = ClassTrace::OUT
 o.puts "="*60
 o.puts "Files Loaded and Classes Defined:"
```

```
 o.puts "="*60
 ClassTrace::T.each do |what,where|
 if what.is_a? Class # Report class (with hierarchy) defined
 o.puts "Defined: #{what.ancestors.join('<-')} at #{where}"
 else # Report file loaded
 o.puts "Loaded: #{what} at #{where}"
 end
 end
}
```

8.11.2 Chaining Methods for Thread Safety

Example 2 defined a synchronized method (based on an Object.mutex method) that executed a block under the protection of a Mutex object. Then, Example 5 redefined the synchronized method so that when it was invoked without a block, it would return a SynchronizedObject wrapper around an object, protecting access to any methods invoked through that wrapper object. Now, in Example 9, we augment the synchronized method again so that when it is invoked within a class or module definition, it alias chains the named methods to add synchronization.

The alias chaining is done by our method Module.synchronize_method, which in turn uses a helper method Module.create_alias to define an appropriate alias for any given method (including operator methods like +).

After defining these new Module methods, Example 9 redefines the synchronized method again. When the method is invoked within a class or a module, it calls synchronize_method on each of the symbols it is passed. Interestingly, however, it can also be called with no arguments; when used this way, it adds synchronization to whatever instance method is defined next. (It uses the method_added hook to receive notifications when a new method is added.) Note that the code in this example depends on the Object.mutex method of Example 2 and the SynchronizedObject class of Example 5.

Example 9. Alias chaining for thread safety

```
# Define a Module.synchronize_method that alias chains instance methods
# so they synchronize on the instance before running.
class Module
 # This is a helper function for alias chaining.
 # Given a method name (as a string or symbol) and a prefix, create
```

```
 # a unique alias for the method, and return the name of the alias
 # as a symbol. Any punctuation characters in the original method name
 # will be converted to numbers so that operators can be aliased.
 def create_alias(original, prefix="alias")
 # Stick the prefix on the original name and convert punctuation
 aka = "#{prefix}_#{original}"
 aka.gsub!(/([\=\|\&\+\-\*\/\^\!\?\~\%\<\>\[\]])/) {
 num = $1[0] # Ruby 1.8 character -> ordinal
 num = num.ord if num.is_a? String # Ruby 1.9 character -> ordinal
 '_' + num.to_s
 }
# Keep appending underscores until we get a name that is not in use
 aka += "_" while method_defined? aka or private_method_defined? aka
 aka = aka.to_sym                    # Convert the alias name to a symbol
 alias_method aka, original          # Actually create the alias
 aka                                 # Return the alias name
 end

# Alias chain the named method to add synchronization
 def synchronize_method(m)
 # First, make an alias for the unsynchronized version of the method.
 aka = create_alias(m, "unsync")
 # Now redefine the original to invoke the alias in a synchronized block.
 # We want the defined method to be able to accept blocks, so we
 # can't use define_method, and must instead evaluate a string with
 # class_eval. Note that everything between %Q{ and the matching }
 # is a double-quoted string, not a block.
 class_eval %Q{
 def #{m}(*args, &block)
 synchronized(self) { #{aka}(*args, &block) }
 end
 }
 end
```

```
end

# This global synchronized method can now be used in three different ways.
def synchronized(*args)
 # Case 1: with one argument and a block, synchronize on the object
 # and execute the block
 if args.size == 1 && block_given?
 args[0].mutex.synchronize { yield }

# Case two: with one argument that is not a symbol and no block
 # return a SynchronizedObject wrapper
 elsif args.size == 1 and not args[0].is_a? Symbol and not block_given?
 SynchronizedObject.new(args[0])

# Case three: when invoked on a module with no block, alias chain the
 # named methods to add synchronization. Or, if there are no arguments,
 # then alias chain the next method defined.
 elsif self.is_a? Module and not block_given?
 if (args.size > 0) # Synchronize the named methods
 args.each {|m| self.synchronize_method(m) }
 else

# Case 4: any other invocation is an error
 else
 raise ArgumentError, "Invalid arguments to synchronize()"
 end
end
```

8.11.3 Chaining Methods for Tracing

Example 10 is a variant on Example 4 that supports tracing of named methods of an object. Example 4 used delegation and method_missing to define an Object.trace method that would return a traced wrapper object. This version uses chaining to alter

methods of an object in place. It defines trace! and untrace! to chain and unchain named methods of an object.

The interesting thing about this example is that it does its chaining in a different way from Example 9; it simply defines singleton methods on the object and uses super within the singleton to chain to the original instance method definition. No method aliases are created.

Example 8. Chaining with singleton methods for tracing

```
# Define trace! and untrace! instance methods for all objects.
# trace! "chains" the named methods by defining singleton methods
# that add tracing functionality and then use super to call the original.
# untrace! deletes the singleton methods to remove tracing.
class Object
 # Trace the specified methods, sending output to STDERR.
 def trace!(*methods)
 @_traced = @_traced || []       # Remember the set of traced methods

# If no methods were specified, use all public methods defined
 # directly (not inherited) by the class of this object
 methods = public_methods(false) if methods.size == 0
methods.map! {|m| m.to_sym }                    # Convert any strings to symbols
 methods -= @_traced                            # Remove methods that are already traced
 return if methods.empty?                       # Return early if there is nothing to do
 @_traced |= methods                            # Add methods to set of traced methods

# Trace the fact that we're starting to trace these methods
 STDERR << "Tracing #{methods.join(', ')} on #{object_id}\n"

# Singleton methods are defined in the eigenclass
 eigenclass = class << self; self; end
```

```
methods.each do |m|              # For each method m
 # Define a traced singleton version of the method m.
 # Output tracing information and use super to invoke the
 # instance method that it is tracing.
 # We want the defined methods to be able to accept blocks, so we
 # can't use define_method, and must instead evaluate a string.
 # Note that everything between %Q{ and the matching } is a
 # double-quoted string, not a block. Also note that there are
 # two levels of string interpolations here. #{} is interpolated
 # when the singleton method is defined. And \#{} is interpolated
 # when the singleton method is invoked.
 eigenclass.class_eval %Q{
 def #{m}(*args, &block)
begin
 STDERR << "Entering: #{m}(\#{args.join(', ')})\n"
 result = super
 STDERR << "Exiting: #{m} with \#{result}\n"
 result
 rescue
 STDERR << "Aborting: #{m}: \#{$!.class}: \#{$!.message}"
 raise
 end
 end
 }
 end
 end

# Untrace the specified methods or all traced methods
 def untrace!(*methods)
 if methods.size == 0 # If no methods specified untrace
 methods = @_traced # all currently traced methods
 STDERR << "Untracing all methods on #{object_id}\n"
 else # Otherwise, untrace
```

```
 methods.map! {|m| m.to_sym } # Convert string to symbols
 methods &= @_traced # all specified methods that are traced
 STDERR << "Untracing #{methods.join(', ')} on #{object_id}\n"
 end
@_traced -= methods # Remove them from our set of traced methods
 # Remove the traced singleton methods from the eigenclass
 # Note that we class_eval a block here, not a string
 (class << self; self; end).class_eval do

methods.each do |m|
 remove_method m # undef_method would not work correctly
 end
 end
 # If no methods are traced anymore, remove our instance var
 if @_traced.empty?
 remove_instance_variable :@_traced
 end
 end
end
```

8.12 DOMAIN-SPECIFIC LANGUAGES

The goal of metaprogramming in Ruby is often the creation of domain-specific languages, or DSLs. A DSL is just an extension of Ruby's syntax (with methods that look like keywords) or API that allows you to solve a problem or represent data more naturally than you could otherwise. For our examples, we will take the problem domain to be the output of XML formatted data, and we will define two DSLs—one very simple and one more clever—to tackle this problem.

8.12.1 Simple XML Output with method_missing

We begin with a simple class named XML for generating XML output. Here's an example of how the XML can be used:

```
pagetitle = "Test Page for XML.generate"
XML.generate(STDOUT) do
 html do
 head do
 title { pagetitle }
 comment "This is a test"
 end
 body do
 h1(:style => "font-family:sans-serif") { pagetitle }
 ul :type=>"square" do
 li { Time.now }
 li { RUBY_VERSION }
 end
 end
 end
end
```

This code does not look like XML, and it only sort of looks like Ruby. Here's the output it generates (with some line breaks added for legibility):

<html><head>
Test Page for XML.generate</little>
<!-- This is a test -->
</head><body>
< h1 style='font-family:sans-serif' > Test Page for XML.generate</h1>
<ul type='square'>
2007-08-19 16:19:58 -0700
1.9.0
</body></html>

To implement this class and the XML generation syntax it supports, we rely on:

- Ruby's block structure
- Ruby's parentheses-optional method invocations
- Ruby's syntax for passing hash literals to methods without curly braces
- The method_missing method

Example 11 shows the implementation for this simple DSL.

Example 11. A simple DSL for generating XML output

```
class XML
 # Create an instance of this class, specifying a stream or object to
 # hold the output. This can be any object that responds to <<(String).
4 def initialize(out)
 @out = out # Remember where to send our output
 end

# Output the specified object as CDATA, return nil.
 def content(text)
 @out << text.to_s
 nil
 end
```

```
    # Output the specified object as a comment, return nil.
     def comment(text)
     @out << "<!-- #{text} -->"
nil

     end

    # Output a tag with the specified name and attributes.
     # If there is a block invoke it to output or return content.
     # Return nil.
     def tag(tagname, attributes={})
     # Output the tag name
     @out << "<#{tagname}"

    # Output the attributes
     attributes.each {|attr,value| @out << " #{attr}='#{value}'" }

     if block_given?
     # This block has content
     @out << '>'                    # End the opening tag
     content = yield                # Invoke the block to output or return content
     if content                     # If any content returned
     @out << content.to_s           # Output it as a string
     End
     @out << "</#{tagname}>"        # Close the tag

    else
     # Otherwise, this is an empty tag, so just close it.
     @out << '/>'
     end
     nil # Tags output themselves, so they don't return any content
     end
     # The code below is what changes this from an ordinary class into a DSL.
```

```
 # First: any unknown method is treated as the name of a tag.
 alias method_missing tag
 # Second: run a block in a new instance of the class.
 def self.generate(out, &block)
 XML.new(out).instance_eval(&block)
 end
end
```

8.12.2 Validated XML Output with Method Generation

The XML class of Example 11 is helpful for generating well-formed XML, but it does no error checking to ensure that the output is valid according to any particular XML grammar. In the next example, Example 12, we add some simple error checking (though not nearly enough to ensure complete validity—that would require a much longer example). This example is really two DSLs in one. The first is a DSL for defining an XML grammar: a set of tags and the allowed attributes for each tag. You use it like this:

```
class HTMLForm < XMLGrammar
 element :form, : action => REQ,
 : method => "GET",
 : enctype => "application/x-www-form-urlencoded",
 : name => OPT
 element :input, : type => "text", :name => OPT, :value => OPT,
 : maxlength => OPT, :size => OPT, :src => OPT,
 : checked => BOOL, :disabled => BOOL, :readonly => BOOL
 element :textarea, : rows => REQ, :cols => REQ, :name => OPT,
 : disabled => BOOL, :readonly => BOOL
 element :button, :name => OPT, :value => OPT,
 : type => "submit", :disabled => OPT
end
```

This first DSL is defined by the class method XMLGrammar.element. You use it by subclassing XMLGrammar to create a new class. The element method expects the name of a tag as its first argument and a hash of legal attributes as the second argument. The keys of the hash are attribute names. These names may map to default values for the attribute, to the constant REQ for required attributes, or to the constant OPT

for optional attributes. Calling element generates a method with the specified name in the subclass you are defining.

The subclass of XMLGrammar you define is the second DSL, and you can use it to generate XML output that is valid according to the rules you specified. The XMLGrammar class does not have a method_missing method so it won't allow you to use a tag that is not part of the grammar. And the tag method for outputting tags performs error checking on your attributes. Use the generated grammar subclass like the XML class of

Example 11:

```
HTMLForm.generate(STDOUT) do
 comment "This is a simple HTML form"
 form : name => "registration",
 : action => "http://www.example.com/register.cgi" do
 content "Name:"
 input :name => "name"
 content "Address:"
 textarea :name => "address", :rows=>6, :cols=>40 do
 "Please enter your mailing address here"
 end
 button { "Submit" }
 end
end
```

Example 12 shows the implementation of the XMLGrammar class.

Example 12. A DSL for validated XML output

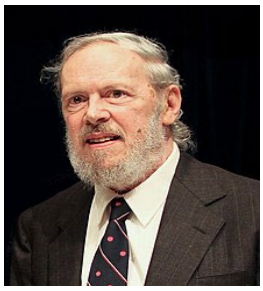
```
class XMLGrammar
 # Create an instance of this class, specifying a stream or object to
 # hold the output. This can be any object that responds to <<(String).
 def initialize(out)
 @out = out # Remember where to send our output
 End

