LEARNING TO PROGRAM

STEVEN FOOTE

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Learning to Program

Addison-Wesley Learning Series

LEARNING JavaScript
A Hands-On Guide to the Fundamentals of Modern JavaScript
TIM WRIGHT
LEARNING OBJECTIVE-C 2.0
A Hands-on Guide to Objective-C for Mac and iOS Developers
SECOND EDITION
ROBERT CLAIR
LEARNING Cocos2D
Hands-On Guide to Building iOS Games with Cocos2D, Box2D, and Chipmunk
ROD STROUGO
RAY WENDERLICH
LEARNING ANDROID GAME PROGRAMMING
A Hands-On Guide to Building Your First Android Game
RICK ROGERS
Addison-Wesley

Visit informit.com/learningseries for a complete list of available publications.

The Addison-Wesley Learning Series is a collection of hands-on programming guides that help you quickly learn a new technology or language so you can apply what you've learned right away.

Each title comes with sample code for the application or applications built in the text. This code is fully annotated and can be reused in your own projects with no strings attached. Many chapters end with a series of exercises to encourage you to reexamine what you have just learned, and to tweak or adjust the code as a way of learning.

Titles in this series take a simple approach: they get you going right away and leave you with the ability to walk off and build your own application and apply the language or technology to whatever you are working on.

Addison-Wesley informIT the trusted technology learning source | Safari Books Online

ALWAYS LEARNING PEARSON

Learning to Program

Steven Foote

Addison-Wesley

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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2014951063

Copyright © 2015 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-7897-5339-7

ISBN-10: 0-7897-5339-1

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.

First printing: November 2014

Editor-in-Chief
Mark Taub

Acquisitions Editor
Mark Taber

Managing Editor
Sandra Schroeder

Project Editor
Mandie Frank

Copy Editor
Krista Hansing

Indexer
Ken Johnson

Proofreader
Jess DeGabriele

Technical Reviewers
Seth McLaughlin
Jeremy Foote

Editorial Assistant
Vanessa Evans

Designer
Chuti Prasertsith

Compositor
Jake McFarland

❖

For Paige

❖

Contents at a Glance

Introduction Why I Wrote This Book 1

1 "Hello, World!" Writing Your First Program 5

2 How Software Works 23

3 Getting to Know Your Computer 47

4 Build Tools 63

5 Data (Types), Data (Structures), Data(bases) 83

6 Regular Expressions 109

7 `if`, `for`, `while`, and When 131

8 Functions and Methods 153

9 Programming Standards 173

10 Documentation 185

11 Planning 199

12 Testing and Debugging 209

13 Learning to Fish: How to Acquire a Lifetime of Programming Knowledge 233

14 Building Your Skills 243

15 Advanced Topics 255

Glossary 273

Index 293

Table of Contents

Introduction Why I Wrote This Book 1

Why You Should Read This Book 3

Your Project 3

1 “Hello, World!” Writing Your First Program 5

Choose a Text Editor 5

- Core Features 6

Making Your Choice 8

- Sublime Text 9
- TextMate 9
- Notepad++ 9
- Gedit 9
- Vim 10
- Eclipse 10
- IntelliJ 11
- Xcode 11
- Visual Studio 11

Create a Project Directory 12

Start Small: Create a Test File 12

- How HTML and JavaScript Work Together in a Browser 13
- The Value of Small Changes 15

Build on Your Success 17

- Reference Your JavaScript in `manifest.json` 20
- Let It Run! 20

Great Power, Great Responsibility 21

Summing Up 21

2 How Software Works 23

What Is “Software”? 23

Software Life Cycle 24

Source Code—Where It All Starts 25

- A Set of Instructions 25
- Programming Languages 26
- From Source Code to 0’s and 1’s 31

Compiled vs. Interpreted Languages: When Does the Source Code Become Binary? 31
Runtime Environment 32
Execution by the Processor 34
Input and Output 34
Making Software Useful (and Reusable) with Input 34
Where Does the Input Come From? 35
How the Software Gets the Input 36
Types of Output 36
GIGO: Garbage In, Garbage Out 37
State 38
Add State to Kittenbook 39
Memory and Variables 41
Variables 41
Variable Storage 42
A Finite Resource 44
Memory Leaks 44
Summing Up 45

3 Getting to Know Your Computer 47
Your Computer Is Stupid 47
Your Computer Is Magic 48
Standing on the Shoulders of Giants 48
Computer Guts 48
Processor 48
Short-Term Memory 49
Long-Term Memory 49
Using Your Computer 50
The File System 50
The Command Line: Take Control 52
Summing Up 62

4 Build Tools 63
Automate (Almost) Everything 63
Install Node 64
Install Grunt 66
Software That Helps You Create Software 69

Avoid Mistakes 70
Work Faster 70
Tasks to Automate 70
Compile 71
Test 71
Package 72
Deploy 72
Build Your Own Build 72
`Gruntfile.js` 73
Use Grunt Plug-ins 73
Load Grunt Plug-ins 76
Register Tasks 76
Watch This! 78
Summing Up 80

5 Data (Types), Data (Structures), Data(bases) 83
Data Types 83
Why Different Data Types Exist 84
Primitive Data Types 84
Composite Data Types 89
Dynamically and Statically Typed Languages 96
Data Structures 96
Set 99
Stack 99
Tree 100
Graph 101
How to Choose an Effective Data Structure 104
Databases 104
Long-Term (Persistent) Storage 104
Relational Databases 104
A Brief Introduction to SQL 106
Summing Up 107

6 Regular Expressions 109
Ctrl+F on Steroids: Looking for Patterns 109
Using Regular Expressions in JavaScript 110
Repetition 111

? 111
+ 111
* 112
Special Characters and Escaping 112
{1,10}: Make Your Own Super Powers 113
Match Anything, Period 113
Don’t Be Greedy 114
Understanding Brackets from [A-Za-z] 115
Lists of Characters 115
Ranges 115
Negation 116
A Pattern for Phone Numbers 116
I Need My \s 119
Shortcuts for Brackets 119
Limitations 121
Capture the Tag 124
Advanced Find and Replace 125
The Beginning and the End (of a Line) 126
Flags 126
Global 126
Ignore Case 126
Multiline 127
When Will You Ever Use Regex? 127
grep 127
Code Refactoring 127
Validation 128
Data Extraction 128
Summing Up 129

7 if, for, while, and When 131
Operators 131
Comparison Operators 131
Logical Operators 132
Unary Operators 134
Binary Operators 134
Ternary Operators 136
“Truthy” and “Falsy” 139

“Syntactic Sugar” 140
Looping Through an Array 142
Looping Through Images 142
Nested Loops 143
You Need a Break 143
Infinite Loops 145
Take Another Break 146
When You Don’t Know When to Stop 146
When 147
Events 147
Listeners 147
Cron Jobs 148
Timeouts 149
Catch When Things Go Wrong 150
Writing Robust Code 151
Summing Up 151

8 Functions and Methods 153
Function Structure 153
Definition 154
Invocation 154
Arguments 155
Call Stack 157
Code Encapsulation 158
Do One Thing Well 158
Divide and Conquer 159
A Place for Everything and Everything in Its Place 162
Code Reuse 163
Solve the General Problem 163
Do More with Less 163
Don’t Repeat Yourself (DRY) 165
Scope 166
Global 167
Local 168
How Variable Lookups Happen 168
Summing Up 171

9 Programming Standards 173

Coding Conventions 173
- Setting Standards 174
- To Hack or Not to Hack 174
- Pay Now or Pay Later 175
- Writing Maintainable Code 175

Code Formatting 176
- Keep It Consistent 177
- Whitespace 178

It Doesn’t Happen on Its Own: Make Rules 178

Using the Work of Others 180
- Build Faster 180
- Open Source Software 181
- Built by the Community 181
- When to Build It Yourself 182

Best Practices 182
- Documentation 182
- Planning 183
- Testing 183

Summing Up 183

10 Documentation 185

Document Intentions 186
- Self-Documenting Code 187
- Don’t Document the Obvious 189
- The Danger of Outdated Documentation 190
- Find Bugs Using Documentation 191

Document for Yourself 191
- How Good Is Your Memory? 191
- Document to Learn 192
- Documentation Beyond Comments 192

Document for Others 196
- Document Your Decisions 196
- Document Your Resources 197
- Document to Teach 197

Summing Up 198

11 Planning 199

Think Before You Build 199

Create a Specification 200

Design an Architecture 200

- Draw Diagrams 201
- Try to Break Your System 202

Iterative Planning 203

Design for Extensibility 203

What Are Your Priorities? 204

- User Experience 204
- Performance 204
- Security 205
- Scalability 205
- Deadlines 205

The Balancing Act 206

- Identify and Create Constraints 206
- Know What You Can and Can’t Do 206

Summing Up 207

12 Testing and Debugging 209

Manual Testing 209

- Test As You Work 210
- Try Something Crazy 210
- Eat Your Own Dog Food 211

Automated Testing 211

- Unit Tests 212
- Set Up Tests for Kittenbook 215
- Epic Fail! 218
- Spies Like Us (and We Like Spies) 219

Integration Tests 221

- Catch Problems Early 222

Debugging 222

- Errors 223
- Logs 224
- Breakpoints 224
- Inspecting, Watching, and the Console 228

Stepping Through the Code 229
Call Stack 231
Find the Root Cause 231
Code, Test, Debug, Repeat 232
Summing Up 232

13 Learning to Fish: How to Acquire a Lifetime of Programming Knowledge 233
How to Search 234
Finding the Right Terms 235
Working Backward 236
Identifying Quality Resources 236
Personal Blogs: Hidden Gems 237
Where, When, and How to Ask Programming Questions 237
Where 237
When 240
How 241
Learn by Teaching 241
Summing Up 242

14 Building Your Skills 243
Make kittenbook Your Own 243
Restyle Facebook 243
Add New Functionality 244
Share Your Version of Kittenbook 245
Find Your Own Project 246
Solve Your Own Problem 246
Be Ambitious 246
Get Help, Give Help 247
Open Source Projects 247
GitHub 247
Finding a Project 248
Different Ways to Contribute 248
Create Your Own 249
Free Online Education 249
Project Euler 249
Udacity 250
Coursera 250

- codecademy 251
- Khan Academy 251
- Tutorials 251

Paid Education 252

- Read a Book 252
- Udacity and Coursera 252
- Treehouse 253

Summing Up 253

15 Advanced Topics 255

Version Control 255

- Why Use Version Control? 256
- Working with a Team 257
- Subversion 260
- Git 260

OOP (Object-Oriented Programming) 265

- Classes 266
- Inheritance 266
- Instances 267

Design Patterns 268

- Pub Sub 268
- Mediator 269
- Singleton 270

Summing Up 270

Glossary 273

Index 293

Acknowledgments

I wish to thank the following people for their help in the preparation of various versions of this text: Jimmy Chan, Prash Jain, and Jim Gourgoutis.