# Invoke the block in an instance that outputs to the specified stream.
```

```
 def self.generate(out, &block)
 new(out).instance_eval(&block)
 end

# Define an allowed element (or tag) in the grammar.
 # This class method is the grammar-specification DSL
 # and defines the methods that constitute the XML-output DSL.
 def self.element(tagname, attributes={})
 @allowed_attributes ||= {}
 @allowed_attributes[tagname] = attributes

class_eval %Q{
 def #{tagname}(attributes={}, &block)
 tag(:#{tagname},attributes,&block)
 end
 }
 End

# These are constants used when defining attribute values.
 OPT =: opt # for optional attributes
 REQ =: req # for required attributes
 BOOL =: bool # for attributes whose value is their own name
 def self.allowed_attributes
 @allowed_attributes
 end

# Output the specified object as CDATA, return nil.
 def content(text)
 @out << text.to_s
 nil
 end

# Output the specified object as a comment, return nil.
```

```
 def comment(text)
 @out << "<!-- #{text} -->"
 nil
 end

# Output a tag with the specified name and attribute.
 # If there is a block, invoke it to output or return content.
 # Return nil.
 def tag(tagname, attributes={})
 # Output the tag name
 @out << "<#{tagname}"

# Get the allowed attributes for this tag.
 allowed = self.class.allowed_attributes[tagname]
 # First, make sure that each of the attributes is allowed.
 # Assuming they are allowed, output all of the specified ones.
 attributes.each_pair do |key,value|
 raise "unknown attribute: #{key}" unless allowed.include?(key)
 @out << " #{key}='#{value}'"
 end

# Now look through the allowed attributes, checking for
 # required attributes that were omitted and for attributes with
 # default values that we can output.
 allowed.each_pair do |key,value|
 # If this attribute was already output, do nothing.
 next if attributes.has_key? key
 if (value == REQ)
 raise "required attribute '#{key}' missing in <#{tagname}>"
 elsif value.is_a? String
 @out << " #{key}='#{value}'"
 end
 end
```

```
if block_given?
 # This block has content
 @out << '>' # End the opening tag
 content = yield # Invoke the block to output or return content
 if content # If any content returned
 @out << content.to_s # Output it as a string
 end
 @out << "</#{tagname}>" # Close the tag
 else
 # Otherwise, this is an empty tag, so just close it.
 @out << '/>'
 end
nil # Tags output themselves, so they don't return any content.
 end
end
```

ROLE MODEL



DENNIS RITCHIE: AMERICAN COMPUTER SCIENTIST

Dennis MacAlistair Ritchie was an American computer scientist. He created the C programming language and, with long-time colleague Ken Thompson, the Unix operating system and B programming language. Ritchie and Thompson were awarded the Turing Award from the ACM in 1983, the Hamming Medal from the IEEE in 1990 and the National Medal of Technology from President Bill Clinton in 1999. Ritchie was the head of Lucent Technologies System Software Research Department when he retired in 2007. He was the "R" in K&R C, and commonly known by his username dmr.

Personal life and career

Dennis Ritchie was born in Bronxville, New York. His father was Alistair E. Ritchie, a longtime Bell Labs scientist and co-author of The Design of Switching Circuits on switching circuit theory. As a child, Dennis moved with his family to Summit, New Jersey, where he graduated from Summit High School.

In 1967, Ritchie began working at the Bell Labs Computing Sciences Research Center, and in 1968, he defended his PhD thesis on "Computational Complexity and Program Structure" at Harvard under the supervision of Patrick C. Fischer. However, Ritchie never officially received his PhD degree as he did not submit a bound copy of his dissertation to the Harvard library, a requirement for the degree. In 2020, the Computer History museum worked with Ritchie's family and Fischer's family and found a copy of the lost dissertation.

During the 1960s, Ritchie and Ken Thompson worked on the Multics operating system at Bell Labs. Thompson then found an old PDP-7 machine and developed his own application programs and operating system from scratch, aided by Ritchie and others. In 1970, Brian Kernighan suggested the name "Unix", a pun on the name "Multics". To supplement assembly language with a system-level programming language, Thompson created B. Later, B was replaced by C, created by Ritchie, who continued to contribute to the development of

Unix and C for many years. During the 1970s, Ritchie collaborated with James Reeds and Robert Morris on a ciphertext-only attack on the M-209 US cipher machine that could solve messages of at least 2000–2500 letters. Ritchie relates that, after discussions with the National Security Agency, the authors decided not to publish it, as they were told that the principle was applicable to machines still in use by foreign governments.

Ritchie was also involved with the development of the Plan 9 and Inferno operating systems, and the programming language Limbo.

As part of an AT&T restructuring in the mid-1990s, Ritchie was transferred to Lucent Technologies, where he retired in 2007 as head of System Software Research Department.

C and Unix

Ritchie is best known as the creator of the C programming language, a key developer of the Unix operating system, and co-author of the book The C Programming Language; he was the 'R' in K&R (a common reference to the book's authors Kernighan and Ritchie). Ritchie worked together with Ken Thompson, who is credited with writing the original version of Unix; one of Ritchie's most important contributions to Unix was its porting to different machines and platforms. They were so influential on Research Unix that Doug McIlroy later wrote, "The names of Ritchie and Thompson may safely be assumed to be attached to almost everything not otherwise attributed."

Ritchie liked to emphasize that he was just one member of a group. He suggested that many of the improvements he introduced simply "looked like a good thing to do," and that anyone else in the same place at the same time might have done the same thing.

Nowadays, the C language is widely used in application, operating system, and embedded system development, and its influence is seen in most modern programming languages. C fundamentally changed the way computer programs were written. For the first time C enabled the same program to work on different machines. Much modern software[which?] is written using one of C's more evolved dialects.[citation needed] Apple has used Objective-C in macOS (derived from NeXTSTEP) and Microsoft uses C#, and Java is used by Android. Ritchie and Thompson used C to write UNIX. Unix has been influential establishing computing concepts and principles that have been widely adopted.

In an interview from 1999, Ritchie clarified that he saw Linux and BSD operating systems as a continuation of the basis of the Unix operating system, and as derivatives of Unix:

I think the Linux phenomenon is quite delightful, because it draws so strongly on the basis that Unix provided. Linux seems to be among the healthiest of the direct Unix derivatives, though there are also the various BSD systems as well as the more

official offerings from the workstation and mainframe manufacturers.

In the same interview, he stated that he viewed both Unix and Linux as "the continuation of ideas that were started by Ken and me and many others, many years ago."

Awards

In 1983, Ritchie and Thompson received the Turing Award "for their development of generic operating systems theory and specifically for the implementation of the UNIX operating system". Ritchie's Turing Award lecture was titled "Reflections on Software Research". In 1990, both Ritchie and Thompson received the IEEE Richard W. Hamming Medal from the Institute of Electrical and Electronics Engineers (IEEE), "for the origination of the UNIX operating system and the C programming language".

In 1997, both Ritchie and Thompson were made Fellows of the Computer History Museum, "for co-creation of the UNIX operating system, and for development of the C programming language."

On April 21, 1999, Thompson and Ritchie jointly received the National Medal of Technology of 1998 from President Bill Clinton for co-inventing the UNIX operating system and the C programming language which, according to the citation for the medal, "led to enormous advances in computer hardware, software, and networking systems and stimulated growth of an entire industry, thereby enhancing American leadership in the Information Age".

In 2005, the Industrial Research Institute awarded Ritchie its Achievement Award in recognition of his contribution to science and technology, and to society generally, with his development of the Unix operating system.

In 2011, Ritchie, along with Thompson, was awarded the Japan Prize for Information and Communications for his work in the development of the Unix operating system.

Death

Ritchie was found dead on October 12, 2011, at the age of 70 at his home in Berkeley Heights, New Jersey, where he lived alone. First news of his death came from his former colleague, Rob Pike. He had been in frail health for several years following treatment for prostate cancer and heart disease. News of Ritchie's death was largely overshadowed by the media coverage of the death of Apple co-founder Steve Jobs, which occurred the week before.

Legacy

Following Ritchie's death, computer historian Paul E. Ceruzzi stated:

Ritchie was under the radar. His name was not a household name at all, but... if you had a microscope and could look in a computer, you'd see his work everywhere inside.

In an interview shortly after Ritchie's death, long time colleague Brian Kernighan said Ritchie never expected C to be so significant. Kernighan told The New York Times "The tools that Dennis built—and their direct descendants—run pretty much everything today." Kernighan reminded readers of how important a role C and Unix had played in the development of later high-profile projects, such as the iPhone. Other testimonials to his influence followed.

Reflecting upon his death, a commentator compared the relative importance of Steve Jobs and Ritchie, concluding that "[Ritchie's] work played a key role in spawning the technological revolution of the last forty years—including technology on which Apple went on to build its fortune." Another commentator said, "Ritchie, on the other hand, invented and co-invented two key software technologies which make up the DNA of effectively every single computer software product we use directly or even indirectly in the modern age. It sounds like a wild claim, but it really is true." Another said, "many in computer science and related fields knew of Ritchie's importance to the growth and development of, well, everything to do with computing,..."

The Fedora 16 Linux distribution, which was released about a month after he died, was dedicated to his memory. FreeBSD 9.0, released January 12, 2012 was also dedicated in his memory.

Asteroid 294727 Dennisritchie, discovered by astronomers Tom Glinos and David H. Levy in 2008, was named in his memory. The official naming citation was published by the Minor Planet Center on 7 February 2012 (M.P.C. 78272).

SUMMARY

- Ruby is a very dynamic language; you can insert new methods into classes at runtime, create aliases for existing methods, and even define methods on individual objects.
- Ruby's reflection API—along with its generally dynamic nature, its blocks-and-iterators control structures, and its parentheses-optional syntax—makes it an ideal language for metaprogramming.
- Metaprogramming is closely tied to the idea of writing domain specific languages, or DSLs. DSLs in Ruby typically use method invocations and blocks as if they were keywords in a task-specific extension to the language.
- A method related to the private include method is the public Object.extend. This method extends an object by making the instance methods of each of the specified modules into singleton methods of the object.
- A Binding object represents the state of Ruby's variable bindings at some moment. The Kernel.binding object returns the bindings in effect at the location of the call.
- The Object class defines a method named instance_eval, and the Module class defines a method named class_eval. (module_eval is a synonym for class_eval.)
- The Object and Module classes define a number of methods for listing, querying, invoking, and defining methods.
- Ruby defines a number of features for tracing the execution of a program. These are mainly useful for debugging code and printing informative error messages.
- The ObjectSpace module defines a handful of low-level methods that can be occasionally useful for debugging or metaprogramming.
- Ruby's use of blocks, coupled with its parentheses-optional syntax, make it very easy to define iterator methods that look like and behave like control structures.
- Ruby's use of blocks, coupled with its parentheses-optional syntax, make it very easy to define iterator methods that look like and behave like control structures.
- The method_missing method is a key part of Ruby's method lookup algorithm and provides a powerful way to catch and handle arbitrary invocations on an object.
- One important metaprogramming technique is the use of methods that create methods. The attr_reader and attr_accessor methods are examples.

KNOWLEDGE CHECK

1. **What is a way of passing arguments into a method that pairs a key that functions as the argument name, with its value?**
   a. Arguments
   b. Mass Assignment
   c. Keyword Arguments
   d. Keyword Parameters
2. **Which of the following is supports by Ruby?**
   a. Multiple programming paradigms
   b. Dynamics type of system
   c. Automatic memory management
   d. All of the above
3. **It is must for Ruby to use a compiler.**
   a. True
   b. False
4. **Which of the following statement is not a feature of ruby?**
   a. Ruby cannot be connected to Database
   b. Ruby is interpreted programming language
   c. Ruby can be embedded into HTML.
   d. None of the above
5. **Which of the following is not a valid datatype in Ruby?**
   a. Integer
   b. String
   c. Timedate
   d. Float

REVIEW QUESTIONS

1. Which methods are used to determine the type of an object? Explain.
2. How to evaluate strings and blocks? Describe.
3. Discuss about thread safety with synchronized blocks.
4. Focus on tracing method invocations with method_missing.
5. Describe chaining methods for thread safety.

Check Your Result

1. (c) 2. (d) 3. (b) 4. (a) 5. (c)

REFERENCES

1. Anonymous. (2018). Ruby ProgrammingLanguage Tutorials Point. Available at https://store.tutorialspoint.com
2. Anonymous. (2019). How Ruby Interprets and Runs Your Programs. Available at https://www.honeybadger.io/. Last access 18/04/2019.
3. Anonymous. (2019). Learn Ruby Programming Language. Available at https://www.tutorialspoint.com/ruby/ last access 20/04/2019.
4. Anonymous. (2019). Ruby programming tutorial. Available at https://ruby-lang.co/what-are-the-disadvantages-of-ruby/. Last access 18/04/2019
5. B. Daloze, C. Seaton, D. Bonetta, and H. Mossenb ¨ ock. Techniques ¨ and applications for guest-language safepoints. In Proceedings of the 10th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages and Tools, 2015.
6. C. Humer, C. Wimmer, C. Wirth, A. Woß, and T. W ¨ urthinger. ¨ A domain-specific language for building self-optimizing AST interpreters. In Proceedings of the International Conference on Generative Programming: Concepts and Experiences, 2014.
7. David Flanagan and Yukihiro Matsumoto.(2008).The Ruby Programming Language. First edition
8. G. Duboscq, T. Wurthinger, L. Stadler, C. Wimmer, D. Simon, and ¨ H. Mossenb ¨ ock. An intermediate representation for speculative ¨ optimizations in a dynamic compiler. In VMIL '13: Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages, 2013.
9. Huw Collingbourne. (2008). The Little Book of Rubysecond edition.
10. Marr, C. Seaton, and S. Ducasse. Zero-overhead metaprogramming: Reflection and metaobject protocols fast and without compromises. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015.
11. Robert W. Sebesta. (2012). Concepts of Programming Languages 11th Edition. University of Colorado at Colorado Springs.
12. Seaton, M. L. Van De Vanter, and M. Haupt. Debugging at Full Speed. In Proceedings of the 8th Workshop on Dynamic Languages and Applications (DYLA), 2014.
13. Seaton. Specialising Dynamic Techniques for Implementing The Ruby Programming Language. PhD thesis, The University of Manchester, 2015.
14. T. Wurthinger, A. W ¨ oß, L. Stadler, G. Duboscq, D. Simon, and ¨ C. Wimmer. Self-optimizing AST interpreters. In Proceedings of the 8th Symposium on Dynamic languages, 2013b.

15. T. Wurthinger, C. Wimmer, A. W ¨ oß, L. Stadler, G. Duboscq, ¨ C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In Onward! '13: Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software, 2013a.

16. Woß, C. Wirth, D. Bonetta, C. Seaton, C. Humer, and ¨ H. Mossenb ¨ ock. An object storage model for the Truffle lan- ¨ guage implementation framework. In Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools, 2014.

CHAPTER

9

METHODS, PRCS, LAMBDAS, AND CLOSURE

"The only way to learn a new programming language is by writing programs in it."

–Dennis Ritchie

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Define simple methods
2. Describe method names and how it is used
3. Discuss on methods and parentheses
4. Explain the concept of method arguments
5. Describe how to create proc objects in both proc and lambda forms
6. Define closures and shared variables
7. Deal with unbound method objects
8. Explain the concept of functional programming