At Pearson, I'd like to thank Mark Taber and my project editor, Mandie Frank. Thanks also to my copy editor, Krista Hansing, and my technical editors, Seth McLaughlin and Jeremy Foote. Finally, I'd like to thank all the other people from Pearson who were involved on this project, even if I did not work with them directly.

About the Author

Steven Foote is a web developer at LinkedIn. A self-taught programmer who loves technology, especially the Web, he has a Bachelor's degree and Master's degree in Accountancy from Brigham Young University. While working on his Master's degree, he built all aspects of two AJAX-y web applications, from visual design to server and database maintenance, and everything in between.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write directly to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail we receive, we might not be able to reply to every message.

When you write, please be sure to include this book's title and author, as well as your name and phone number or email address.

Email:	feedback@developers-library.info
Mail:	Reader Feedback Addison-Wesley Developer's Library 800 East 96th Street Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at www.informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

This page intentionally left blank

Introduction

Why I Wrote This Book

Like most great (accountant) stories, mine starts with an Excel spreadsheet. The year was 2008, and I was studying accounting at Brigham Young University in Provo, Utah. I was also working as a clerk at the Law School library. One day my boss asked me if I knew how to randomize a list of names in Excel. "Of course," I lied and then proceeded to do what every well-intentioned overpromiser knows to do. Google showed me at least three different ways to randomize a list in Excel, and within 2 minutes, I returned the randomized list to my boss. At that point, she decided that I was good with computers and should work for the library systems department. I'm not certain that an aptitude to Google stuff equates to being good with computers, but I'm grateful that she thought so.

It was there in the law library systems department that my programming journey began. On my first day, my boss gave me a 25-year-old book called *Programming Perl* and showed me to my desk in a windowless room cluttered with old computers, keyboards, and monitors. He was busy with some other things, he told me as he left, but the Perl book would keep me busy until he came back. I opened the book and started reading, and I've never been the same since.

The book was written before Windows even existed, so it expected the reader to be using the UNIX Operating System (which I had never heard of). I ignored that and kept using my Windows XP computer, which had recently been retired from the library's computer lab (before that day, I had no idea there was such a thing as a computer too old for a library computer lab). The book told me to get a copy of Perl by sending a self-addressed, stamped envelope to some address and then getting a floppy disk mailed to me. I decided to use my Google skills instead and found instructions on downloading and installing Perl. I had downloaded software before, but I had no idea you have to download and install programming languages. I was getting a bit nervous that I might be destroying my computer, but I figured it wouldn't really be missed, so I kept going.

The first chapter of the book showed some sample code that was supposed to print `Hello, world!` It looked something like this:

```
print("Hello, world!");
```

I was supposed to save the sample code to a file named `hello_world.pl`. The only software I had ever used to input text into my computer was Microsoft Word, so I opened Word and started typing. It took me about 45 seconds to realize that Microsoft Word was the wrong place to be writing code and 45 minutes to figure out the right place. For all of my Google know-how, I couldn't figure out what to search for to find the right place to type code. I was in a new and unfamiliar world now, and the search terms I was accustomed to were useless. Eventually, I found an answer in Notepad (as you will learn in Chapter 1, "'Hello, World!' Writing Your First Program," Notepad was not the right answer, but it did work), and I continued trudging along, only a little discouraged.

Finally, I typed the code from the book into Notepad and saved the file. Nothing happened. But I was "good with computers," and I knew that if something isn't working, you should try restarting it. So I closed the file and tried to open it again by double-clicking it. A little black box with white text flashed on the screen for a fraction of a second; then it was gone, and Notepad never opened. By this point, I was sure I had broken the computer, and I wasn't quite as confident that my boss wouldn't be upset with a broken computer, however old. Then the idea crossed my mind that my program had actually worked. I ran to the printer across the room to see if I had successfully printed `Hello world!`. The printer sat idly, with no freshly printed paper in the tray. I tried restarting the printer, just in case. No luck. (I didn't find out until much later that `print` means print to the screen, not print on a piece of paper.) The rest of the afternoon proceeded in a similar, frustrating fashion.

By the end of the day, I was feeling like I was *not* good with computers; indeed, the computers were having their way with me, and they seemed to be enjoying it. I was ready to quit, but I needed the job—and I don't like quitting. I wasn't about to let the computers win. Over the following weeks and months, I made slow and sporadic progress. I backtracked a lot, and I had to keep learning the same things over and over again. The only people who might have been able to guide me and answer my questions were either too busy or too experienced to be helpful. Like the Perl book I was reading (which assumed I already knew how to program in C, whatever that was), these potential mentors assumed I knew a lot more than I did. I didn't want to let on how little I knew, for fear of losing my job (probably not a smart move). However, even in those frustrating early days, I could see how powerful programming would be, and I was even having fun. Eventually, the pieces started to come together, and understanding began to emerge.

My introduction to programming was wrong in so many ways, and I realized that anyone trying to teach themselves how to program would have a similar experience. Several times, I wanted to quit, thinking that programming was just for nerdy computer science students anyway. Starting to learn to program is daunting, and the experienced programmers know so much that they seem too intimidating to ask questions. But despite how hard it can be and how many times you might want to bang your head against your desk or throw your computer across the room, programming can be amazingly fun and rewarding. When I realized how great programming can be, I left a Master's degree in accounting and a job at a top accounting firm to pursue it. And I've never looked back. I wrote this book to be the book I wish I'd had when I started programming.

Why You Should Read This Book

Computers are all around us, in almost every aspect of our lives, yet most of us don't really understand how they work or how to make them work for us. We are limited to what the computer already knows how to do. But computers have always been intended to be programmable machines. You can program the computer that is already on your desk to do whatever you want it to do. As the world moves to relying more on computers, programming skills will become essential for everyone, not just professional software engineers and developers. This book will help you learn to program—and have fun doing it.

In the pages that follow, you will build a foundation in programming that will prepare you to achieve all your programming goals. Whether you want to become a professional software programmer, you want to learn how to more effectively communicate with programmers, or you are just curious about how programming works, this book is a great first step in helping to get you there. Learning to program will still be hard, but it will be possible—and hopefully it can be fun instead of frustrating.

Your Project

The best way to learn how to program is to actually program. Throughout this book, you will be programming a complete Chrome extension. A Chrome extension is a program that enhances (or extends) the functionality of the Chrome web browser. The extension we will be building together will ask a user for a name and phone number, then change all the images in the user's Facebook newsfeed to kittens or puppies, based on the user's location (as determined by the phone number). This extension (which we call kittenbook) might not be particularly useful, but it will be fun to build, and it will help you learn a lot of important programming concepts along the way.

This page intentionally left blank

1

"Hello, World!" Writing Your First Program

Note

Project: Create a Google Chrome Extension.

Welcome to the world of programming! I believe that the best way to learn how to program is to actually program (but you already know that because you totally didn't skip the introduction), so we're going to build a program in this chapter. The program will be the first part of a larger project that we will build throughout this book. Go get in front of your computer if you're not already there—you have some code to write!

One word of warning before we get started: We are going to cover a few things in this chapter that you won't immediately understand. Not understanding what you are doing is (for me at least) a big part of programming computers. By the end of the book, all the mysteries will be revealed. For now, trust me.

Choose a Text Editor

One of the most important parts of programming is writing code (but, as discussed in the sidebar, coding and programming are not exactly the same thing). When I say *code*, I mean instructions for a computer written in a programming language that the computer can understand. Code is written in plain-text files using a text editor. Your text editor is perhaps your most important tool (like Robin Hood's bow, King Arthur's Excalibur, and Susan Boyle's voice). You have many text editors to choose from, so choose wisely.

Programming vs. Coding

The difference between programming and coding might be subtle, but it is significant. Coding is just a part of programming. In fact, coding is probably the *easiest* part of programming. Programming includes such tasks as creating an environment in which your code can successfully be executed, organizing your code in a logical way, testing your code to identify errors, debugging your code to find what is causing those errors, working with code and frameworks that others write, and packaging your code in a way that others can use it. When you know how to program, coding can be a lot more fun.

Tip

My first mistake in learning to program was my choice of text editors. I had previously used only Microsoft Word for putting my words and thoughts into a computer, so I tried programming in Word. It didn’t take long for me to realize that didn’t work. Next, I opened up Notepad and used that for months before I discovered the amazing alternatives I could have been using. Notepad doesn’t include any of the core features every good text editor should have. Learn from my mistakes. For crying out loud, don’t use Notepad for programming.

Core Features

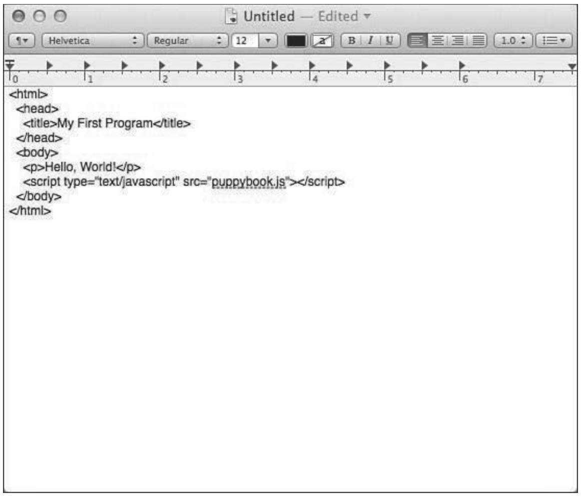
Several text editors are designed specifically for coding. The next section outlines some of the more popular text editors and their most important features. All these editors share a few core features: monospace type font, syntax highlighting, text completion, and extensibility.

Monospace Type Font

Monospace type font is a font style in which every character takes up the same amount of space. In other words, an *i* is as wide as a *w* and also as wide as a space. At first you’ll think the characters look ugly and awkward, but with time, you will come to tolerate, appreciate, then find beauty in the formatting that monospace type font brings to your code.

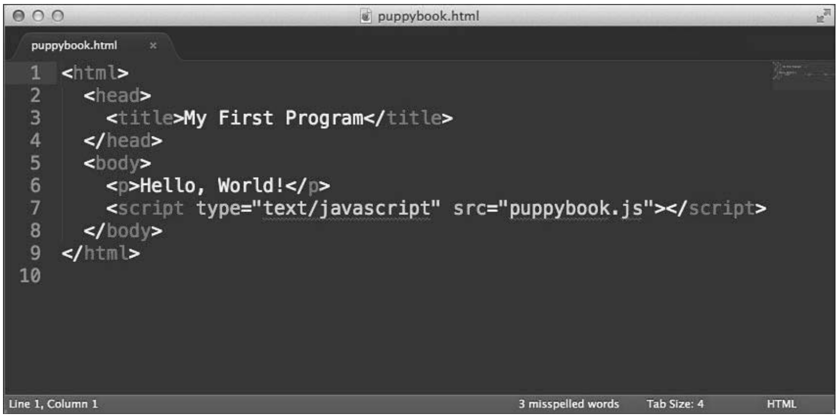
Syntax Highlighting

Just as the English language has nouns, verbs, adjectives, and so on, programming languages are made up of different parts (such as *variables*, *reserved words*, and *strings*). A good text editor differentiates the various parts of programming languages, usually with different text colors. Take a look at Figures 1.1 and 1.2. Even without knowing what the code means, you can see that Figure 1.2 is much easier to look at. Syntax highlighting can also help you find typos in your code.



```
<html>
  <head>
    <title>My First Program</title>
  </head>
  <body>
    <p>Hello, World!</p>
    <script type="text/javascript" src="puppybook.js"></script>
  </body>
</html>
```

Figure 1.1 An HTML document in a nonmonospace type font.



```
<html>
  <head>
    <title>My First Program</title>
  </head>
  <body>
    <p>Hello, World!</p>
    <script type="text/javascript" src="puppybook.js"></script>
  </body>
</html>
```

Figure 1.2 The same HTML document with monospace type font. Is this not much easier to read (and much prettier, too)?

Text Completion

When writing code, you need to type the same words over and over again. For the code to work properly, the words must be typed exactly the same way every time (for example, `myCode` is not the same as `MyCode` or `mycode`). Tiny misspellings and typos are hard to find and can cause major headaches. Every good text editor includes text completion of some sort—similar to the way Google can guess what you're going to search for, your text editor can guess what you're going to type (see Figure 1.3). Using text completion helps you write code faster and can also help you avoid typos and other errors.

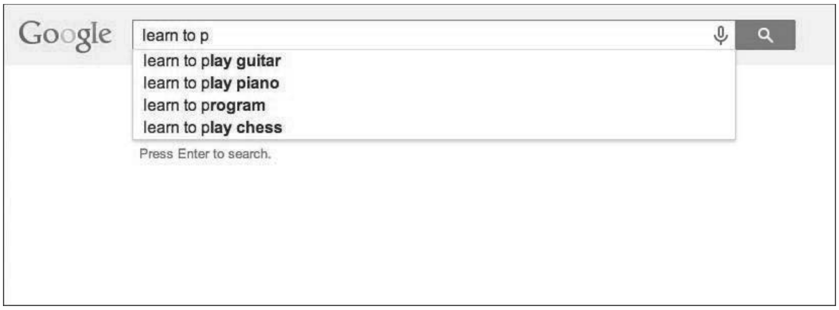


Figure 1.3 Good text editors guess what you want to type, just as Google guesses what you want to search for.

Extensibility

Text editors that are meant for programmers should allow those programmers to build extensions and plug-ins to enhance and modify the behavior of the editor. For example, extensions can modify the appearance of the editor, check code for errors, or quickly add a "snippet" of code to a file. As you will see throughout this book, extensibility is important not only for text editors, but for nearly all software. Almost certainly, there are some features the original creator didn't think of, didn't have time to build, or didn't want. Extensibility makes it possible for those features to be built by anyone who wants to build them.

Making Your Choice

Choosing the best text editor for you depends on your preferences (and your willingness to pay), as well as the project you are working on. I use Vim for most of my coding, but I use IntelliJ when I write Java code, and I use Xcode when I work on iOS apps. The following overview of text editors should help you make your decision.

What Is an IDE?

In the next chapter, you learn about how your computer actually executes the code you write in your text editor. Part of that process involves having an environment that can understand and execute your code. In this book, we use a programming language called JavaScript; the environment that can understand and execute JavaScript is a web browser. For some other programming languages, the text editor is also an environment that understands and executes the code. These special types of text editors are called integrated development environments, but their friends call them IDEs. An IDE is a powerful tool for languages that need an IDE. Your web browser can be just as powerful for writing JavaScript code.

Sublime Text

- Sublime Text has a wide array of plug-ins that can help make you more productive. It is easy to use and has an attractive (dare I say sublime?) user interface.
- It is available on Windows, Mac, and Linux.
- Sublime Text is a good choice for beginners. It is simple and intuitive, and it follows many of the conventions you are familiar with from using a word processor.
- A Sublime Text license costs $70, but you can try it for free.
- Sublime Text is a simple text editor, not a full-fledged IDE.
- You can download Sublime from http://www.sublimetext.com/.

TextMate

- TextMate and Sublime Text have many of the same plug-ins—a plug-in for one usually works with the other. TextMate is easy to learn and easy to use.
- It is available only on Mac.
- As with Sublime Text, TextMate is a good choice for beginners. You can get started and be productive without needing to learn anything new.
- A TextMate license costs $55, but you can try it for free.
- TextMate is a simple text editor, not a full-fledged IDE.
- You can download TextMate from http://macromates.com/.

Notepad++

- Notepad++ is a great option if you are using Windows. The "++" (pronounced "plus plus") comes from the programming language C++; the idea is that C++ is like C, but better—so Notepad++ is like Notepad, but better.
- It is available only on Windows.
- Notepad++ is easy to use and great for beginners.
- Notepad++ is free.
- NotePad++ is a simple text editor, not a full-fledged IDE.
- You can download Notepad++ from http://notepad-plus-plus.org/download/.

Gedit

- Gedit is a good basic editor, but it's not as visually appealing as some of the other editors. If you're using Linux, it is probably already installed.

- It is available on Windows, Mac, and Linux.
- Gedit is easy to use and easy to learn.
- Gedit is free.
- Gedit is a simple text editor, not a full-fledged IDE.
- You can download Gedit from https://wiki.gnome.org/Apps/Gedit/.

Vim

- Vim takes a while to learn, but once you know what you're doing, you can work quickly, with a wide range of shortcuts and plug-ins. Additionally, Vim is compatible with nearly every operating system and is often already installed, so working on an unfamiliar OS is less unfamiliar when you use Vim. I used to be afraid of Vim, but now it's my favorite editor.
- It is available on nearly every operating system ever created (it comes preinstalled on Mac OS X and Linux, but you have to install it yourself on Windows). Vim is like a cockroach: It can survive nearly anywhere, and it will still be around long after the rest of us are gone.
- Vim is not easy to learn and is probably not a great editor for beginners. If you want to learn Vim, though, you can find plenty of resources, including a built-in tutorial (type **`vimtutor`** on the command line).
- Vim is free.
- Vim is a simple text editor, not a full-fledged IDE.
- You can download Vim from www.vim.org/download.php if it's not already installed.

Eclipse

- Eclipse is a full-featured IDE generally used for Java programming. If you are working on a Java project, this is a great choice.
- It is available on Windows, Mac, and Linux.
- Eclipse is not particularly easy for beginners to start using because it is far more than a simple text editor. However, learning to program Java using Eclipse or IntelliJ is much easier than using one of the simple text editors listed previously.
- Eclipse is free.
- Eclipse is an IDE focused on Java development.
- You can download Eclipse from www.eclipse.org/downloads/.

IntelliJ

- IntelliJ is a full-featured IDE for Java that is a bit more lightweight (and, arguably, better looking) than Eclipse. IntelliJ also offers support for other languages, such as Scala and JavaScript.
- It is available on Windows, Mac, and Linux.
- IntelliJ is about as easy to learn as Eclipse.
- IntelliJ is available in a free Community Edition. A license for the Ultimate Edition costs $199.
- IntelliJ is a full-featured IDE.
- You can download IntelliJ from www.jetbrains.com/idea/.

Xcode

- If you are writing an iOS or Mac OS X app, Xcode is the IDE for you. Xcode is an IDE built by Apple for the purpose of building software for Apple platforms.
- It is available only on Mac.
- Xcode is not easy to learn, but Apple has extensive documentation and you can find a lot of community support. If you want to learn to build iPhone apps, you need to learn to use Xcode.
- Xcode is free.
- You can download Xcode from https://developer.apple.com/xcode/ or the Mac App Store.

Visual Studio

- Visual Studio is a full-featured IDE mostly used for .NET development (C#, Visual Basic, and so on), but it can also be used for other languages.
- It is available only on Windows.
- Visual Studio is as easy to learn as any other full-featured IDE—not too easy.
- Visual Studio is available for free as an Express Edition, which is both good and usable. The paid editions of Visual Studio range in price from $1,200 to $13,300.
- You can download or purchase Visual Studio from www.visualstudio.com/downloads/download-visual-studio-vs.

I haven't mentioned all the editors here, but this list should be enough to get you started. For the purposes of this book, I recommend Sublime Text (unless you have experience with one of the other editors). Sublime Text is more than good enough for our purposes and is easy to start using.

Create a Project Directory

Before we start writing our code, we need a place for it to live. Computer programs are usually a collection of multiple files working together, so it is a good idea to group all the files you will need for a project in a single folder (folders are often called directories—I use *folder* and *directory* interchangeably in this book). I have a directory called `projects` where I group all such project directories. Create a directory called `kittenbook` somewhere on your hard drive. You can create a new directory by using your file browser (Finder on Mac, Windows Explorer on Windows, and Nautilus on Linux) and clicking File → New Folder.

Start Small: Create a Test File

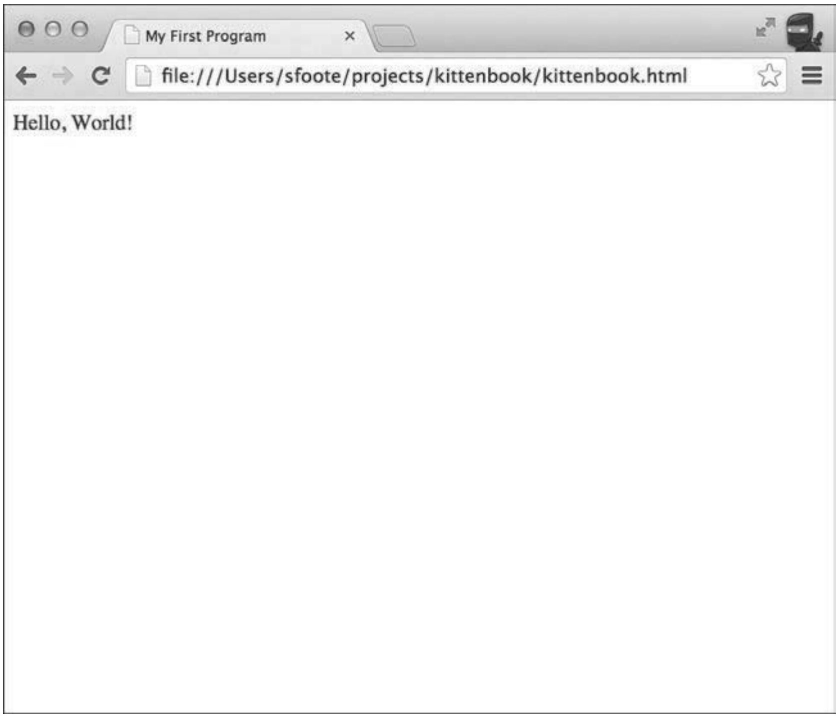
Now that you have your project directory, let's start programming. The first thing you're going to do is create an HTML file in your `kittenbook` directory called `kittenbook.html`. You can create a new file in a few different ways, but the easiest is to open your chosen text editor, create a new file (click File → New File), and then save it as `kittenbook.html` (File → Save As, then find your `kittenbook` directory, type in **`kittenbook.html`** as the filename, and click Save). Now fill in `kittenbook.html` with the code in Listing 1.1.