```
Sihui ~ irb
2.3.0 :001 > apple = 'a green apple'
 => "a green apple"
2.3.0 :002 > apple
 => "a green apple"
2.3.0 :003 > def print_apple
2.3.0 :004?>   apple
2.3.0 :005?>   end
 => :print_apple
2.3.0 :006 > print_apple
NameError: undefined local variable or method `apple' for main:Object
        from (irb):4:in `print_apple'
        from (irb):6
        from /Users/Sihui/.rvm/rubies/ruby-2.3.0/bin/irb:11:in `<main>'
2.3.0 :007 > def print_apple
2.3.0 :008?>   apple = 'an red apple'
2.3.0 :009?>   end
 => :print_apple
2.3.0 :010 > print_apple
 => "an red apple"
2.3.0 :011 > apple
 => "a green apple"
```

INTRODUCTION

A method is a named block of parameterized code associated with one or more objects. A method invocation specifies the method name, the object on which it is to be invoked

(sometimes called the receiver), and zero or more argument values that are assigned to the named method parameters. The value of the last expression evaluated in the method becomes the value of the method invocation expression.

Many languages distinguish between functions, which have no associated object, and methods, which are invoked on a receiver object. Because Ruby is a purely object-oriented language, all methods are true methods and are associated with at least one object. We have not covered class definitions in Ruby yet, so the example methods defined look like global functions with no associated object. In fact, Ruby implicitly defines and invokes them as private methods of the Object class.

```
1
2
3   squared = lambda {|n| puts n*n }
4
5   nums = [1, 2, 3, 4, 5, 6]
6   nums.each(&squared)
7
8   # You might have also seen a lambda declared with a ->
9   # n_times_n = ->(n) { puts n * n }
10
11  # both produce:
12  # => 1, 4, 9, 16, 25, 36
13
```

Methods are a fundamental part of Ruby's syntax, but they are not values that Ruby programs can operate on. That is, Ruby's methods are not objects in the way that strings, numbers, and arrays are. It is possible, however, to obtain a Method object that represents a given method, and we can invoke methods indirectly through Method objects.

Methods are not Ruby's only form of parameterized executable code. Blocks, which we introduced, are executable chunks of code and may have parameters. Unlike methods, blocks do not have names, and they can only be invoked indirectly through an iterator method.

Blocks, like methods, are not objects that Ruby can manipulate. But it's possible to create an object that represents a block, and this is actually done with some frequency in Ruby programs. A Proc object represents a block. Like a Method object, we can execute the code of a block through the Proc that represents it. There are two varieties of Proc objects, called procs and lambdas, which have slightly different behavior. Both procs and lambdas are functions rather than methods invoked on an object. An important feature of procs and lambdas is that they are closures: they retain access to the local variables that were in scope when they were defined, even when the proc or lambda is invoked from a different scope.

9.1 DEFINING SIMPLE METHODS

You have seen many method invocations in examples throughout this book, and method invocation syntax was described. Now we turn to the syntax for defining methods. We explain more advanced material and are relevant to both method definition and method invocation.

Keyword

Method invocation refers to how a method is called in a program. The process of invoking a method in Ruby is quite easy since the use of parenthesis is optional

Methods are defined with the def keyword. This is followed by the method name and an optional list of parameter names in parentheses. The Ruby code that constitutes the method body follows the parameter list, and the end of the method is marked with the end keyword. Parameter names can be used as variables within the method body, and the values of these named parameters come from the arguments to a **method invocation**.

```
# if the first argument begins with a parentheses
# then use parentheses in method invocation
# this is a common source of consternation.
#
method ( 3 + 2 ) % 2 # ambiguous

method(( 3 + 2 ) % 2) # unambiguous

(method( 3 + 2 ) % 2) # unambiguous
```

Here is an example method:

```
# Define a method named 'factorial' with a single parameter 'n'
def factorial(n)
 if n < 1                    # Test the argument value for validity
  raise "argument must be > 0"
 elsif n == 1   # If the argument is 1
1               # then the value of the method invocation is 1
else            # Otherwise, the factorial of n is n times
  n * factorial(n-1)        # the factorial of n-1
 end
```

end

This code defines a method named factorial. The method has a single parameter named n. The identifier n is used as a variable within the body of the method. This is a recursive method, so the body of the method includes an invocation of the method. The invocation is simply the name of the method followed by the argument value in parentheses.

9.1.1 Method Return Value

Methods may terminate normally or abnormally. Abnormal termination occurs when the method raises an exception. The factorial method shown earlier terminates abnormally if we pass it an argument less than 1. If a method terminates normally, then the value of the method invocation expression is the value of the last expression evaluated within the method body. In the factorial method, that last expression will either be 1 or n*factorial(n-1).

The return keyword is used to force a return prior to the end of the method. If an expression follows the return keyword, then the value of that expression is returned. If no expression follows, then the return value is nil. In the following variant of the factorial method, the return keyword is required:

```
def factorial(n)
 raise “bad argument” if n < 1
return 1 if n == 1
 n * factorial(n-1)
end
```

We could also use return on the last line of this method body to emphasize that this expression is the method's return value. In common practice, however, return is omitted where it is not required.

Ruby methods may return more than one value. To do this, use an explicit return statement, and separate the values to be returned with commas:

```
# Convert the Cartesian point (x,y) to polar (magnitude, angle) coordinates
def polar(x,y)
 return Math.hypot(y,x), Math.atan2(y,x)
end
```

When there is more than one return value, the values are collected into an array, and the array becomes the single return value of the method. Instead of using the return statement with multiple values, we can simply create an array of values ourselves:

```
# Convert polar coordinates to Cartesian coordinates
def cartesian(magnitude, angle)
 [magnitude*Math.cos(angle), magnitude*Math.sin(angle)]
end
```

Methods of this form are typically intended for use with parallel assignment so that each return value is assigned to a separate variable:

```
distance, theta = polar(x,y)
x,y = cartesian(distance,theta)
```

9.1.2 Methods and Exception Handling

A def statement that defines a method may include exception-handling code in the form of rescue, else, and ensure clauses, just as a begin statement can. These **exception handling** clauses go after the end of the method body but before the end of the def statement. In short methods, it can be particularly tidy to associate your rescue clauses with the def statement. This also means you don't have to use a begin statement and the extra level of indentation that comes with it.

```
>> require 'benchmark/ips'
=> true
>> Benchmark.ips do |x|
 |   x.report('rescue nil') { Integer('Square') rescue nil }
 |   x.report('exception: false') { Integer('Square', exception: false) }
 |   x.compare!
 | end
Warming up --------------------------------------
          rescue nil    58.148k i/100ms
    exception: false   133.846k i/100ms
Calculating -------------------------------------
          rescue nil    705.680k (± 2.3%) i/s -      3.547M in   5.029110s
    exception: false      1.880M (± 2.4%) i/s -      9.503M in   5.057631s

Comparison:
    exception: false:  1880128.1 i/s
          rescue nil:   705679.8 i/s - 2.66x  slower
```

Keyword

Exception handling is the process of responding to the occurrence of exceptions – anomalous or exceptional conditions requiring special processing – during the execution of a program.

9.1.3 Invoking a Method on an Object

Methods are always invoked on an object. (This object is sometimes called the receiver in a reference to an object-oriented paradigm in which methods are called "messages" and are "sent to" receiver objects.) Within the body of a method, the keyword self refers to the object on which the method was invoked. If we do not specify an object when invoking a method, then the method is implicitly invoked on self.

Notice, however, that you have already seen examples of invoking methods on objects, in code like this:

Like most object-oriented languages, Ruby uses. to separate the object from the method to be invoked on it. This code passes the value of the variable pattern to the method named index of the object stored in the variable text, and stores the return value in the variable first.

9.1.4 Defining Singleton Methods

The methods we have defined so far are all global methods. If we place a def statement like the ones shown earlier inside a class statement, then the methods that are defined are instance methods of the class; these methods are defined on all objects that are instances of the class. It is also possible, however, to use the def statement to define a method on a single specified object. Simply follow the def keyword with an expression that evaluates to an object. This expression should be followed by a period and the name of the method to be defined. The resulting method is known as a singleton method because it is available only on a single object:

```
o = “message” # A string is an object
def o.printme # Define a singleton method for this object
 puts self
end
o.printme # Invoke the singleton
```

Class methods such as Math.sin and File.delete are actually singleton methods. Math is a constant that refers to a Module object, and File is a constant that refers to a Class object. These two objects have singleton methods named sin and delete, respectively.

Ruby implementations typically treat Fixnum and Symbol values as immediate values rather than as true object references For this reason, singleton methods may not be defined on Fixnum and Symbol objects. For consistency, singletons are also prohibited on other Numeric objects.

9.1.5 Undefining Methods

Methods are defined with the def statement and may be undefined with the undef statement:

```
def sum(x,y); x+y; end      # Define a method
puts sum(1,2)               # Use it
undef sum                   # And undefine it
```

In this code, the def statement defines a global method, and undef undefines it. undef also works within classes to undefine the instance methods of the class. Interestingly, undef can be used to undefine inherited methods, without affecting the definition of the method in the class from which it is inherited. Suppose class A defines a method m, and class B is a subclass of A and therefore inherits m. If you do not want to allow instances of class B to be able to invoke m, you can use undef m within the body of the subclass.

undef is not a commonly used statement. In practice, it is much more common to redefine a method with a new def statement than it is to undefine or delete the method.

Note that the undef statement must be followed by a single identifier that specifies the method name. It cannot be used to undefine a singleton method in the way that def can be used to define such a method.

Within a class or module, you can also use undef_method (a private method of Module) to undefine methods. Pass a symbol representing the name of the method to be undefined.

9.2 METHOD NAMES

Keyword

Numerical method is a mathematical tool designed to solve numerical problems. The implementation of a numerical method with an appropriate convergence check in a programming language is called a numerical algorithm.

By convention, method names begin with a lowercase letter. (Method names can begin with a capital letter, but that makes them look like constants.) When a method name is longer than one word, the usual convention is to separate the words with underscore like_this rather than using mixed case likeThis.

Method names may (but are not required to) end with an equals sign, a question mark, or an exclamation point. An equals sign suffix signifies that the method is a setter that can be invoked using assignment syntax. The question mark and exclamation point suffixes have no special meaning to the Ruby interpreter, but they are allowed because they enable two extraordinarily useful naming conventions.

The first convention is that any method whose name ends with a question mark returns a value that answers the question posed by the method invocation. The empty? method of an array, for example, returns true if the array has no elements. Methods like these are called predicates and. Predicates typically return one of the Boolean values true or false, but this is not required, as any value other than false or nil works like true when a Boolean value is required. (The **Numeric method** nonzero?, for example, returns nil if the number it is invoked on is zero, and just returns the number otherwise.)

The second convention is that any method whose name ends with an exclamation mark should be used with caution.

example

The array object has a sort method that makes a copy of the array, and then sorts that copy. It also has a sort! method that sorts the array in place. The exclamation mark indicates that you need to be more careful when using that version of the method.

Often, methods that end with an exclamation mark are mutators, which alter the internal state of an object. But this is not always the case; there are many mutators that do not end

with an exclamation mark, and a number of nonmutators that do. Mutating methods (such as Array.fill) that do not have a nonmutating variant do not typically have an exclamation point.

Consider the global function exit: it makes the Ruby program stop running in a controlled way. There is also a variant named exit! that aborts the program immediately without running any END blocks or shutdown hooks registered with at_exit. exit! isn't a mutator; it's the "dangerous" variant of the exit method and is flagged with ! to remind a programmer using it to be careful.

9.2.1 Operator Methods

Many of Ruby's operators, such as +, *, and even the array index operator [], are implemented with methods that you can define in your own classes. You define an operator by defining a method with the same "name" as the operator. (The only exceptions are the unary plus and minus operators, which use method names +@ and -@.) Ruby allows you to do this even though the method name is all punctuation. You might end up with a method definition like this:

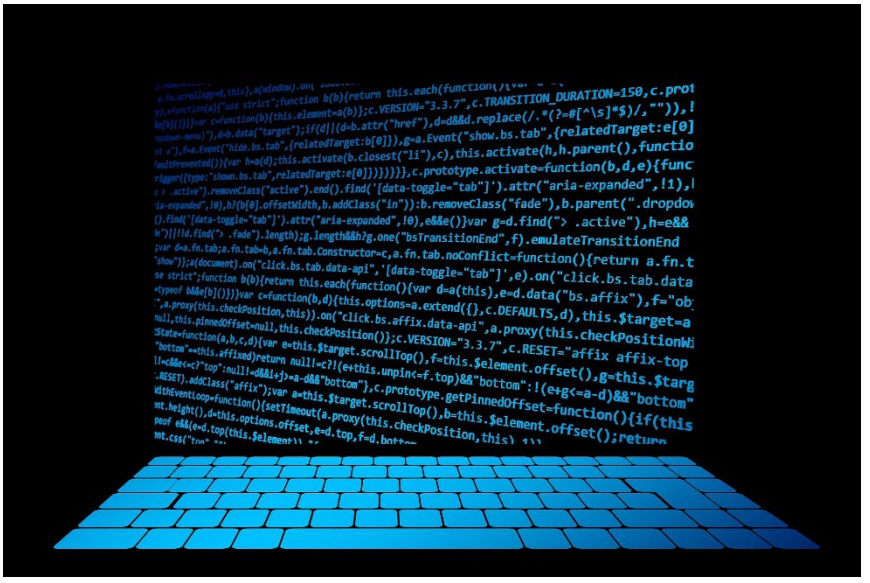
```
def +(other)                    # Define binary plus operator: x+y is x.+(y)
 self.concatenate(other)
end
```

Methods that define a unary operator are passed no arguments. Methods that define binary operators are passed one argument and should operate on self and the argument. The array access operators [] and []= are special because they can be invoked with any number of arguments. For []=, the last argument is always the value being assigned.

9.2.2 Method Aliases

It is not uncommon for methods in Ruby to have more than one name. The language has a keyword alias that serves to define a new name for an existing method. Use it like this:

```
alias aka also_known_as # alias new_name existing_name
```



After executing this statement, the identifier aka will refer to the same method thats also_known_as does.

Method aliasing is one of the things that makes Ruby an expressive and natural language. When there are multiple names for a method, you can choose the one that seems most natural in your code. The Range class, for example, defines a method for testing whether a value falls within the range. You can call this method with the name include? or with the name member?. If you are treating a range as a kind of set, the name member? may be the most natural choice.

A more practical reason for aliasing methods is to insert new functionality into a method. The following is a common idiom for augmenting existing methods:

```
def hello                          # A nice simple method
 puts “Hello World”                # Suppose we want to augment it...
end
alias original_hello hello         # Give the method a backup name
def hello                          # Now we define a new method with the
old name
 puts “Your attention please”      # That does some stuff
 original_hello                    # Then calls the original method
 puts “This has been a test”       # Then does some more stuff
end
```

In this code, we are working on global methods. It is more common to use alias with the instance methods of a class. In this situation, alias must be used within the class whose method is to be renamed. Classes in Ruby can be "reopened"—which means that your code can take an existing class, 'open' it with a class statement, and then use alias as shown in the example to augment or alter the existing methods of that class.

9.3 METHODS AND PARENTHESES

Ruby allows parentheses to be omitted from most method invocations. In simple cases, this results in clean-looking code. In complex cases, however, it causes syntactic ambiguities and confusing corner cases. We will \ consider these as follow.

9.3.1 Optional Parentheses

Parentheses are omitted from method invocations in many common Ruby idioms. The following two lines of code, for example, are equivalent:

```
puts "Hello World"
puts("Hello World")
```

In the first line, puts looks like a keyword, statement, or command built in to the language. The equivalent second line demonstrates that it is simply the invocation of a global method, with the parentheses omitted. Although the second form is clearer, the first form is more concise, more commonly used, and arguably more natural.

Next, consider this code:

```
greeting = "Hello"
size = greeting.length
```

If you are accustomed to other object-oriented languages, you may think that length is a property, field, or variable of string objects. Ruby is strongly object oriented, however, and its objects are fully encapsulated; the only way to interact with them is by invoking their methods. In this code, greeting.length is a method invocation. The length method expects no arguments and is invoked without parentheses. The following code is equivalent:

```
size = greeting.length()
```

Including the optional parentheses emphasizes that a method invocation is occurring. Omitting the parentheses in method invocations with no arguments gives the illusion of property access, and is a very common practice.

Parentheses are very commonly omitted when there are zero or one arguments to the invoked method. Although it is less common, the parentheses may be omitted even when there are multiple arguments, as in the following code:

```
x = 3 # x is a number
x.between? 1,5 # same as x.between?(1,5)
```

Parentheses may also be omitted around the parameter list in method definitions, though it is hard to argue that this makes your code clearer or more readable. The following code, for example, defines a method that returns the sum of its arguments:

```
def sum x, y
 x+y
end
```

9.3.2 Required Parentheses

Some code is ambiguous if the parentheses are omitted, and here Ruby requires that you include them. The most common case is nested method invocations of the form f g x, y. In Ruby, invocations of that form mean f(g(x,y)). Ruby 1.8 issues a warning, however, because the code could also be interpreted as f(g(x),y). The warning has been removed in Ruby 1.9. The following code, using the sum method defined above, prints 4, but issues a warning in Ruby 1.8:

```
puts sum 2, 2
```

To remove the warning, rewrite the code with parentheses around the arguments:

```
puts sum(2,2)
```

Note that using parentheses around the outer method invocation does not resolve the ambiguity:

```
puts(sum 2,2) # Does this mean puts(sum(2,2)) or puts(sum(2), 2)?
```

An expression involving nested function calls is only ambiguous when there is more than one argument. The Ruby interpreter can only interpret the following code in one way:

```
puts factorial x # This can only mean puts(factorial(x))
```

Despite the lack of ambiguity here, Ruby 1.8 still issues a warning if you omit the parentheses around the x.

Sometimes omitting parentheses is a true syntax error rather than a simple warning. The following expressions, for example, are completely ambiguous without parentheses, and Ruby doesn't even attempt to guess what you mean:

```
puts 4, sum 2,2        # Error: does the second comma go with the 1st or 2nd method?
[sum 2,2]              # Error: two array elements or one?
```

There is another wrinkle that arises from the fact that parentheses are optional. When you do use parentheses in a method invocation, the opening **parenthesis** must immediately follow the method name, with no intervening space. This is because parentheses do double-duty: they can be used around an argument list in a method invocation, and they can be used for grouping expressions.

Keyword

Parenthesis or parenthetical phrase is an explanatory or qualifying word, clause, or sentence inserted into a passage.

Consider the following two expressions, which differ only by a single space:

```
square(2+2)*2 # square(4)*2 = 16*2 = 32
square (2+2)*2 # square(4*2) = square(8) = 64
```

In the first expression, the parentheses represent method invocation. In the second, they represent expression grouping. To reduce the potential for confusion, you should always use parentheses around a method invocation if any of the arguments use parentheses. The second expression would be written more clearly as:

```
square((2+2)*2)
```

We'll end this discussion of parentheses with one final twist. Recall that the following expression is ambiguous and causes a warning:

```
puts(sum 2,2) # Does this mean puts(sum(2,2)) or puts(sum(2), 2)?
```

The best way to resolve this ambiguity is to put parentheses around the arguments to the sum method. Another way is to add a space between puts and the opening parenthesis:

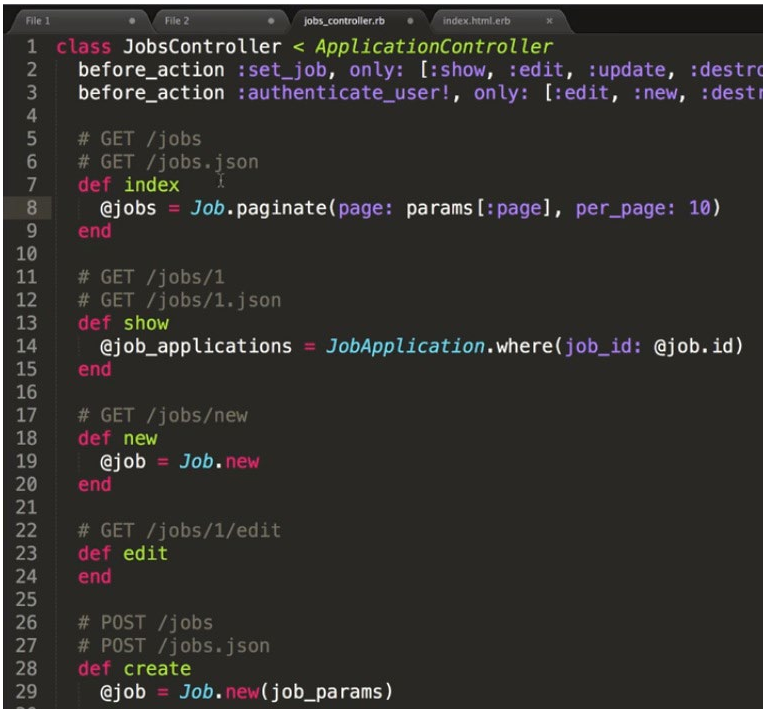
```
puts (sum 2,2)
```

Adding the space converts the method invocation parentheses into expression grouping parentheses. Because these parentheses group a subexpression, the comma can no longer be interpreted as an argument delimiter for the puts invocation.

9.4 METHOD ARGUMENTS

Simple method declarations include a comma-separated list of argument names (in optional parentheses) after the method name. But there is much more to Ruby's method arguments.

- How to declare an argument that has a default value, so that the argument can be omitted when the method is invoked
- How to declare a method that accepts any number of arguments
- How to simulate named method arguments with special syntax for passing a hash to a method
- How to declare a method so that the block associated with an invocation of the method is treated as a method argument



9.4.1 Parameter Defaults

When you define a method, you can specify default values for some or all of the parameters. If you do this, then your method may be invoked with fewer argument values than the declared number of parameters. If arguments are omitted, then the default value of the parameter is used in its place. Specify a default value by following the parameter name with an equals sign and a value:

```
def prefix(s, len=1)
 s[0,len]
end
```

This method declares two parameters, but the second one has a default. This means that we can invoke it with either one argument or two:

```
prefix("Ruby", 3) # => "Rub"
prefix("Ruby") # => "R"
```

Argument defaults need not be constants: they may be arbitrary expressions, and can be referred to instance variables and to previous parameters in the parameter list. For example:

```
# Return the last character of s or the substring from index to the end
def suffix(s, index=s.size-1)
 s[index, s.size-index]
end
```

Parameter defaults are evaluated when a method is invoked rather than when it is parsed. In the following method, the default value [] produces a new empty array on each invocation, rather than reusing a single array created when the method is defined:

```
# Append the value x to the array a, return a.
# If no array is specified, start with an empty one.
def append(x, a=[])
 a << x
```

```
end
```

In Ruby 1.8, method parameters with default values must appear after all ordinary parameters in the parameter list. Ruby 1.9 relaxes this restriction and allows ordinary parameters to appear after parameters with defaults. It still requires all parameters with defaults to be adjacent in the parameter list—you can't declare two parameters with default values with an ordinary parameter between them, for example. When a method has more than one parameter with a default value, and you invoke the method with an argument for some, but not all, of these parameters, they are filled in from left to right. Suppose a method has two parameters, and both of those parameters have defaults. You can invoke this method with zero, one, or two arguments. If you specify one argument, it is assigned to the first parameter and the second parameter uses its default value. There is no way, however, to specify a value for the second parameter and use the default value of the first parameter.

9.4.2 Variable-Length Argument Lists and Arrays

Sometimes we want to write methods that can accept an arbitrary number of arguments. To do this, we put an * before one of the method's parameters. Within the body of the method, this parameter will refer to an array that contains the zero or more arguments passed at that position. For example:

```
# Return the largest of the one or more arguments passed
def max(first, *rest)
 # Assume that the required first argument is the largest
 max = first
 # Now loop through each of the optional arguments looking for bigger ones
 rest.each {|x| max = x if x > max }
 # Return the largest one we found
 max
end
```

The max method requires at least one argument, but it may accept any number of additional arguments. The first argument is available through the first parameter. Any additional arguments are stored in the rest array. We can invoke max like this:

```
max(1)          # first=1, rest=[]
max(1,2)        # first=1, rest=[2]
```

```
max(1,2,3)                    # first=1, rest=[2,3]
```

Note that in Ruby, all Enumerable objects automatically have a max method, so the method defined here is not particularly useful.

No more than one parameter may be prefixed with an *. In Ruby 1.8, this parameter must appear after all ordinary parameters and after all parameters with defaults specified. It should be the last parameter of the method, unless the method also has a parameter with an & prefix. In Ruby 1.9, a parameter with an * prefix must still appear after any parameters with defaults specified, but it may be followed by additional ordinary parameters. It must also still appear before any &-prefixed parameter.

Keyword

Ruby arrays are ordered collections of objects. They can hold objects like integer, number, hash, string, symbol or any other array.

Passing arrays to methods

We have seen how * can be used in a method declaration to cause multiple arguments to be gathered or coalesced into a single array. It can also be used in a method invocation to scatter, expand, or explode the elements of an array (or range or enumerator) so that each element becomes a separate method argument. The * is sometimes called the splat operator, although it is not a true operator.

Suppose we wanted to find the maximum value in an array (and that we did not know that **Ruby arrays** have a built-in max method!). We could pass the elements of the array to the max method like this:

```
data = [3, 2, 1]
m = max(*data) # first = 3, rest=[2,1] => 3
```

Consider what happens without the *:

```
m = max(data) # first = [3,2,1], rest=[] => [3,2,1]
```

In this case, we hare passing an array as the first and only argument, and our max method returns that first argument without performing any comparisons on it.

The * can also be used with methods that return arrays to expand those arrays for use in another method invocation. Consider the polar and cartesian methods defined:

```
# Convert the point (x,y) to Polar coordinates, then back to Cartesian
x,y = cartesian(*polar(x, y))
```

In Ruby 1.9, enumerators are splattable objects. To find the largest letter in a string, for example, we could write:

```
max(*"hello world".each_char) # => 'w'
```

9.4.3 Mapping Arguments to Parameters

When a method definition includes parameters with default values or a parameter prefixed with an *, the assignment of argument values to parameters during method invocation gets a little bit tricky.

In Ruby 1.8, the position of the special parameters is restricted so that argument values are assigned to parameters from left to right. The first arguments are assigned to the ordinary parameters. If there are any remaining arguments, they are assigned to the parameters that have defaults. And if there are still more arguments, they are assigned to the array argument.

Ruby 1.9 has to be more clever about the way it maps arguments to parameters because the order of the parameters is no longer constrained. Suppose we have a method that is declared with o ordinary parameters, d parameters with default values, and one array parameter prefixed with *. Now assume that we invoke this method with a arguments.

If a is less than o, an ArgumentError is raised; we have not supplied the minimum required number of arguments.

If a is greater than or equal to o and less than or equal to o+d, then the leftmost a–o parameters with defaults will have arguments assigned to them. The remaining (to the right) o+d–a parameters with defaults will not have arguments assigned to them, and will just use their default values.

If a is greater than o+d, then the array parameter whose name is prefixed with an * will have a–o–d arguments stored in it; otherwise, it will be empty.

Once these calculations are performed, the arguments are mapped to parameters from left to right, assigning the appropriate number of arguments to each parameter.

9.4.4 Hashes for Named Arguments

When a method requires more than two or three arguments, it can be difficult for the programmer invoking the method to remember the proper order for those arguments.

```ruby
# Keyword Arguments: Missing parameter
class Bicycle
  attr_accessor :make, :frame_size, :weight, :color

  def initialize(make: 'Canyon', frame_size:, weight:, color:)
    @make = make
    @frame_size = frame_size
    @weight = weight
    @color = color
  end
end

bicycle = Bicycle.new(
  make: 'Canyon',
  frame_size: '178cm',
  weight: '7,8kg',
)
# keyword_arguments.rb:131:in `initialize': missing keyword: color (ArgumentError)
#         from keyword_arguments.rb:139:in `new'
#         from keyword_arguments.rb:139:in `<main>'
```

Some languages allow you to write method invocations that explicitly specify a parameter name for each argument that is passed. Ruby does not support this method invocation syntax, but it can be approximated if you write a method that expects a hash as its argument or as one of its arguments:

```
# This method returns an array a of n numbers. For any index i, 0 <= i < n,
# the value of element a[i] is m*i+c. Arguments n, m, and c are passed
# as keys in a hash, so that it is not necessary to remember their order.
def sequence(args)
 # Extract the arguments from the hash.
 # Note the use of the || operator to specify defaults used
 # if the hash does not define a key that we are interested in.
 n = args[:n] || 0
 m = args[:m] || 1
 c = args[:c] || 0

a = []                          # Start with an empty array
 n.times {|i| a << m*i+c }      # Calculate the value of each array element
```

```
 a                    # Return the array
end
```

You might invoke this method with a hash literal argument like this:

```
sequence({:n=>3, :m=>5}) # => [0, 5, 10]
```

In order to better support this style of programming, Ruby allows you to omit the curly braces around the hash literal if it is the last argument to the method (or if the only argument that follows it is a block argument, prefixed with &). A hash without braces is sometimes called a bare hash, and when we use one it looks like we are passing separate named arguments, which we can reorder however we like:

```
sequence(:m=>3, :n=>5) # => [0, 3, 6, 9, 12]
```

As with other ruby methods, we can omit the parentheses, too:

```
# Ruby 1.9 hash syntax
sequence c:1, m:3, n:5 # => [1, 4, 7, 10, 13]
```

If you omit the parentheses, then you must omit the curly braces. If curly braces follow the method name outside of parentheses, Ruby thinks you are passing a block to the method:

```
sequence {:m=>3, :n=>5} # Syntax error!
```

9.4.5 Block Arguments

A block is a chunk of Ruby code associated with a method invocation, and that an iterator is a method that expects a block. Any method invocation may be followed by a block, and any method that has a block associated with it may invoke the code in that block with the yield statement. To refresh your memory, the following code is a block-oriented variant on the sequence method:

```
# Generate a sequence of n numbers m*i + c and pass them to the block
def sequence2(n, m, c)
 i = 0
 while(i < n)              # loop n times
```

```
 yield i*m + c              # pass next element of the sequence to the block
 i += 1
 end
end
# Here is how you might use this version of the method
sequence2(5, 2, 2) {|x| puts x } # Print numbers 2, 4, 6, 8, 10
```

One of the features of blocks is their anonymity. They are not passed to the method in a traditional sense, they have no name, and they are invoked with a keyword rather than with a method. If you prefer more explicit control over a block (so that you can pass it on to some other method, for example), add a final argument to your method, and prefix the argument name with an ampersand.* If you do this, then that argument will refer to the block—if any—that is passed to the method. The value of the argument will be a Proc object, and instead of using yield, you invoke the call method of the Proc:

```
def sequence3(n, m, c, &b)          # Explicit argument to get block as a Proc
 i = 0
 while(i < n)
 b.call(i*m + c)                    # Invoke the Proc with its call method
 i += 1
 end
end
# Note that the block is still passed outside of the parentheses
sequence3(5, 2, 2) {|x| puts x }
```

Notice that using the ampersand in this way changes only the method definition. The method invocation remains the same. We end up with the block argument being declared inside the parentheses of the method definition, but the block itself is still specified outside the parentheses of the method invocation.

A special kind of parameter must be the last one in the parameter list. Block arguments prefixed with ampersands must really be the last one. Because blocks are passed unusually in method invocations, named block arguments are different and do not interfere with array or hash parameters in which the brackets and braces have been omitted. The following two methods are legal, for example:

```
def sequence5(args, &b) # Pass arguments as a hash and follow with a block
```

```
 n, m, c = args[:n], args[:m], args[:c]
 i = 0
 while(i < n)
 b.call(i*m + c)
 i += 1
 end
end

# Expects one or more arguments, followed by a block
def max(first, *rest, &block)
 max = first
 rest.each {|x| max = x if x > max }
 block.call(max)
 max
end
```

These methods work fine, but notice that you can avoid the complexity of these cases by simply leaving your blocks anonymous and calling them with yield.

It is also worth noting that the yield statement still works in a method defined with an & parameter. Even if the block has been converted to a Proc object and passed as an argument, it can still be invoked as an anonymous block, as if the block argument was not there.

Using & in method invocation

We saw earlier that you can use * in a method definition to specify that multiple arguments should be packed into an array, and that you can use * in a method invocation to specify that an array should be unpacked so that its elements become separate arguments. & can also be used in definitions and invocations. We've just seen that & in a method definition allows an ordinary block associated with a method invocation to be used as a named Proc object inside the method. When & is used before a Proc object in a method invocation, it treats the Proc as if it was an ordinary block following the invocation.

Consider the following code which sums the contents of two arrays:

```
a, b = [1,2,3], [4,5] # Start with some data.
```

```
sum = a.inject(0) {|total,x| total+x } # => 6. Sum elements of a.
sum = b.inject(sum) {|total,x| total+x } # => 15. Add the elements of b in.
```

If you do not remember, you can look up its documentation with ri Enumerable. inject. The important thing to notice about this example is that the two blocks are identical. Rather than having the Ruby interpreter parse the same block twice, we can create a Proc to represent the block, and use the single Proc object twice:

```
a, b = [1,2,3], [4,5]                    # Start with some data.
summation = Proc.new {|total,x| total+x } # A Proc object for summations.
sum = a.inject(0, &summation)                    # => 6
sum = b.inject(sum, &summation)          # => 15
```

If you use & in a method invocation, it must appear before the last argument in the invocation. Blocks can be associated with any method call, even when the method is not expecting a block, and never uses yield. In the same way, any method invocation may have an & argument as its last argument.

In a method invocation an & typically appears before a Proc object. But it is actually allowed before any object with a to_proc method. The Method class has such a method, so Method objects can be passed to iterators just as Proc objects can.

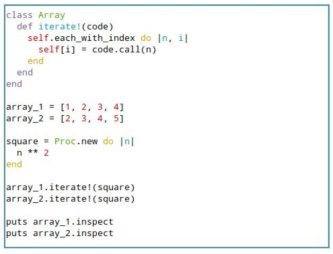
In Ruby 1.9, the Symbol class defines a to_proc method, allowing symbols to be prefixed with & and passed to iterators. When a symbol is passed like this, it is assumed to be the name of a method. The Proc object returned by the to_proc method invokes the named method of its first argument, passing any remaining arguments to that named method. The canonical case is this: given an array of strings, create a new array of those strings, converted to uppercase. Symbol.to_proc allows us to accomplish this elegantly as follows:

```
words = ['and', 'but', 'car']          # An array of words
uppercase = words.map &:upcase         # Convert to uppercase with String.
upcase
upper = words.map {|w| w.upcase }      # This is the equivalent code with a
block
```

9.5 PROCS AND LAMBDAS

Blocks are syntactic structures in Ruby; they are not objects, and cannot be manipulated as objects. It is possible, however, to create an object that represents a block. Depending on how the object is created, it is called a proc or a lambda. Procs have block-like behavior and lambdas have method-like behavior. Both, however, are instances of class Proc.

- How to create Proc objects in both proc and lambda forms
- How to invoke Proc objects
- How to determine how many arguments a Proc expects
- How to determine if two Proc objects are the same
- How procs and lambdas differ from each other



```ruby
class Array
  def iterate!(code)
    self.each_with_index do |n, i|
      self[i] = code.call(n)
    end
  end
end

array_1 = [1, 2, 3, 4]
array_2 = [2, 3, 4, 5]

square = Proc.new do |n|
  n ** 2
end

array_1.iterate!(square)
array_2.iterate!(square)

puts array_1.inspect
puts array_2.inspect
```

```ruby
class Array
  def iterate!(code)
    self.each_with_index do |n, i|
      self[i] = code.call(n)
    end
  end
end

array = [1, 2, 3, 4]

array.iterate!(Proc.new do |n|
  n ** 2
end)

puts array.inspect
```

9.5.1 Creating Procs

We have already seen one way to crfate a Proc object: by associating a block with a method that is defined with an ampersand-prefixed block argument. There is nothing preventing such a method from returning the Proc object for use outside the method:

```
# This method creates a proc from a block
def makeproc(&p) # Convert associated block to a Proc and store in p
```

With a makeproc method like this defined, we can create a Proc object for ourselves:

```
adder = makeproc {|x,y| x+y }
```

The variable adder now refers to a Proc object. Proc objects created in this way are procs, not lambdas. All Proc objects have a call method that, when invoked, runs the code contained by the block from which the proc was created. For example:

```
sum = adder.call(2,2) # => 4
```

In addition to being invoked, Proc objects can be passed to methods, stored in data structures and otherwise manipulated like any other Ruby object.

As well as creating procs by method invocation, there are three methods that create Proc objects (both procs and lambdas) in Ruby. These methods are commonly used, and it is not actually necessary to define a makeproc method like the one shown earlier. In addition to these Proc-creation methods, Ruby 1.9 also supports a new literal syntax for defining lambdas.

Proc.new

This is the normal new method that most classes support, and it's the most obvious way to create a new instance of the Proc class. Proc.new expects no arguments, and returns a Proc object that is a proc (not a lambda). When you invoke Proc. new with an associated block, it returns a proc that represents the block. For example:

Keyword

lambdas allow you to encapsulate logic and data in an eminently portable variable. A lambda function can be passed to object methods, stored in data structures, and executed when needed.