Listing 1.1 **`kittenbook.html`**

```
<html>
    <head>
        <title>My First Program</title>
    </head>
    <body>
        <p>Hello, World!</p>
    </body>
</html>
```

Save the file and then open it in a web browser by double-clicking it in your file browser. You should see something similar to Figure 1.4.

Take a break to enjoy this moment. Now look back at your code and compare it to what you see in the web browser. You should see `Hello, World!` in the window body, and you should see `My First Program` in the tab. HTML (Hypertext Markup Language) is made up of elements. An element consists of an opening tag, an optional body, and a closing tag. For instance `<p>` is an opening tag for the paragraph element, `Hello, World` is the content, and `</p>` is the closing tag. The body of an element can also contain other elements. For instance, see Listing 1.2.



Hello, World!

Figure 1.4 You told the computer to do something, and it worked!

Listing 1.2 **The `<head>` Tag of `kittenbook.html`**

```
<head>
    <title>My First Program</title>
</head>
```

The `head` element contains a title element. This tells the web browser the title of the HTML page, and most web browsers display that title in the tab. You have plenty more to learn about HTML, but we'll leave it at that for now.

How HTML and JavaScript Work Together in a Browser

You've just created your first web page. Now let's make the page a little more interactive. Create another file in your `kittenbook` directory called `kittenbook.js`. This will be a JavaScript file, and it will make your web page more interesting.

Open `kittenbook.js` with your text editor and fill it with the code in Listing 1.3.

Listing 1.3 **Hello, Friend!**

```
alert('Hello, [your name]!');
```

Replace `[your name]` with your real name. For me, this would be `alert('Hello, Steven!')`.

Now open your web page again (or reload it) and see what happens. It's a trick! The page should look exactly the same as it did before. The HTML doesn't know that the JavaScript exists, and that's because we didn't tell the HTML about the JavaScript. Update `kittenbook.html` by adding the line from Listing 1.4 inside the `<body>` element.

Listing 1.4 **Add Your JavaScript to `kittenbook.html`**

```
<script type="text/javascript" src="kittenbook.js"></script>
```

Now `kittenbook.html` should look like Listing 1.5.

Listing 1.5 **`kittenbook.html` with JavaScript**

```
<html>
    <head>
        <title>My First Program</title>
    </head>
    <body>
        <p>Hello, World!</p>
        <script type="text/javascript" src="kittenbook.js"></script>
    </body>
</html>
```

This time, you should see something similar to Figure 1.5.

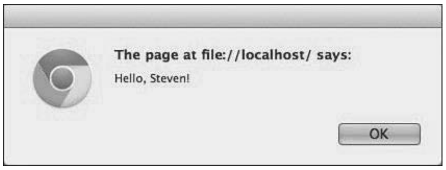

The page at file://localhost/ says:
Hello, Steven!

Figure 1.5 Hello, you wrote a real program!

That was awesome! You made a window appear with your name in it. You could type whatever you wanted, and it would show up in that window. Let's talk about what you just did; then we'll take it to the next level.

The JavaScript you wrote is a single instruction for your computer. `alert` is something called a function that tells the web browser to open a window with an OK button. You can add some text to the window by adding an argument to `alert`. A function is a piece of code that can be called to perform a task. An argument can modify how the function performs the task. The semicolon at the end of the line indicates to JavaScript the end of an instruction. Programs usually have many instructions, so the semicolons separate the instructions.

The Value of Small Changes

Now we're going to enhance our program so that we can say hello to anyone, instead of just "Hello, World!" As you make the enhancements, make small changes. Then test those small changes before you make more changes. At first, this process might seem tedious and unnecessary. However, if you make many changes before testing and find that your program isn't working, you will have a hard time determining which of your changes broke the program. If you make one small change before testing and see that your program is broken, you will know exactly what you did to break it.

Our first small change is to ask for a name. A personalized greeting isn't much good without a name. For this example, we use a function called `prompt` because it is simple and it works well here. However, I beg you not to use `prompt` (or `alert`) with a real website. Although `prompt` and `alert` are convenient for testing and learning, they make for a poor and outdated user experience. We cover alternatives for them in Chapter 8, "Functions and Methods." For now, update your `kittenbook.js` to use `prompt` instead of `alert`, and change the message to say "Hello, what's your name?" (see Listing 1.6).

Listing 1.6 **Prompt for a Name**

```
prompt('Hello, what\'s your name?');
```

You might be wondering why this code says `what\'s` instead of `what's`. A group of characters such as `'Hello, World!'` is called a string. A string starts with a quote and ends with a quote. If I type `prompt('Hello, what's your name?');`, then I have three quotes (even though one of them is intended to be an apostrophe), which confuses the computer. The computer sees the string `'Hello, what'`, then sees some other characters that it doesn't understand, and then sees another string: `');'`. I use a backslash before the single quote in `what\'s` to tell the computer that I don't mean the apostrophe to signify the end of the string. The backslash is called an escape character.

Now if you refresh your page, you should see a window with a place for you to type your name (see Figure 1.6).

Notice that when you type in your name and click OK, the page still says `Hello, World!` What gives? Why ask for a name if you're not going to show it? Well, we haven't told the computer what to do with the name. We made a small change: We switched the `alert` window for a `prompt` window. It worked, so now we can tell the computer what to do with the name.

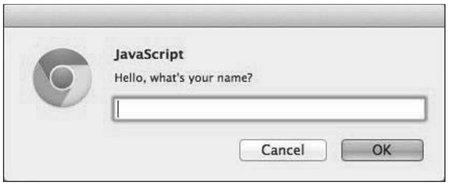


Figure 1.6 What's your name?

One of the most important tools in programming is the variable. A variable is a place to store some data that affects how the program works—Chapter 5, "Data (Types), Data (Structures), Data(bases)," discusses variables in great detail. In this case, we want to store the name from the prompt window. (See Listing 1.7.)

Listing 1.7 **Your First Variable**

```
var userName = prompt('Hello, what\'s your name?');
```

All variables have names, and we have called our variable `userName`. To tell the computer that we are creating a variable, we use a reserved word: `var`. A reserved word is a word that has special meaning to a programming language. It is reserved, so you cannot use that word as the name of a variable; you don't want to confuse your computer.

Now that we have stored our variable, refresh the page to see that everything still works. The web page should look the same as the last time you refreshed. We have stored the username, but we haven't told the computer to do anything with it yet. The next step is to insert the username in the web page (see Listing 1.8).

Listing 1.8 **Put Your Variable to Work**

```
var userName = prompt('Hello, what\'s your name?');
document.body.innerHTML = 'Hello, ' + userName + '!';
```

We add a new instruction to change the HTML `document`'s `body`. Remember the `<body>` element from `kittenbook.html`? `document.body.innerHTML` refers to everything between the opening tag and the closing tag of the `<body>` element. `document.body.innerHTML = 'Hello, ' + userName + '!';` effectively changes the HTML to look like Listing 1.9 (assuming that I type `Steven` when the prompt asks for my name).

Listing 1.9 **kittenbook.html When Your JavaScript Is Done with It**

```
<html>
    <head>
        <title>My First Program</title>
    </head>
    <body>
```

```
        Hello, Steven!
    </body>
</html>
```

Congratulations! You just created a real program. Admittedly, this is not the most useful program that has ever been written, but it is certainly not the *least* useful—and you learned a lot along the way.

Build on Your Success

Now we take the program that you just built and turn it into a Chrome extension. A Chrome extension is a small program that can be installed in Google Chrome to enhance the user's Chrome experience. A Chrome extension can be powerful; some companies' main product is a Chrome extension. The extension that we start in this chapter serves as the basis of the project for the rest of the book.

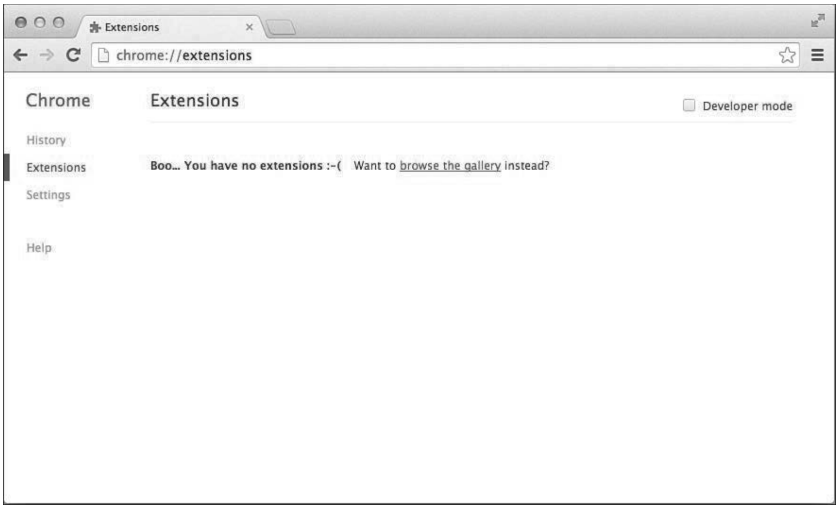
The first step is to create a new file called `manifest.json`. This file gives Chrome information about the extension and how it works. JSON is a type of document that stores data in an efficient way and that both computers and humans can easily read. Fill in your `manifest.json` file to look like Listing 1.10 (remember that every character is important, including the funny-looking characters called curly braces on the first and last lines).

Listing 1.10 **Example `manifest.json` for a Chrome Extension**

```
{
    "manifest_version": 2,
    "name": "kittenbook",
    "description": "Replace photos on Facebook with kittens",
    "version": "0.0.1"
}
```

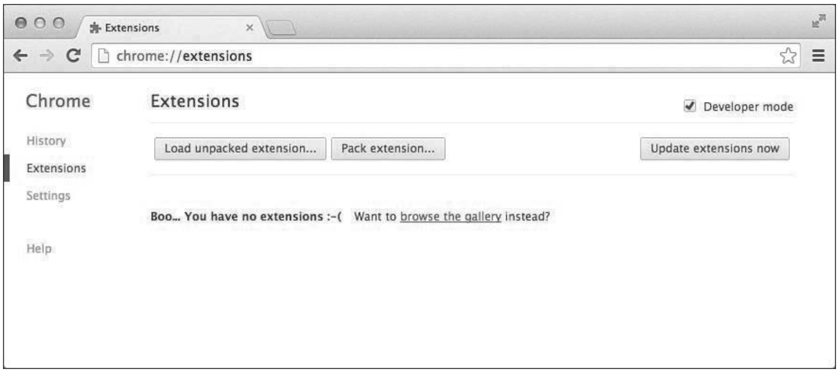
Now that you have a `manifest.json` file, you have everything you need to create a Chrome extension. At this point, your extension won't do anything, but you can still add it to Chrome to make sure your `manifest.json` file doesn't have any problems. To add your extension to Chrome, you need to open the Chrome web browser (download it if you haven't already) and then enter **`chrome://extensions`** in the address bar. You should see something similar to Figure 1.7.

Click the checkbox next to Developer Mode (because you're a developer now!) to be able to add your extension. Now you should see a button that reads Load Unpacked Extension (see Figure 1.8).