```
p = Proc.new {|x,y| x+y }
```

If Proc.new is invoked without a block from within a method that does have an associated block, then it returns a proc representing the block associated with the containing method. Using Proc.new in this way provides an alternative to using an ampersandprefixed block argument in a method definition. The following two methods are equivalent, for example:

```
def invoke(&b)          def invoke
 b.call Proc.           new.call
end                     end
```

Kernel.lambda

Another technique for creating Proc objects is with the lambda method. **lambda** is a method of the Kernel module, so it

behaves like a global function. As its name suggests, the Proc object returned by this method is a lambda rather than a proc.

lambda expects no arguments, but there must be a block associated with the invocation:

```
is_positive = lambda {|x| x > 0 }
```

Kernel.proc

In Ruby 1.8, the global proc method is a synonym for lambda. Despite its name, it returns a lambda, not a proc. Ruby 1.9 fixes this; in that version of the language, proc is a synonym for Proc.new.

Because of this ambiguity, you should never use proc in Ruby 1.8 code. The behavior of your code might change if the interpreter was upgraded to a newer version. If you are using Ruby 1.9 code and are confident that it will never be run with a Ruby 1.8 interpreter, you can safely use proc as a more elegant shorthand for Proc.new.

Lambda Literals

Ruby 1.9 supports an entirely new syntax for defining lambdas as literals. We will begin with a Ruby 1.8 lambda, created with the lambda method:

```
succ = lambda {|x| x+1}
```

In Ruby 1.9, we can convert this to a literal as follows:

- Replace the method name lambda with the punctuation ->.
- Move the list of arguments outside of and just before the curly braces.
- Change the argument list delimiters from || to ().

With these changes, we get a Ruby 1.9 lambda literal:

```
succ = ->(x){ x+1 }
```

succ now holds a Proc object, which we can use just like any other:

```
succ.call(2) # => 3
```

The introduction of this syntax into Ruby was controversial, and it takes some getting used to. Note that the arrow characters -> are different from those used in hash literals. A lambda literal uses an arrow made with a hyphen, whereas a hash literal uses an arrow made with an equals sign.

As with blocks in Ruby 1.9, the argument list of a lambda literal may include the declaration of block-local variables that are guaranteed not to overwrite variables

with the same name in the enclosing scope. Simply follow the parameter list with a semicolon and a list of local variables:

```
# This lambda takes 2 args and declares 3 local vars
f = ->(x,y; i,j,k) { ... }
```

One benefit of this new lambda syntax over the traditional block-based lambda creation methods is that the Ruby 1.9 syntax allows lambdas to be declared with argument defaults, just as methods can be:

```
zoom = ->(x,y,factor=2) { [x*factor, y*factor] }
```

As with method declarations, the parentheses in lambda literals are optional, because the parameter list and local variable lists are completely delimited by the ->, ;, and {.

We could rewrite the three lambdas above like this:

```
succ = ->x { x+1 }
f = -> x,y; i,j,k { ... }
zoom = ->x,y,factor=2 { [x*factor, y*factor] }
```

Lambda parameters and local variables are optional, of course, and a lambda literal can omit this altogether. The minimal lambda, which takes no arguments and returns nil, is the following:

```
->{}
```

One benefit of this new syntax is its succinctness. It can be helpful when you want to pass a lambda as an argument to a method or to another lambda:

```
def compose(f,g)                          # Compose 2 lambdas
 ->(x) { f.call(g.call(x)) }
end
succOfSquare = compose(->x{x+1}, ->x{x*x})
succOfSquare.call(4)                      # => 17: Computes (4*4)+1
```

Lambda literals create Proc objects and are not the same thing as blocks. If you want to pass a lambda literal to a method that expects a block, prefix the literal with &, just as you would with any other Proc object. Here is how we might sort an array of numbers into descending order using both a block and a lambda literal:

```
data.sort {|a,b| b-a } # The block version
data.sort &->(a,b){ b-a } # The lambda literal version
```

In this case, as you can see, regular block syntax is simpler.

9.5.2 Invoking Procs and Lambdas

Procs and lambdas are objects, not methods, and they cannot be invoked in the same way that methods are. If p refers to a Proc object, you cannot invoke p as a method. But because p is an object, you can invoke a method of p. We have already mentioned that the Proc class defines a method named call. Invoking this method executes the code in the original block. The arguments you pass to the call method become arguments to the block, and the return value of the block becomes the return value of the call method:

```
f = Proc.new {|x,y| 1.0/(1.0/x + 1.0/y) }
z = f.call(x,y)
```

The Proc class also defines the array access operator to work the same way as call. This means that you can invoke a proc or lambda using a syntax that is like method invocation, where parentheses have been replaced with square brackets. The proc invocation above, for example, could be replaced with this code:

```
z = f[x,y]
```

Ruby 1.9 offers an additional way to invoke a Proc object; as an alternative to square brackets, you can use parentheses prefixed with a period:

```
z = f.(x,y)
```

.() looks like a method invocation missing the method name. This is not an operator that can be defined, but rather is syntactic-sugar that invokes the call method. It can be used with any object that defines a call method and is not limited to Proc objects.

9.5.3 The Arity of a Proc

The arity of a proc or lambda is the number of arguments it expects. (The word is derived from the "ary" suffix of unary, binary, ternary, etc.) Proc objects have an arity method that returns the number of arguments they expect. For example:

```
lambda{||}.arity          # => 0. No arguments expected
lambda{|x| x}.arity       # => 1. One argument expected
lambda{|x,y| x+y}.arity   # => 2. Two arguments expected
```

The notion of arity gets confusing when a Proc accepts an arbitrary number of arguments in an *-prefixed final argument. When a Proc allows optional arguments, the arity method returns a negative number of the form -n-1. A return value of this form indicates that the Proc requires n arguments, but it may optionally take additional arguments as well. -n-1 is known as the one's-complement of n, and you can invert it with the ~ operator. So if arity returns a negative number m, then ~m (or -m-1) gives you the number of required arguments:

```
lambda {|*args|}.arity          # => -1. ~-1 = -(-1)-1 = 0 arguments required
lambda {|first, *rest|}.arity   # => -2. ~-2 = -(-2)-1 = 1 argument required
```

There is one final wrinkle to the arity method. In Ruby 1.8, a Proc declared without any argument clause at all (that is, without any || characters) may be invoked with any number of arguments (and these arguments are ignored). The arity method returns –1 to indicate that there are no required arguments. This has changed in Ruby 1.9: a Proc declared like this has an arity of 0. If it is a lambda, then it is an error to invoke it with any arguments:

```
puts lambda {}.arity # –1 in Ruby 1.8; 0 in Ruby 1.9
```

9.5.4 Proc Equality

The Proc class defines an == method to determine whether two Proc objects are equal. It is important to understand, however, that merely having the same source code is not enough to make two procs or lambdas equal to each other:

```
lambda {|x| x*x } == lambda {|x| x*x } # => false
```

The == method only returns true if one Proc is a clone or duplicate of the other:

```
p = lambda {|x| x*x }
q = p.dup
p == q                          # => true: the two procs are equal
p.object_id == q.object_id      # => false: they are not the same object
```

9.5.5 How Lambdas Differ from Procs

A proc is the object form of a block, and it behaves like a block. A lambda has slightly modified behavior and behaves more like a method than a block. Calling a proc is like yielding to a block, whereas calling a lambda is like invoking a method. In Ruby 1.9, you can determine whether a Proc object is a proc or a lambda with the instance method lambda?. This predicate returns true for lambdas and false for procs.

Return in blocks, procs, and lambdas

The return statement returns from the lexically enclosing method, even when the statement is contained within a block. The return statement in a block does not just return from the block to the invoking iterator, it returns from the method that invoked the iterator. For example:

```
def test
 puts “entering method”
 1.times { puts “entering block”; return } # Makes test method return
 puts “exiting method” # This line is never executed
end
test
```

A proc is like a block, so if you call a proc that executes a return statement, it attempts to return from the method that encloses the block that was converted to the proc. For example:

```
def test
 puts “entering method”
 p = Proc.new { puts “entering proc”; return }
 p.call # Invoking the proc makes method return
 puts “exiting method” # This line is never executed
```

```
end
test
```

Using a return statement in a proc is tricky, however, because procs are often passed around between methods. By the time a proc is invoked, the lexically enclosing method may already have returned:

```
def procBuilder(message) # Create and return a proc
 Proc.new { puts message; return } # return returns from procBuilder
 # but procBuilder has already returned here!
end

def test
 puts "entering method"
 p = procBuilder("entering proc")
 p.call # Prints "entering proc" and raises LocalJumpError!
 puts "exiting method" # This line is never executed
end
test
```

By converting a block into an object, we are able to pass that object around and use it "out of context." If we do this, we run the risk of returning from a method that has already returned, as was the case here. When this happens, Ruby raises a LocalJumpError.

The fix for this contrived example is to remove the unnecessary return statement, of course. But a return statement is not always unnecessary, and another fix is to use a lambda instead of a proc. As we said earlier, lambdas work more like methods than blocks. A return statement in a lambda, therefore, returns from the lambda itself, not from the method that surrounds the creation site of the lambda:

```
def test
 puts "entering method"
 p = lambda { puts "entering lambda"; return }
 p.call # Invoking the lambda does not make the method return
```

```
 puts “exiting method” # This line *is* executed now
end
test
```

The fact that return in a lambda only returns from the lambda itself means that we never have to worry about LocalJumpError:

```
def lambdaBuilder(message) # Create and return a lambda
 lambda { puts message; return } # return returns from the lambda
end
def test
 puts “entering method”
 l = lambdaBuilder(“entering lambda”)
 l.call # Prints “entering lambda”
 puts “exiting method” # This line is executed
end
test
```

Break in blocks, procs and lambdas

We illustrated the behavior of the break statement in a block; it causes the block to return to its iterator and the iterator to return to the method that invoked it. Because procs work like blocks, we expect break to do the same thing in a proc. We can't easily test this, however. When we create a proc with Proc.new, Proc.new is the iterator that break would return from. And by the time we can invoke the proc object, the iterator has already returned. So it never makes sense to have a top-level break statement in a proc created with Proc.new:

```
def test
 puts “entering test method”
 proc = Proc.new { puts “entering proc”; break }
 proc.call # LocalJumpError: iterator has already returned
 puts “exiting test method”
end
test
```

If we create a proc object with an & argument to the iterator method, then we can invoke it and make the iterator return:

def iterator(&proc) puts “entering iterator” proc.call # invoke the proc puts “exiting iterator” # Never executed if the proc breaks end def test iterator { puts “entering proc”; break } end test

Lambdas are method-like, so putting a break statement at the top-level of a lambda, without an enclosing loop or iteration to break out of, doesn’t actually make any sense! We might expect the following code to fail because there is nothing to break out of in the lambda. In fact, the top-level break just acts like a return:

```
def test
 puts “entering test method”
 lambda = lambda { puts “entering lambda”; break; puts “exiting lambda” }
 lambda.call
 puts “exiting test method”
end
test
```

Other control-flow statements

A top-level next statement works the same in a block, proc, or lambda: it causes the yield statement or call method that invoked the block, proc, or lambda to return. If next is followed by an expression, then the value of that expression becomes the return value of the block, proc, or lambda.

redo also works the same in procs and lambdas: it transfers control back to the beginning of the proc or lambda.’

retry is never allowed in procs or lambdas: using it always results in a LocalJumpError.

raise behaves the same in blocks, procs, and lambdas. Exceptions always propagate up the call stack. If a block, proc, or lambda raises an exception and there is no local rescue clause, the exception first propagates to the method that invoked the block with yield or that invoked the proc or lambda with call.

Argument passing to procs and lambdas

Invoking a block with yield is similar to, but not the same as, invoking a method. There are differences in the way argument values in the invocation are assigned to the argument variables declared in the block or method. The yield statement uses yield semantics, whereas method invocation uses invocation semantics. As you might expect, invoking a proc uses yield semantics and invoking a lambda uses invocation semantics:

```
p = Proc.new {|x,y| print x,y }
p.call(1) # x,y=1: nil used for missing rvalue: Prints 1nil
p.call(1,2) # x,y=1,2: 2 lvalues, 2 rvalues: Prints 12
p.call(1,2,3) # x,y=1,2,3: extra rvalue discarded: Prints 12
p.call([1,2]) # x,y=[1,2]: array automatically unpacked: Prints 12
```

This code demonstrates that the call method of a proc handles the arguments it receives flexibly: silently discarding extras, silently adding nil for omitted arguments, and even unpacking arrays. (Or, not demonstrated here, packing multiple arguments into a single array when the proc expects only a single argument.)

Lambdas are not flexible in this way; like methods, they must be invoked with precisely the number of arguments they are declared with:

```
l = lambda {|x,y| print x,y }
l.call(1,2)          # This works
l.call(1)            # Wrong number of arguments
l.call(1,2,3)        # Wrong number of arguments
l.call([1,2])        # Wrong number of arguments
l.call(*[1,2])       # Works: explicit splat to unpack the array
```

9.6 CLOSURES

In Ruby, procs and lambdas are closures. The term “closure” comes from the early days of computer science; it refers to an object that is both an invocable function and a variable binding for that function. When you create a proc or a lambda, the resulting Proc object holds not just the executable block but also bindings for all the variables used by the block.

You already know that blocks can use local variables and method arguments that are defined outside the block. In the following code, for example, the block associated with the collect iterator uses the method argument n:

```
# multiply each element of the data array by n
def multiply(data, n)
 data.collect {|x| x*n }
end
puts multiply([1,2,3], 2) # Prints 2,4,6
```

What is more interesting, and possibly even surprising, is that if the block were turned into a proc or lambda, it could access n even after the method to which it is an argument had returned. The following code demonstrates:

```
# Return a lambda that retains or "closes over" the argument n
def multiplier(n)
 lambda {|data| data.collect{|x| x*n } }
end
doubler = multiplier(2) # Get a lambda that knows how to double
puts doubler.call([1,2,3]) # Prints 2,4,6
```

The multiplier method returns a lambda. Because this lambda is used outside of the scope in which it is defined, we call it a closure; it encapsulates or "closes over" (or just retains) the binding for the method argument n.

9.6.1 Closures and Shared Variables

It is important to understand that a closure does not just retain the value of the variables it refers to—it retains the actual variables and extends their lifetime. Another way to say this is that the variables used in a lambda or proc are not statically bound when the lambda or proc is created. Instead, the bindings are dynamic, and the values of the variables are looked up when the lambda or proc is executed.

As an example, the following code defines a method that returns two lambdas. Because the lambdas are defined in the same scope, they share access to the variables in that scope. When one lambda alters the value of a shared variable, the new value is available to the other lambda:

```
# Return a pair of lambdas that share access to a local variable.
def accessor_pair(initialValue=nil)
 value = initialValue # A local variable shared by the returned lambdas.
 getter = lambda { value } # Return value of local variable.
 setter = lambda {|x| value = x } # Change value of local variable.
 return getter,setter # Return pair of lambdas to caller.
end

getX, setX = accessor_pair(0) # Create accessor lambdas for initial value 0.
puts getX[] # Prints 0. Note square brackets instead of call.
setX[10] # Change the value through one closure.
puts getX[] # Prints 10. The change is visible through the other.''
```

The fact that lambdas created in the same scope share access to variables can be a feature or a source of bugs. Any time you have a method that returns more than one closure, you should pay particular attention to the variables they use. Consider the following code:

```
# Return an array of lambdas that multiply by the arguments
def multipliers(*args)
 x = nil
 args.map {|x| lambda {|y| x*y }}
end
double,triple = multipliers(2,3)
puts double.call(2) # Prints 6 in Ruby 1.8
```

This multipliers method uses the map iterator and a block to return an array of lambdas (created inside the block). In Ruby 1.8, block arguments are not always local to the block, and so all of the lambdas that are created end up sharing access to x, which is a local variable of the multipliers method. As noted above, closures do not capture the current value of the variable: they capture the variable itself. Each of the lambdas created here share the variable x. That variable has only one value, and all of the returned lambdas use that same value. That is why the lambda we name double ends up tripling its argument instead of doubling it.

In this particular code, the issue goes away in Ruby 1.9 because block arguments are always block-local in that version of the language. Still, you can get yourself in trouble any time you create lambdas within a loop and use a loop variables (such as an array index) within the lambda.

9.6.2 Closures and Bindings

The Proc class defines a method named binding. Calling this method on a proc or lambda returns a Binding object that represents the bindings in effect for that closure.

A Binding object doesn't have interesting methods of its own, but it can be used as the second argument to the global eval function, providing a context in which to evaluate a string of Ruby code. In Ruby 1.9, Binding has its own eval method, which you may prefer to use. (Use ri to learn more about Kernel.eval and Binding.eval.)

The use of a Binding object and the eval method gives us a back door through which we can manipulate the behavior of a closure. Take another look at this code from earlier:

```
# Return a lambda that retains or "closes over" the argument n
def multiplier(n)
 lambda {|data| data.collect{|x| x*n } }
end
doubler = multiplier(2)          # Get a lambda that knows how to double
puts doubler.call([1,2,3])       # Prints 2,4,6
```

Now suppose we want to alter the behavior of doubler:

```
eval("n=3", doubler.binding)   # Or doubler.binding.eval("n=3") in Ruby 1.9
puts doubler.call([1,2,3])     # Now this prints 3,6,9!
```

As a shortcut, the eval method allows you to pass a Proc object directly instead of passing the Binding object of the Proc. So we could replace the eval invocation above with:

```
eval("n=3", doubler)
```

Bindings are not only a feature of closures. The Kernel.binding method returns a Binding object that represents the bindings in effect at whatever point you happen to call it.

9.7 METHOD OBJECTS

Ruby's methods and blocks are executable language constructs, but they are not objects. Procs and lambdas are object versions of blocks; they can be executed and also manipulated as data. Ruby has powerful metaprogramming (or reflection) capabilities, and methods can actually be represented as instances of the Method class.

Remember

You should note that invoking a method through a Method object is less efficient than invoking it directly. Method objects are not typically used as often as lambdas and procs.

The Object class defines a method named method. Pass it a method name, as a string or a symbol, and it returns a Method object representing the named method of the receiver (or throws a NameError if there is no such method).

For example:

```
m = 0.method(:succ) # A Method representing the succ method of Fixnum 0
```

In Ruby 1.9, you can also use public_method to obtain a Method object. It works like method does but ignores protected and private methods.

The Method class is not a subclass of Proc, but it behaves much like it. Method objects are invoked with the call method (or the [] operator), just as Proc objects are. And Method defines an arity method just like the arity method of Proc. To invoke the Method m:

```
puts m.call # Same as puts 0.succ. Or use puts m[].
```

Invoking a method through a Method object does not change the invocation semantics, nor does it alter the meaning of control-flow statements such as return and break. The call method of a Method object uses method-invocation semantics, not yield semantics. Method objects, therefore, behave more like lambdas than like procs.

Method objects work very much like Proc objects and can usually be used in place of them. When a true Proc is required, you can use Method.to_proc to convert a Method

to a Proc. This is why Method objects can be prefixed with an ampersand and passed to a method in place of a block.

For example: def square(x); x*x; end
puts (1..10).map(&method(:square))

One important difference between Method objects and Proc objects is that Method objects are not closures. Ruby's methods are intended to be completely self-contained, and they never have access to local variables outside of their own scope. The only binding retained by a Method object, therefore, is the value of self—the object on which the method is to be invoked.

In Ruby 1.9, the Method class defines three methods that are not available in 1.8: name returns the name of the method as a string; owner returns the class in which the method was defined; and receiver returns the object to which the method is bound. For any method object m, m.receiver.class must be equal to or a subclass of m.owner.

9.7.1 Unbound Method Objects

In addition to the Method class, Ruby also defines an UnboundMethod class. As its name suggests, an UnboundMethod object represents a method, without a binding to the object on which it is to be invoked. Because an UnboundMethod is unbound, it cannot be invoked, and the UnboundMethod class does not define a call or [] method.

To obtain an UnboundMethod object, use the instance_method method of any class or module:

unbound_plus = Fixnum.instance_method("+")

In Ruby 1.9, you can also use public_instance_method to obtain an UnboundMethod object. It works like instance_method does, but it ignores protected and private methods.

In order to invoke an unbound method, you must first bind it to an object using the bind method:

plus_2 = unbound_plus.bind(2) # Bind the method to the object 2

The bind method returns a Method object, which can be invoked with its call method:

```
sum = plus_2.call(2) # => 4
```

Another way to obtain an UnboundMethod object is with the unbind method of the Method class:

```
plus_3 = plus_2.unbind.bind(3)
```

In Ruby 1.9, UnboundMethod has name and owner methods that work just as they do for the Method class.

9.8 FUNCTIONAL PROGRAMMING

Ruby is not a functional programming language in the way that languages like Lisp and Haskell are, but Ruby's blocks, procs, and lambdas lend themselves nicely to a functional programming style. Any time you use a block with an Enumerable iterator like map or inject, you're programming in a functional style. Here are examples using the map and inject iterators:

```
# Compute the average and standard deviation of an array of numbers
mean = a.inject {|x,y| x+y } / a.size
sumOfSquares = a.map{|x| (x-mean)**2 }.inject{|x,y| x+y }
standardDeviation = Math.sqrt(sumOfSquares/(a.size-1))
```

If the functional programming style is attractive to you, it is easy to add features to Ruby's built-in classes to facilitate functional programming.

9.8.1 Applying a Function to an Enumerable

map and inject are two of the most important iterators defined by Enumerable. Each expects a block. If we are to write programs in a function-centric way, we might like methods on our functions that allow us to apply those functions to a specified Enumerable object:

```
# This module defines methods and operators for functional programming.
module Functional
 # Apply this function to each element of the specified Enumerable,
```

```
 # returning an array of results. This is the reverse of Enumerable.map.
# Use | as an operator alias. Read "|" as "over" or "applied over".
 #
 # Example:
 # a = [[1,2],[3,4]]
 # sum = lambda {|x,y| x+y}
 # sums = sum|a # => [3,7]
 def apply(enum)
 enum.map &self
 end
 alias | apply

 # Use this function to "reduce" an enumerable to a single quantity.
 # This is the inverse of Enumerable.inject.
 # Use <= as an operator alias.
 # Mnemonic: <= looks like a needle for injections
 # Example:
 # data = [1,2,3,4]
 # sum = lambda {|x,y| x+y}
 # total = sum<=data # => 10
 def reduce(enum)
 enum.inject &self
 end
 alias <= reduce
end

# Add these functional programming methods to Proc and Method classes.
class Proc; include Functional; end
class Method; include Functional; end
```

Notice that we define methods in a module named Functional, and then we include this module into both the Proc and Method classes. In this way, apply and reduce work for both proc and method objects. Most of the methods that follow also define methods in this Functional module, so that they work for both Proc and Method.

With apply and reduce defined as above, we could refactor our statistical computations as follows:

```
sum = lambda {|x,y| x+y }                    # A function to add two numbers
mean = (sum<=a)/a.size                       # Or sum.reduce(a) or a.inject(&sum)
deviation = lambda {|x| x-mean }             # Function to compute difference from mean
square = lambda {|x| x*x }                   # Function to square a number
standardDeviation = Math.sqrt((sum<=square|(deviation|a))/(a.size-1))
```

Notice that the last line is succinct but that all the nonstandard operators make it hard to read. Also notice that the | operator is left-associative, even when we define it ourselves. The syntax, therefore, for applying multiple functions to an Enumerable requires parentheses. That is, we must write square|(deviation|a) instead of square|deviation|a.

Did You Know?

The first functional programming language, LISP, was developed in the late 1950s for the IBM 700/7000 series of scientific computers by John McCarthy while at Massachusetts Institute of Technology (MIT). LISP functions were defined using Church's lambda notation, extended with a label construct to allow recursive functions.

9.8.2 Composing Functions

If we have two functions f and g, we sometimes want to define a new function h which is f(g()), or f composed with g. We can write a method that performs function composition automatically, as follows:

```
module Functional
 # Return a new lambda that computes self[f[args]].
 # Use * as an operator alias for compose.
 # Examples, using the * alias for this method.
 #
 # f = lambda {|x| x*x }
 # g = lambda {|x| x+1 }
 # (f*g)[2] # => 9
```

```
# (g*f)[2] # => 5
#
# def polar(x,y)
# [Math.hypot(y,x), Math.atan2(y,x)]
# end
# def cartesian(magnitude, angle)
# [magnitude*Math.cos(angle), magnitude*Math.sin(angle)]
# end
# p,c = method :polar, method :cartesian
# (c*p)[3,4] # => [3,4]
#
def compose(f)
if self.respond_to?(:arity) && self.arity == 1
lambda {|*args| self[f[*args]] }
else
lambda {|*args| self[*f[*args]] }
end
end
# * is the natural operator for function composition.
alias * compose
end
```

The example code in the comment demonstrates the use of compose with Method objects as well as lambdas. We can use this new * function composition operator to slightly simplify our computation of standard deviation. Using the same definitions of the lambdas sum, square, and deviation, the computation becomes:

```
standardDeviation = Math.sqrt((sum<=square*deviation|a)/(a.size-1))
```

The difference is that we compose square and deviation into a single function before applying it to the array a.

9.8.3 Partially Applying Functions

In functional programming, partial application is the process of taking a function and a partial set of argument values and producing a new function that is equivalent to the original function with the specified arguments fixed. For example:

```
product = lambda {|x, y| x*y }              # A function of two arguments
double = lambda {|x| product(2,x) }         # Apply one argument
```

Partial application can be simplified with appropriate methods (and operators) in our Functional module:

```
module Functional
 #
 # Return a lambda equivalent to this one with one or more initial
 # arguments applied. When only a single argument
 # is being specified, the >> alias may be simpler to use.
 # Example:
 # product = lambda {|x,y| x*y}
 # doubler = lambda >> 2
 #
 def apply_head(*first)
 lambda {|*rest| self[*first.concat(rest)]}
 end