Boo... You have no extensions :-(Want to browse the gallery instead?

Figure 1.7 The Chrome Extensions page

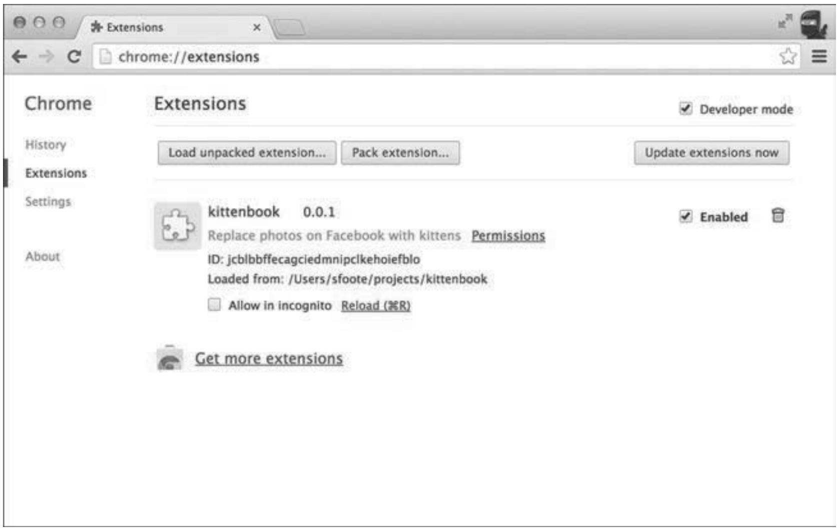


Boo... You have no extensions :-(Want to browse the gallery instead?

Figure 1.8 The Chrome Extensions page in Developer mode

When you click that button, you want to select your entire project directory (instead of a single file within the project directory). If you have selected the correct directory and the `manifest.json` file has no problems, you should see your extension added to the list of extensions (see Figure 1.9).

If your `manifest.json` file has a problem, you get a message like the one in Figure 1.10. If you do see such a message, you should copy the contents of `manifest.json` and paste them into a JSON validator such as http://jsonlint.com/, which shows you just where your error is and how to fix it.



Extensions

✓ Developer mode

kittenbook 0.0.1

Replace photos on Facebook with kittens Permissions

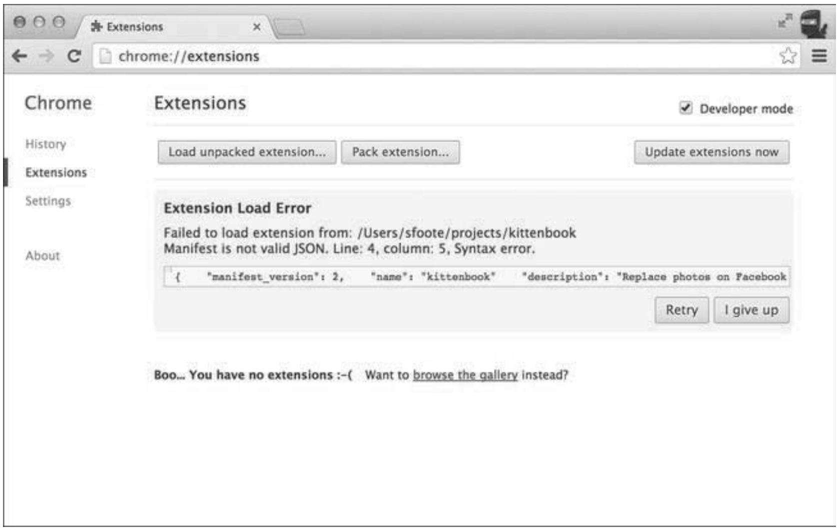
ID: jcblbbffecagciedmnipclkehoiefblo

Loaded from: /Users/sfoote/projects/kittenbook

☐ Allow in incognito Reload (⌘R)

✓ Enabled

Figure 1.9 Kittenbook making its debut in Chrome



Extensions

✓ Developer mode

Extension Load Error

Failed to load extension from: /Users/sfoote/projects/kittenbook
Manifest is not valid JSON. Line: 4, column: 5, Syntax error.

```
{    "manifest_version": 2,    "name": "kittenbook"    "description": "Replace photos on Facebook
```

Boo... You have no extensions :-(Want to browse the gallery instead?

Figure 1.10 When `manifest.json` is busted

Reference Your JavaScript in `manifest.json`

After the kittenbook extension is successfully installed, you are ready to make it actually do something. You already have a JavaScript file written, so all you need to do is make that JavaScript available on Facebook.com. We ran into this problem earlier when we tried to add the JavaScript file to `kittenbook.html`. To solve that problem, we had to tell `kittenbook.html` about the JavaScript file. In this case, we need to tell `manifest.json` about the JavaScript file. We also need to tell `manifest.json` the website to which our JavaScript file should be added. We can do this by using a property called `content_scripts` (see Listing 1.11). Content scripts are JavaScript files that should be added to the content of a given web page or set of web pages.

Listing 1.11 **`manifest.json` with Content Scripts**

```
{
    "manifest_version": 2,
    "name": "kittenbook",
    "description": "Replace photos on Facebook with kittens",
    "version": "0.0.1",
    "content_scripts": [
        {
            "matches": ["*://www.facebook.com/*"],
            "js": ["kittenbook.js"]
        }
    ]
}
```

With the addition in Listing 1.11, our extension now adds `kittenbook.js ("js": ["kittenbook.js"])` to every page that contains www.facebook.com in the URL (`"matches": ["*://www.facebook.com/*"]`). For your extension to pick up the changes you made to `manifest.json`, you need to reload your extension, which you can do by clicking the Reload link in Figure 1.9.

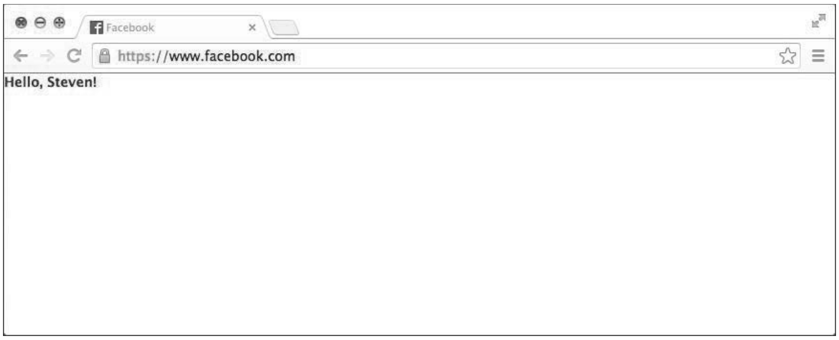
Let It Run!

If your extension has successfully reloaded, you can make one last change to `kittenbook.js` to make the greeting look a little nicer (see Listing 1.12).

Listing 1.12 **Updated `kittenbook.js` for Use in the Chrome Extension**

```
var userName = prompt('Hello, what\'s your name?');
document.body.innerHTML = '<h1>Hello, ' + userName + '!</h1>';
```

Adding the `<h1>` tag styles the greeting as a heading (big and bold text). Now reload the extension and open Facebook to see what happens. You should see your prompt window open and then see something similar to Figure 1.11.



Hello, Steven!

Figure 1.11 Hello, Facebook!

That was amazing—you just changed Facebook. *The* Facebook is showing *your* greeting. That is really cool but not useful at all. In fact, the extension we just built will prevent you from seeing any real Facebook page. That's really annoying. While I was building the extension, my wife tried to check Facebook on my computer; she was not particularly impressed with the greeting that took over her Facebook page. When you're not working on your extension, you can disable it by unchecking the Enabled check box (refer to Figure 1.9).

Great Power, Great Responsibility

By reading this chapter, you have started to acquire remarkable power. You are beginning to learn that you can give your computer instructions and that your computer will do what you say. Software is no longer something you just buy; you can *make* it.

The Chrome Extension that we started in this chapter will be a great tool to help you learn about many different concepts in programming. However, in its current state, the extension is a nuisance. It would be unkind to share it with friends (especially if you don't tell them how to disable it). You have power to create useful, helpful, and fun programs, but you also have the power to create annoying, harmful, and malicious programs. Always use your power for good.

Summing Up

In this chapter you learned about:

- Text editors, the place where computer programs are created
- Project directories to keep your program organized
- HTML and JavaScript, and how they work together
- Small iterations for finding problems early
- Variables and escape characters

Oh, and you built a real, functioning program. Then you turned that program into a Chrome Extension that actually changes Facebook.com. Just think, 40 minutes ago, you didn't even know what a variable was. You should be proud—and probably a bit tired. Now might be a good time for a break.

In the next chapter (after your break), you will learn about:

- How software works
- Compiled software
- Interpreted software
- Input and output
- Memory and variables

This page intentionally left blank

Index

Symbols

&& (and) operator (JavaScript), 133
\ (backslash)
  JavaScript, 15
  regular expressions, 112-113, 119
    \B, 119
    \b, 119
    \D, 119
    \d, 119
  limitations of, 121
    \n, 119
    \r, 119
    \s, 119
    \t, 119
    \W, 119
    \w, 119
[ ] (brackets), regular expressions, 115
    list of acceptable characters, 115
    ranges, 115-116
    ^ (caret), regular expressions, 116
{ } (curly braces), regular expressions, 113
/ (division) operator, 136
== (equals) operator, 131
=== (equals) operator (JavaScript), 131
> (greater than) operator, 132
>= (greater than or equal to) operator, 132
< (less than) operator, 132
<= (less than or equal to) operator, 132
% (modulo) operator, 136
! (not) operator (JavaScript), 134
!== (not equals) operator, 132
|| (or) operator (JavaScript), 133

( ) (parentheses), capture groups, 121-122
advanced find/replace, 125-126
capture HTML tag example, 124-125
+ (plus sign)
addition operator, 136
regular expressions, 111-112
? (question mark), regular expressions, 111, 121-122
*** (star)**
multiplication operator, 136
regular expressions, 112
- (subtraction) operator, 136

A

absolute (full) paths, 56-57
abstractions, 48
accounts and permissions, 51
adding
files to repositories via Git, 261-262
items to arrays, 93
Analytical Engine, 24
and (&&) operator, 133
anonymous functions, 165-166
answering questions (error-handling), 241-242
architectures
design patterns, 268
Mediator design pattern, 269
Pub Sub design pattern, 268-269
Singleton design pattern, 270
planning, 200-201
breakdowns, exploring, 202-203
draw diagrams, 201-202
weaknesses, exploring, 202-203
arguments, 155
multiple arguments in functions, 156
parameters, 155-156
arrays, 90-91
adding items to arrays, 93
associative array. *See* dictionary
forEach method and for loops, 144-145
image arrays, 93-94
for loops and, 141-145
"month names" example, 92
node.js and, 92
pop method, 93
push method, 93
removing items from arrays, 93
sets versus arrays, 99
stacks, 99-100
storing data in, 98
trees, 100-101
assigning
binary operators, 135
functions to variables, 164
associative array. *See* dictionary
autocompleting commands in command line, 57
automated testing, 211
failures, 218-219
functions, 219
kittenbook, 215-218
unit tests, 212-217
automating tasks, 63-64, 70-71
compiling, 71
concatenating files, 69, 74-75
copying files, 75
deploying, 72
error-checking files, 75-76
Grunt
Gruntfile.js, 73
installing, 66-69
loading plug-ins, 76
plug-ins, 73-76
JSHint, 75-76
mistakes, avoiding, 70
node.js, installing, 64-66
NPM, installing, 66
packaging, 72
packaging extensions, 78
registering tasks, 76-78
software creating software, 69
testing, 71-72
watching tasks, 79-80
working faster as a result of, 70
\B, 119
\b, 119
\D, 119
\d, 119
\n, 119
\r, 119
\s, 119
\t, 119
\W, 119
\w, 119

B

Babbage, Charles, 24
backslash (\\)
- JavaScript, 15
- regular expressions, 112-113, 119
- limitations of, 121

backward, searching, 236
balance, designing for, 206
- capabilities and limitations, designing to, 206-207
- constraints, identifying and creating, 206
- limitations and capabilities, designing to, 206-207
- permissions, 207

Bash, 27
basic commands and command line, 54
Bernoulli numbers, 24
best practices in coding
- documentation, 182-183
- planning, 183
- testing, 183

binary operators, 134
- % (modulo) operator, 136
- assigning, 135
- coercion, 134-135
- compiled languages, 31
  - *interpreted languages, 31*
  - *source code's transformation to binary, 31*
- math operators, 136
- multiple operator assignments, 135
- order of operation, 135

blogs
- as a programming resource, 237
- writing, 241-242
  - *list of acceptable characters, 115*
  - *ranges, 115-116*

Boolean data types, 84-87
brackets ([ ]), regular expressions, 115
branches in Git, 265
break statements
- infinite loops, 146
- for loops, 143-144
- while loops, 146

breakdowns, exploring during planning, 202-203
breakpoints and debugging, 224-226
browsers, HTML and JavaScript functionality, 13-15
bugs
- debugging, 222-223
  - *breakpoints, 224-226*
  - *call stacks, 231*
  - *code inspection via Console (Chrome Dev Tools), 228-231*
  - *errors, 223-224*
  - *logs, 224*
  - *repetition, importance of, 232*
  - *root causes, determining, 231*
  - *"stepping through" code, 229-231*
- finding via documentation, 191

build tools, 63-64
- automating tasks, 70-72
  - *compiling, 71*
  - *concatenating files, 74-75*
  - *copying files, 75*
  - *error-checking files, 75-76*
  - *JSHint, 75-76*
  - *loading plug-ins, 76*
  - *packaging, 72*
  - *packaging extensions, 78*
  - *registering tasks, 76-78*
  - *testing, 71-72*
  - *watching tasks, 79-80*
- concatenating files, 69, 74-75
- copying files, 75
- error-checking files, 75-76
- Grunt
  - *Gruntfile.js, 73*
  - *installing, 66-69*
  - *loading plug-ins, 76*
  - *plug-ins, 73-76*
- JSHint, 75-76
- mistakes, avoiding, 70
- node.js, installing, 64-66
- NPM, installing, 66
- packaging extensions, 78
- registering tasks, 76-78
- software creating software, 69
- watching tasks, 79-80
- working faster as a result of, 70

C

C, 27
C++, 27
C#, 27-28
call stacks, 231
callbacks, 165-166
calling functions. *See* invoking, functions
capabilities and limitations, designing to, 206-207
capture groups, 122-124
  advanced find/replace, 125-126
  capture HTML tag example, 124-125
caret (^), regular expressions, 116
cat (type or copy) command, concatenating files, 60-61
catch/try, 150-151
certifications
  Coursera, 252-253
  Udacity, 252-253
character data types, 88
choosing data structures, 104
Chrome extension
  adding to Chrome, 17-19
  Chrome Extensions page, Developer mode, 18
  Dev Tools, 85-86
    breakpoints, 224-226
    Console and code inspection, 228-231
    image arrays, 93-94
    viewing call stacks, 231
  disabling, 21
  manifest.json files, 17
    JavaScript references, 20
    troubleshooting, 19
  updating in Chrome, 20
classes in OOP, 266
Clock, input (source code) from, 35
cloning repositories and Git, 261
codeacademy, 251
coding
  architectures, 200-201
    breakdowns, exploring, 202-203
    draw diagrams, 201-202
    weaknesses, exploring, 202-203
  automated testing, 211
    failures, 218-219
    functions, 217
    kittenbook, 215-218
    "spying" on functions, 219-220
    unit tests, 212-217
  balance, designing for, 206
    capabilities and limitations, designing to, 206-207
    constraints, identifying and creating, 206
    limitations and capabilities, designing to, 206-207
    pemissions, 207
  best practices, 182
    documentation, 182-183
    planning, 183
    testing, 183
  communal builds, 181-182
  computers' understanding of code, 177
  conventions, establishing, 173-174, 178-180
    "hacky" code, 174-175
    high-quality code, importance of, 174-175
    "maintainable" code, writing, 175-176
    price of poor coding, 175
    standards, 174, 178-180
    "tech debt"175
  debugging, 222-223
    breakpoints, 224-226
    call stacks, 231
    code inspection via Console (Chrome Dev Tools), 228-231
    errors, 223-224
    logs, 224
    repetition, importance of, 232
    root causes, determining, 231
    "stepping through" code, 229-231
  design step, 199-200
  documentation,
    audience, writing for an, 196-197
    best practices, 182-183
    bugs, finding, 191
    comments, usefulness of, 192-193
    decision-making process, documenting, 196-197
    diagrams, 192-193

Grunt, 194
JSDoc, 194
kittenbook, 192-194
learning via documentation, 192
Markdown files, 193
memory versus documentation, 191-192
obvious, documenting the, 189
outdated documentation, 190-191
resources, documenting, 197
self-documenting code, 187-188
"sneaky" code, 188
teaching via documentation, 197
encapsulating code, 158
dividing jobs per function, 159-162
one job per function, 158-159
formatting code, 176-177
consistency in coding, 177
whitespace in code, 178
inspecting code via Console (Chrome Dev Tools), 228-231
"maintainable" code, 175-176
open source software, benefits from, 181
communal builds, 181-182
faster builds, 180-181
solo builds, recognizing the need for, 182
outdated documentation, 190-191
permissions, 207
planning, 199
architectures, 200-203
balance, 206-207
best practices, 183
breakdowns, exploring, 202-203
design step, 199-200
draw diagrams, 201-202
extensibility, designing for, 203
iterative planning, 203
priorities, 204-205
specifications, 200, 216-217
weaknesses, exploring, 202-203
priorities, 204
deadlines, 205
performance, 204-205
scalability, 205
security, 205
user experience, 204
programming vs, 5
refactoring code and regular expressions, 127
referencing your personal code when asking questions (error-handling), 241
reusing code, 163
DRY concept, 165-166
general problem solving, 163
generalized functions, 163-164
reviewing, 260
self-documenting code, 187-188
"sneaky" code, 188
solo builds, recognizing the need for, 182
"spaghetti" code, 203
specifications, 216-217
"stepping through" code, 229-231
testing
automated testing, 211-220
best practices, 183
"crazy" changes, making while testing, 210-211
"dogfooding"211
integration tests, 221-222
Jasmine testing library, 215-220
manual testing, 209-211
repetition, importance of, 232
test-driven development, 220
testing while working, 210
unit tests, 212-217
whitespace in code, 178
coercion, 134-135
collaboration
open source projects and, 247-249
version control and, 256-260
command line, 52-54
absolute (full) paths, 56-57
autocompleting commands, 57
basic commands, 54
cat (type or copy) command, 60-61
cp (copy) command, 58
directories
Home directory access, 56
moving up, 56
returning to previous directory, 56
grep (findstr) command
finding text within files, 61-62
regular expressions and, 127

Grunt, installing, 66-69
ls (dir) command, 57-58
mkdir (md) command, creating directories, 59-60
mv (move) command
moving files, 58
renaming files, 58-59
NPM, installing, 66
prompt, 54-56
relative (partial) paths, 56-57
rm (del) command, 59
deleting directories, 59
deleting files, 59
terminal, opening, 54

commands
basic commands and command line, 54
cat (type or copy) command, 60-61
cp (copy) command, 58
grep (findstr) command
finding text within files, 61-62
regular expressions and, 127
ls (dir) command, 57-58
mkdir (md) command, creating directories, 59-60
mv (move) command
moving files, 58
renaming files, 58-59
rm (del) command, 59
deleting directories, 59
deleting files, 59

comments, 41-42
audience, writing for an, 196
decision-making process, documenting, 196-197
resources, documenting, 197
teaching via documentation, 197
bugs, finding, 191
diagrams and documentation, 192-193
Grunt documentation, 194
JSDoc documentation, 194
kittenbook, 192-194
learning via documentation, 192
Markdown files, 193
memory versus documentation, 191-192
obvious, documenting the, 189
outdated documentation, 190-191
self-documenting code, 187-188
"sneaky" code, 188
usefulness of, 192-193

changes to files via Git, 262

committing communal builds, 181-182

comparison operators, 131
== (equals) operator, 131
=== (equals) operator (JavaScript), 131
> (greater than) operator, 132
>= (greater than or equal to) operator, 132
< (less than) operator, 132
<= (less than or equal to) operator, 132
!== (not equals) operator, 132

compiling
automating, 71
languages, 31

completing text via text editors, 7-8

complex relative paths, 57

composite data types, 89
[ ] (brackets), 115-116
^ (caret), 116
+ (plus sign), 111-112
* (star), 112
arrays, 90-93
adding items to arrays, 93
associative array. See dictionary
forEach method and for loops, 144-145
image arrays, 93-94
for loops and, 142-145
"month names" example, 92
node.js and, 92
pop method, 93
push method, 93
removing items from arrays, 93
sets versus arrays, 99
stacks, 99-100
storing data in, 98
trees, 100-101
dates, 90
dictionary, 94-95
objects
storing data in, 98
trees, 101
regular expressions, 90, 111-114
\ (backslash), 112-113, 119-121
{ } (curly braces), 113