#
 # Return a lambda equivalent to this one with one or more final arguments
 # applied. When only a single argument is being specified,
 # the << alias may be simpler.
 # Example:
 # difference = lambda {|x,y| x-y }
 # decrement = difference << 1
 #
 def apply_tail(*last)
```

```
 lambda {|*rest| self[*rest.concat(last)]}
 end
 # Here are operator alternatives for these methods. The angle brackets
 # point to the side on which the argument is shifted in.
 alias >> apply_head # g = f >> 2 -- set first arg to 2
 alias << apply_tail # g = f << 2 -- set last arg to 2
end
```

Using these methods and operators, we can define our double function simply as product>>2. We can use partial application to make our standard deviation computation somewhat more abstract, by building our deviation function from a more generalpurpose difference function:

```
difference = lambda {|x,y| x-y } # Compute difference of two numbers
deviation = difference<<mean # Apply second argument
```

9.8.4 Memoizing Functions

Memoization is a functional programming term for caching the results of a function invocation. If a function always returns the same value when passed the same arguments, if there is reason to believe that the same arguments will be used repeatedly, and if the computation it performs is somewhat expensive, then memoization may be a useful optimization. We can automate memoization for Proc and Method objects with the following method:

```
module Functional
 #
 # Return a new lambda that caches the results of this function and
 # only calls the function when new arguments are supplied.
 #
 def memoize
 cache = {} # An empty cache. The lambda captures this in its closure.
 lambda {|*args|
 # notice that the hash key is the entire array of arguments!
 unless cache.has_key?(args) # If no cached result for these args
 cache[args] = self[*args] # Compute and cache the result
```

```
  end
  cache[args] # Return result from cache
  }
  end
  # A (probably unnecessary) unary + operator for memoization
  # Mnemonic: the + operator means "improved"
  alias +@ memoize # cached_f = +f
end
```

Here's how we might use the memoize method or the unary + operator:

```
# A memoized recursive factorial function
factorial = lambda {|x| return 1 if x==0; x*factorial[x-1]; }.memoize
# Or, using the unary operator syntax
factorial = +lambda {|x| return 1 if x==0; x*factorial[x-1]; }
```

Note that the factorial function here is a recursive function. It calls the memorized version of itself, which produces optimal caching. It would not work as well if you defined a recursive nonmemoized version of the function and then defined a distinct memoized version of that:

```
factorial = lambda {|x| return 1 if x==0; x*factorial[x-1]; }
cached_factorial = +factorial # Recursive calls aren't cached!
```

9.8.5 Symbols, Methods, and Procs

There is a close relationship between the Symbol, Method, and Proc classes. We've already seen the method method, which takes a Symbol argument and returns a Method object. Ruby 1.9 adds a useful to_proc method to the Symbol class. This method allows a symbol to be prefixed with & and passed as a block to an iterator. The symbol is assumed to name a method. When the Proc created with this to_proc method is invoked, it calls the named method of its first argument, passing any remaining arguments to that named method. Here's how you might use it:

```
# Increment an array of integers with the Fixnum.succ method
```

[1,2,3].map(&:succ) # => [2,3,4]
Without Symbol.to_proc, we'd have to be slightly more verbose:
[1,2,3].map {|n| n.succ }

Symbol.to_proc was originally devised as an extension for Ruby 1.8, and it is typically implemented like this:

```
class Symbol
 def to_proc
 lambda {|receiver, *args| receiver.send(self, *args)}
 end
end
```

This implementation uses the send method to invoke a method named by a symbol. We could also do it like this:

```
class Symbol
 def to_proc
 lambda {|receiver, *args| receiver.method(self)[*args]}
 end
end
```

In addition to to_proc, we can define some related and possibly useful utilities. Let's start with the Module class:

```
class Module
 # Access instance methods with array notation. Returns UnboundMethod,
 alias [] instance_method
end
```

Here, we're simply defining a shorthand for the instance_method method of the Module class. Recall that that method returns an UnboundMethod object, that cannot be invoked until bound to a particular instance of its class. Here's an example using this new notation (notice the appeal of indexing a class with the names of its methods!):

String[:reverse].bind("hello").call # => "olleh"

Binding an unbound method can also be made simpler with a bit of the same syntactic sugar:

gar:

```
class UnboundMethod
 # Allow [] as an alternative to bind.
 alias [] bind
end
```

With this alias in place, and using the existing [] alias for calling a method, this code becomes:

```
String[:reverse]["hello"][] # => "olleh"
```

The first pair of brackets indexes the method, the second pair binds it, and the third pair calls it.

Next, if we're going to use the [] operator for looking up the instance methods of a class, how about using []= for defining instance methods:

```
class Module
 # Define a instance method with name sym and body f.
 # Example: String[:backwards] = lambda { reverse }
 def []=(sym, f)
self.instance_eval { define_method(sym, f) }
 end
end
```

The definition of this []= operator may be confusing—this is advanced Ruby. define_method is a private method of Module. We use instance_eval (a public method of Object) to run a block (including the invocation of a private method) as if it were inside the module on which the method is being defined.

Let's use this new []= operator to define a new Enumerable.average method:

```
Enumerable[:average] = lambda do
 sum, n = 0.0, 0
 self.each {|x| sum += x; n += 1 }
 if n == 0
 nil
 else
 sum/n
 end
end
```

We've used the [] and []= operators here to get and set instance methods of a class or module. We can do something similar for the singleton methods of an object (which include the class methods of a class or module). Any object can have a singleton method, but it does not make sense to define an [] operator on the Object class, as so many subclasses define that operator. For singleton methods, therefore, we could take the opposite approach and define operators on the Symbol class:

```
#
# Add [] and []= operators to the Symbol class for accessing and setting
# singleton methods of objects. Read : as "method" and [] as "of".
# So :m[o] reads "method m of o".
#
class Symbol
 # Return the Method of obj named by this symbol. This may be a singleton
 # method of obj (such as a class method) or an instance method defined
 # by obj.class or inherited from a superclass.
 # Examples:
 # creator = :new[Object] # Class method Object.new
 # doubler = :*[2] # * method of Fixnum 2
 #
 def [](obj)
 obj.method(self)
 end
```

```
 # Define a singleton method on object o, using Proc or Method f as its body.
 # This symbol is used as the name of the method.
 # Examples:
 #
 # :singleton[o] = lambda { puts "this is a singleton method of o" }
 # :class_method[String] = lambda { puts "this is a class method" }
 #
 # Note that you can't create instance methods this way. See Module.[]=
#
 def []=(o,f)
 # We can't use self in the block below, as it is evaluated in the
 # context of a different object. So we have to assign self to a variable.
 sym = self
 # This is the object we define singleton methods on.
 eigenclass = (class << o; self end)
 # define_method is private, so we have to use instance_eval to execute it.
 eigenclass.instance_eval { define_method(sym, f) }
 end
end
```

With this Symbol.[] method defined, along with the Functional module, we can write clever (and unreadable) code like this:

```
dashes = :*['-'] # Method * of '-'
puts dashes[10] # Prints "----------"

y = (:+[1]*:*[2])[x] # Another way to write y = 2*x + 1
```

The definition of []= for Symbol is like that of []= for Module, in that it uses instance_eval to invoke the define_method method. The difference is that singleton methods are not defined within a class, as instance methods are, but in the eigenclass of the object.

SUMMARY

- A method is a named block of parameterized code associated with one or more objects. A method invocation specifies the method name, the object on which it is to be invoked (sometimes called the receiver), and zero or more argument values that are assigned to the named method parameters.
- Many languages distinguish between functions, which have no associated object, and methods, which are invoked on a receiver object.
- A def statement that defines a method may include exception-handling code in the form of rescue, else, and ensure clauses, just as a begin statement can.
- An equals sign suffix signifies that the method is a setter that can be invoked using assignment syntax.
- Method aliasing is one of the things that makes Ruby an expressive and natural language. When there are multiple names for a method, you can choose the one that seems most natural in your code.
- Ruby allows parentheses to be omitted from most method invocations. In simple cases, this results in clean-looking code.
- Simple method declarations include a comma-separated list of argument names (in optional parentheses) after the method name.
- A block is a chunk of Ruby code associated with a method invocation, and that an iterator is a method that expects a block.
- A proc is the object form of a block, and it behaves like a block. A lambda has slightly modified behavior and behaves more like a method than a block.
- The return statement returns from the lexically enclosing method, even when the statement is contained within a block.
- Ruby's methods and blocks are executable language constructs, but they are not objects. Procs and lambdas are object versions of blocks; they can be executed and also manipulated as data

KNOWLEDGE CHECK

1. **Which of the following is not a valid datatype in Ruby?**
   a. Float
   b. Integer
   c. Binary
   d. Timedate
2. **Which of the following are valid floating point literal?**
   a. 5
   b. 2
   c. 0.5
   d. None of the mentioned
3. **Why do we use =begin and =end?**
   a. To mark the start and end of multiline comment
   b. To comment multiple lines
   c. To avoid the use of # again and again
   d. All of above
4. **How do you express error messages in a form (do |f|)?**
   a. ruby make
   b. f.error_messages
   c. for ad in @ads
   d. underscores
5. **What is naming convention for classes?**
   a. ActionPack
   b. CamelCase
   c. ruby make
   d. a web page

REVIEW QUESTIONS

1. What do you understand by method return value? Discuss.
2. How to invoke a method on an object.
3. How to declare an argument that has a default value, so that the argument can be omitted when the method is invoked?

4. Give a detailed overview on functional programming
5. How lambdas differ from procs?

Check Your Result

1. (d) 2. (c) 3. (d) 4. (b) 5. (b)

REFERENCES

1. Abran, A., Lopez, M., and Habra, N. 2004. An Analysis of the McCabe Cyclomatic Complexit Number, Proceedings of the 14th International Workshop on Software Measurement (IWSM) IWSM-Metrikon, 2004, Magdeburg, Germany: SpringerVerlag, 391-405.
2. Bin Tang, C., 2015. Explore MQTT and the Internet of Things service on IBM Bluemix. http://ibm.co/1LDiJFD
3. Harrison, W. 2000. N=1, an Alternative for Software Engineering Research? Proc. Workshop Beg, Borrow, or Steal: Using Multidisciplinary Approaches in Empirical Software Eng. Research, Int'l Conf. Software Eng., Aug. 2000.
4. Khare, S., Tambe, S., An, K., Gokhale, A. and Pazandak, P. 2015. Functional Reactive Stream Processing for Data-centric Publish/Subscribe Systems. http://bit.ly/1Y CDz15.
5. Namiot, D. and Sneps-Sneppe, M. 2014. On IoT Programming, International Journal of Open Information Technologies 2(10).
6. Newton, R. and Welsh, M. 2004. Region streams: functional macroprogramming for sensor networks, Proceeedings of the 1st international workshop on Data management for sensor networks: in conjunction with VLDB, 78-87.
7. Ray, B., Posnett, D., Filkov, V. and Devanbu, P. T. 2014. "A Large Scale Study of Programming Languages and Code Quality in Github" Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.
8. Reese, L. 2015. A Comparison of Open Source Hardware: Intel Galileo vs. Raspberry Pi. Technical Report. Mouser Electronics. http://bit.ly/1Opw3np.
9. Schneier, B. 2010. The Dangers of a Software Monoculture. Information Security Magazine, November 2010.
10. Sivieri, A., Mottola, L. and Cugola, G. 2012. Drop the Phone and Talk to the Physical World: Programming the Internet of Things with Erlang, SESENA '12 Proceedings of the Third International Workshop on Software Engineering for Sensor Network Applications.
11. Subramaniam, V. 2014. Functional Programming in Java: Harnessing the Power of Java 8 Lambda Expressions, O'Reilly.
12. Wortmann, F. and Flüchter, K. 2015. Internet of Things - Technology and Value Added, Business & Information Systems Engineering 57(3): 221-224.
13. Zhou, C. and Zhang, X., 2014. Toward the Internet of Things Application and Management: A Practical Approach, WOWMOM, 2014, 2014 IEEE 15th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM).

Index

W

X

Level: Beginner to Advanced
Subject: Computer and Information Science

Basic Computer Coding: RUBY

2nd Edition

Ruby is a dynamic, reflective, object-oriented, general-purpose programming language. Ruby is a pure Object-Oriented language developed by Yukihiro Matsumoto. One of the goals of Ruby is to allow the simple and fast creation of web applications. The language itself satisfies this goal. Because of this, there is much less tedious work with this language than many other programming languages. More specifically, Ruby is a scripting language designed for front- and back-end web development, as well as other similar applications. It's a robust, dynamically typed, object-oriented language, with high-level syntax that makes programming with it feel almost like coding in English. In fact, some people feel that they can practically understand Ruby code before even learning how to program. In the world of computer programming, there is an infinite amount of information to learn. This book covers certain topics that are beneficial to the beginner.

This edition is systematically divided into nine chapters. This book is your guide to rapid, real-world software development with this unique and elegant language. The book will excite students on the capabilities of computer programming and inspire them to delve deeper into the computer science discipline. It will give you plenty of practice to commit basic Ruby syntax to long-term memory so you can focus on solving real-world problems and building real-world applications.

ISBN 978-1-98467-605-4

BIBLIOTEX
Digital Library

www.3ge-learning.com
email: info@3ge-learning.com