? (question mark), 111
(?:) groups, 121-122
capture groups, 122-126
code refactoring, 127
data extraction, 128
defining, 109-110
escaping, 112-113
as Find feature, 109-110
flags, 126-127
global flags, 126
greed and, 114
grep (findstr) command, 127
ignore case flags, 126
JavaScript, 110
multiline flags, 127
negation in, 116
pattern-matching, 113-114
phone number validation project, 116-119, 120-124
regexpal, 116
Regexper visualization tool, 114
shortcuts, 119-121
syntax of, 110
testing, 110
uses for, 127-128
validation, 128
visualizing, 114
strings, 89-90
creating in Java, 96
creating in JavaScript, 96
storing key-value pairs, 97-98
URL, 97-98
computers
components of, 48-49
CPU, 49
evolution of, 48
file systems, 50-51
command line, 52-56
directories, 51
hidden files, 52
permissions, 51
symbolic links, 51-52
hard drives, 49
instructions, executing, 47-48
magic of, 48
memory
hard drives, 49
long-term memory, 49
short-term memory, 49
processors and CPU, 48-49
RAM, 49
stupidity of, 47-48
automating tasks, 69
cat (type or copy) command, 60-61
Grunt plug-ins, 74-75
concatenating files
conflict resolution and version control, 258-260
consistency in coding, 177
constraints, identifying and creating, 206
contributing to open source projects, 247-249
converting strings to numbers via parseInt, 234
copying
directories via cp (copy) command, 58
files
cp (copy) command, 58
Grunt plug-ins, 75
Coursera, 250-253
cp (copy) command, 58
CPU (Central Processing Unit), 48-49
"crazy" changes, making while testing, 210-211
cron jobs, 147-148
CRUD (Create, Read, Update, Delete) functions, 106
curly braces ({ }), regular expressions, 113

D

data extraction and regular expressions, 128
data structures, 96-97
arrays
sets versus arrays, 99
storing data in, 98
choosing, 104
graphs, 101-104
objects, storing data in, 98

sets, 99
stacks, 99-100
strings, 97-98
trees, 100-101

data types, 83-84
arrays, 90-93
adding items to arrays, 93
associative array. See dictionary
forEach method and for loops, 144-145
image arrays, 93-94
for loops and, 142-145
"month names" example, 92
node.js and, 92
pop method, 93
push method, 93
removing items from arrays, 93
sets versus arrays, 99
stacks, 99-100
storing data in, 98
trees, 100-101
Boolean data types, 84-87
character data types, 88
composite data types, 89-95
dates, 90
dictionary, 94-95
different data types, need for, 84
dynamically/statically typed languages, 96
mixing, 84
number data types, 88-89
objects
storing data in, 98
trees, 101
primitive data types, 84-88
regular expressions, 90, 111-114
\ (backslash), 112-113, 119-121
[ ] (brackets), 115-116
^ (caret), 116
{ } (curly braces), 113
** (star), 112*
+ (plus sign), 111-112
? (question mark), 111
(?:) groups, 121-122
** (star), 112*
capture groups, 122-126
code refactoring, 127
data extraction, 128
defining, 109-110
escaping, 112-113
as Find feature, 109-110
flags, 126-127
global flags, 126
greed and, 114
grep (findstr) command, 127
ignore case flags, 126
JavaScript, 110
multiline flags, 127
negation in, 116
pattern-matching, 113-114
phone number validation project, 116-124
regexpal, 116
Regexper visualization tool, 114
shortcuts, 119-121
syntax of, 110
testing, 110
uses for, 127-128
validation, 128
visualizing, 114
statically/dynamically typed languages, 96
strings, 89-90
creating in Java, 96
creating in JavaScript, 96
storing key-value pairs, 97-98
URL, 97-98

databases, 104
CRUD functions, 106
long term (persistent) storage, 104
relational databases, 104
many-to-many relationships, 106
one-to-many relationships, 106
one-to-one relationships, 106
relationships (overview of), 105-106
SQL, 106-107
tables, 105

dates, 90

deadlines, planning for, 205

debugging, 222-223
breakpoints, 224-226
call stacks, 231
code inspection via Console (Chrome Dev Tools), 228-231
errors, 223-224
kittenbook, 223-224

logs, 224
repetition, importance of, 232
root causes, determining, 231
"stepping through" code, 229-231
decimal points. *See* floats
decision-making process, documenting, 196-197
defensive coding, 37
definitions (functions), 154
del command, 59
deleting
directories, 59
files, 59
items from arrays, 93
dependencies, 68
deploying, automating deployments, 72
describe function, testing, 217
design patterns, 268
Mediator design pattern, 269
Pub Sub design pattern, 268-269
Singleton design pattern, 270
design step, role in programming, 199-200
Dev Tools (Chrome), 85-86
breakpoints, 224-226
call stacks, viewing, 231
code inspection, 228-231
image arrays, 93-94
Developer mode (Chrome Extensions page), 18
diagrams
documentation and, 192-193
draw diagrams, 201-202
dictionary, 94-95
dir (ls) command. *See* ls (dir) command
directories, 51
copying, 58
creating, 59-60
deleting, 59
files
concatenating, 60-61
deleting, 59
finding text within files, 61-62
moving, 58
renaming, 58-59
Home directory, accessing from command line, 56
kittenbook directories
creating, 12
test files, 12-13
listing, 57-58
ls (dir) command, 57-58
moving up, 56
previous directory, returning to, 56
project directories, creating, 12
test files
creating, 12-13
head tags, 13
HTML functionality and JavaScript, 13-15
JavaScript functionality and HTML, 13-15
name prompt, 15-16
disabling Chrome extension, 21
division (/) operator, 136
documentation,
audience, writing for an, 196
decision-making process, documenting, 196-197
resources, documenting, 197
teaching via documentation, 197
best practices, 182-183
bugs, finding, 191
comments, usefulness of, 192-193
diagrams, 192-193
Grunt, 194
JSDoc, 194
kittenbook, 192-194
learning via documentation, 192
Markdown files, 193
memory versus documentation, 191-192
obvious, documenting the, 189
outdated documentation, 190-191
self-documenting code, 187-188
"sneaky" code, 188
choosing, 5-11
core features, 6-8
Eclipse text editor, 10
extensibility, 8
Gedit text editor, 9-10
IntelliJ text editor, 11
monospace type font, 6
Notepad++ text editor, 9
Sublime Text text editor, 9-11

syntax highlighting, 6
text completion, 7-8
TextMate text editor, 9
Vim text editor, 10
Visual Studio text editor, 11
Xcode text editor, 11
"dogfooding" and testing, 211
draw diagrams, 201-202
DRY (Don't Repeat Yourself) concept, 165-166
dynamically/statically typed languages, 96

E

Eclipse text editor, 10
editing and text editors
education, furthering
books, 252
codeacademy, 251
Coursera, 250-253
free online education, 249
codeacademy, 251
Coursera, 250-251
Google, 249
Khan Academy, 251
Project Euler, 249
tutorials, 251
Udacity, 250
Google, 249
Khan Academy, 251
paid education, 252
books, 252
Coursera, 252-253
Treehouse, 253
Udacity, 252-253
Project Euler, 249
Treehouse, 253
tutorials, 251
Udacity, 250-253
else if statements, 138
encapsulating code, 158
dividing jobs per function, 159-162
one job per function, 158-159
equals (==) operator, 131
equals (===) operator (JavaScript), 131
error-handling
debugging, 223-224
deciphering errors, 236
JSHint, checking files via, 75-76
parseInt as a search tool, 234
questions
answering, 241-242
phrasing a question, 241
referencing your personal code, 241
resources
blogs, 237
forums, 239
friends, 237-238
StackOverflow, 238
searching for information
blogs, 237
deciphering error messages, 236
"leveling up," 235-236
parseInt as a search tool, 234
right terms, finding, 235-236
right terms, finding when searching, 235
searching backward, 236
"stuffed duck, the," 240
teaching and, 241
try/catch, 150-151
when to ask questions, determining, 240
escape characters, 15
escaping regular expressions, 112-113
events, 147-148
expect function, testing, 217
extensibility
planning for, 203
text editors and, 8
extracting data and regular expressions, 128

F

Facebook
functionality, adding, 244
restyling, 243-244
failures and testing, 218-219
false/true and if statements, 139
file systems, 50-51
command line, 52-54
absolute (full) paths, 56-57

autocompleting commands, 57
basic commands, 54
cp (copy) command, 58
Home directory access, 56
ls (dir) command, 57-58
moving up directories, 56
mv (move) command, 58-59
opening terminal, 54
prompt, 54-56
relative (partial) paths, 56-57
returning to previous directory, 56
directories, 51
cp (copy) command, 58
Home directory access, 56
listing, 57-58
ls (dir) command, 57-58
moving up, 56
returning to previous directory, 56
hidden files, 52
locating files
absolute (full) paths, 56
relative (partial) paths, 56-57
permissions, 51
symbolic links, 51-52

files
concatenating
cat (type or copy) command, 60-61
Grunt plug-ins, 74-75
copying
cp (copy) command, 58
Grunt plug-ins, 75
deleting, 59
error-checking via JSHint, 75-76
finding text within files, 61-62
hidden files, 52
locating in file systems
absolute (full) paths, 56
relative (partial) paths, 56
moving, 58
renaming, 58-59
test files
creating, 12-13
head tags, 13
HTML functionality and JavaScript, 13-15
JavaScript functionality and HTML, 13-15
name prompt, 15-16

finding
advanced find/replace via capture groups, 125-126
bugs via documentation, 191
dependencies, 68
files in file systems
absolute (full) paths, 56
relative (partial) paths, 56
open source projects, 248
projects (new), 246-247
regular expressions as Find feature, 109-110
right terms when searching, 235
text within files, 61-62
global flags, 126
ignore case flags, 126
multiline flags, 127

findstr command. *See* grep (findstr) command

flags, regular expressions, 126

floats, 88-89

flow control, 131
catch/try, 150-151
cron jobs, 148
else if statements, 138
events, 147-148
if statements, 137-138
else if statements, 138
true/false and if statements, 139
loops
infinite loops, 145-146
for loops, 141-145
nested loops, 143
return statements, 157
while loops, 145-147
operators, 131
binary operators, 134-136
coercion, 134-135
comparison operators, 131-132
increment operator, 134
logical operators, 132-134
math operators, 136
ternary operators, 136-137
unary operators, 134

switch statements, 139-140
"syntatic sugar," 140
timeouts, 149
try/catch, 150-151
when actions, 147
cron jobs, 148
events, 147-148
timeouts, 149
folders. *See* directories
for loops, 141
arrays and, 142
break statements, 143-144
forEach method, 144-145
nested loops, 143
pictures, looping through, 142
whitespace in code, 178
forEach method and for loops, 144-145
formatting code, 176-177
consistency in coding, 177
whitespace in code, 178
forums as an error-handling resource, 239
frameworks (software), importance of understanding, 26
free online education, 249
codeacademy, 251
Coursera, 250-251
Google, 249
Khan Academy, 251
Project Euler, 249
tutorials, 251
Udacity, 250
friends as error-handling resources, 237-238
full paths. *See* absolute (full) paths
functionality, adding to Facebook, 244
functions, 153
anonymous functions, 165-166
arguments, 155
multiple arguments in functions, 156
parameters, 155-156
assigning to variables, 164
call stacks, 231
callbacks, 165-166
code encapsulation, 158
dividing jobs per function, 159-162
one job per function, 158-159
CRUD functions, 106
definitions, 154
describe function, testing, 217
expect function, testing, 217
generalized functions and code reuse, 163-164
IIFE, 169
invoking, 154-157, 169
it function, testing, 217
naming, 159
prompt function, 156
return statements, 156-157
reusing, 163
DRY concept, 165-166
general problem solving, 163
generalized functions, 163-164
scope, 166-167
global variables, 167-168
local variables, 168
lookups, 168-169, 170-171
restricting, 169
software, input, 34-35
"spying" on functions, 219-220
syntax of, 153-154
testing, 217

G

Gedit text editor, 9-10
GIGO (Garbage In, Garbage Out) programming, 37
defensive coding, 37
sanitization, 37-38
validation, 37-38
Git version control system, 260-261
adding files to repositories, 261-262
branches, 265
cloning repositories, 261
committing changes to files, 262
kittenbook, 265
push/pull in, 263-264
tracking changes to files, 261-262
workflow of, 264
GitHub and open source projects, 247-248
global flags, regular expressions, 126

global variables, 167-168
Google
  education, furthering via Google, 249
  right terms, finding when searching, 235
graphs, 101-104
greater than (>) operator, 132
greater than or equal to (>=) operator, 132
grep (findstr) command
  finding text within files, 61-62
  regular expressions and, 127
groups
  capture groups, 122-124
    advanced find/replace, 125-126
    capture HTML tag example, 124-125
  regular expressions, 121-122
Grunt
  documentation, 194
  Gruntfile.js, 73
  installing, 66-68-69
  Jasmine testing library, 215-220
  packaging extensions, 78
  plug-ins, 73-74
    concatenating files, 74-75
    copying files, 75
    error-checking files, 75-76
    JSHint, 75-76
    loading, 76
  registering tasks, 76-78
  watching tasks, 79-80

H

hacking, 174-175
hard drives, 35, 49
hash. ***See*** **dictionary**
head tags, test files, 13
hexadecimal numbers, 88-89
hidden files, 52
highlighting syntax, text editors, 6
high-quality code, importance of, 174-175
Home directory, accessing from command line, 56
HTML (HyperText Markup Language)
  capture HTML tag example, 124-125
  JavaScript functionality and HTML, 13-15
  test files, creating, 12-13

I

IDE (Integrated Development Environments), 8
if statements, 137-138
  else if statements, 138
  true/false and if statements, 139
ignore case flags, regular expressions, 126
IIFE (Immediately Invoked Function Expression), 169
images and for loops, 142
increment operator, 134
infinite loops, 145-146
inheritance in OOP, 266-267
input (source code)
  from the Clock, 35
  from external devices, 36
  functions, 34-35
  GIGO programming, 37
    defensive coding, 37
    sanitization, 37-38
    validation, 37-38
  from the hard drive, 35
  from the Internet, 36
  need for input, 34
  origins of, 35-36
  from other programs, 35
  receiving, 36
inspecting code via Console (Chrome Dev Tools), 228-231
installing
  Grunt, 66-69
  node.js, 64-66
  NPM, 66
  programming languages, 32-33
instances in OOP, 267
instructions
  evolution of, 48
  executing, stupidity of computers, 47-48
    && (and) operator, 133
    === (equals) operator, 131
  source code as a set of, 25-26
integers, 88-89
integration tests, 221-222
IntelliJ text editor, 11
Internet, input (source code) from, 36

interpreted languages, 31
invoking functions, 154-157, 169
it function, testing, 217
iterative planning, 203

J

Jasmine testing library, 215-220
Java, 28, 96
JavaScript, 28
- \ (backslash), 15
- ! (not) operator, 134
- || (or) operator, 133
- automating tasks, 73
  - *comments, 41-42*
- escape characters, 15
- functions, 169
- HTML functionality and JavaScript, 13-15
- JSHint, 75-76
- manifest.json files and JavaScript references, 20
- mistakes, avoiding, 70
- node.js, installing, 64-66
- pop method, adding items to arrays, 93
- push method, removing items from arrays, 93
- regular expressions, 110
- setTimeout, 149
- strings, 96
- variables, 16
  - *adding functionality to, 16*
  - *assigning functions to, 164*
  - *global variables, 167-168*
  - *local variables, 168*
  - *lookups, 168-169, 170-171*
  - *scope, 166-171*
  - *setting value of, 41*
  - *"working" variables, 16*

JSDoc, 194
JSHint, 75-76
JSON documents, 17

K

key-value pairs, storing in strings, 97-98
Khan Academy, 251
kittenbook
- adding to Chrome, 17-19
- automating tasks
  - *concatenating files, 74-75*
  - *copying files, 75*
  - *error-checking files, 75-76*
  - *Grunt plug-ins, 73-76*
  - *Gruntfile.js, 73*
  - *JSHint, 75-76*
  - *loading plug-ins, 76*
  - *packaging extensions, 78*
  - *registering tasks, 76-78*
  - *watching tasks, 79-80*
- code conventions/standards, establishing, 178-180
- code encapsulation, 159-162
- debugging, 223-224
- documentation, 192-194
- functions, code encapsulation, 159-162
- Git version control system, 265
- image arrays, 93-94
- organizing code, 159-162
- personalizing, 243
  - *adding functionality to Facebook, 244*
  - *restyling Facebook, 243-244*
  - *sharing your version of kittenbook, 245*
- phone number validation, 120-124
- project directories, creating, 12
- README.md, 193-192
- source code as instructions, 25-26
- state, adding, 39-40
- test files
  - *creating, 12-13*
  - *head tags, 13*
  - *HTML functionality and JavaScript, 13-15*
  - *JavaScript functionality and HTML, 13-15*
  - *name prompt, 15-16*
- testing, 215-216, 217-218
- updating in Chrome, 20

Kleene star. *See* * (star)

L

languages (programming), 26-27
- Bash, 27
- C, 27
- C++, 27
- C#, 27-28
- comments, 41-42
- compiled languages, 31
- dynamically/statically typed languages, 96
- installing, 32-33
- interpreted languages, 31
- Java, 28
- JavaScript, 28
- Objective-C, 28-29
- Perl, 29
- PHP, 29
- Python, 29-30
  - *for loops, 178*
  - *whitespace in code, 178*
- R, 30
- Ruby, 30
- runtime environments, 32-33
- source code's transformation to binary, 31
- statically/dynamically typed languages, 96
- Swift, 30
- variables, 41-42
  - *setting value of, 41*
  - *storing, 42-44*
- VBA, 30-31

learning via documentation, 192
less than (<) operator, 132
less than or equal to (<=) operator, 132
limitations and capabilities, designing to, 206-207
links (symbolic), 51-52
listeners (event), 147-148
listing directories, 57-58
local variables, 168
locating
- advanced find/replace via capture groups, 125-126
- bugs via documentation, 191
- dependencies, 68
- files in file systems
  - *absolute (full) paths, 56*
  - *relative (partial) paths, 56*
- open source projects, 248
- projects (new), 246-247
- regular expressions as Find feature, 109-110
- right terms when searching, 235
  - *&& (and) operator (JavaScript), 133*

logical operators, 132-133
- ! (not) operator (JavaScript), 134
- || (or) operator (JavaScript), 133

logs and debugging, 224
long integers, 88-89
long term (persistent) storage, 104
long-term memory, 49
looping and memory leaks, 44-45
loops
- infinite loops, 145-146
- for loops, 141
  - *arrays and, 142*
  - *break statements, 143-144*
  - *forEach method, 144-145*
  - *nested loops, 143*
  - *pictures, looping through, 142*
  - *whitespace in code, 178*
- nested loops, 143
- return statements, 157
- while loops, 145-147
  - *break statements, 146*
  - *infinite loops, 145-146*

Lovelace, Ada, 24
ls (dir) command, 57-58

M

"maintainable" code, writing, 175-176
manifest.json files
- Chrome extension, 17
- JavaScript references, 20
- troubleshooting, 19

manual testing, 209
- "crazy" changes, making while testing, 210-211
- "dogfooding," 211
- testing while working, 210

many-to-many relationships, 106
map. *See* dictionary
Markdown files, 193-192

matching patterns, regular expressions, 113-114
math operators, 136
md command. *See* mkdir (md) command
Mediator design pattern, 269
memory
- documentation versus memory, 191-192
- hard drives, 49
- leaks, 44-45
- limits of, 44
- long-term memory, 49
- looping and memory leaks, 44-45
- RAM, 49
- short-term memory, 49
- variables, storing, 42-44

merging changes and version control, 258-260
mixing data types, 84
mkdir (md) command, creating directories, 59-60
modulo (%) operator, 136
monospace type font and text editors, 6
moving
- files, 58
- up in directories, 56

multiline flags, regular expressions, 127
multiple operator assignments, 135
multiplication (*) operator, 136
mv (move) command
- moving files, 58
- renaming files, 58-59

N

name prompt, test files, 15-16
naming
- files, 58-59
- functions, 159
- variables, 168

nested loops, 143
node.js
- arrays and, 92
- installing, 64-66

not equals (!==) operator, 132
not operator (!), 134
Notepad++ text editor, 9
NPM (Node Package Manager), installing, 66
numbers
- converting strings to numbers via parseInt, 234
- data types, 88-89

O

Objective-C, 28-29
objects
- storing data in, 98
- trees, 101

one-to-many relationships, 106
one-to-one relationships, 106
online resources
- blogs, 237, 241-242
- codeacademy, 251
- Coursera, 250-253
- forums, 239
- free online education, 249
  - *codeacademy, 251*
  - *Coursera, 250-251*
  - *Google, 249*
  - *Khan Academy, 251*
  - *Project Euler, 249*
  - *tutorials, 251*
  - *Udacity, 250*
- Google, 249
- Khan Academy, 251
- paid education
  - *Coursera, 252-253*
  - *Treehouse, 253*
  - *Udacity, 252-253*
- Project Euler, 249
- StackOverflow, 238
- Treehouse, 253
- tutorials, 251
- Udacity, 250-253

OOP (Object-Oriented Programming), 265-266
- classes, 266
- inheritance, 266-267
- instances, 267

open source projects, 247-249
open source software, benefits from, 181
communal builds, 181-182
faster builds, 180-181
solo builds, recognizing the need for, 182
opening terminal, 54
operators, 131
binary operators, 134
% (modulo) operator, 136
assigning, 135
coercion, 134-135
math operators, 136
multiple operator assignments, 135
order of operation, 135
coercion, 134-135
comparison operators, 131-132
increment operator, 134
logical operators, 132-133
&& (and) operator (JavaScript), 133
|| (or) operator (JavaScript), 133
! (not) operator (JavaScript), 134
math operators, 136
ternary operators, 136-137
unary operators, 134
or (||) operator, 133
organizing code
code encapsulation, 158
dividing jobs per function, 159-162
one job per function, 158-159
conventions, establishing, 173-174, 178-180
"hacky" code, 174-175
high-quality code, importance of, 174-175
"maintainable" code, writing, 175-176
price of poor coding, 175
standards, 174, 178-180
"tech debt," 175
DRY concept, 165-166
reusing code, 163
DRY concept, 165-166
general problem solving, 163
generalized functions, 163-164
scope, variables/functions and
global variables, 167-168
local variables, 168
lookups, 168-171
restricting via functions, 169
outdated documentation, 190-191
output (source code)
GIGO programming, 37
defensive coding, 37
sanitization, 37-38
validation, 37-38
types of, 36
advanced find/replace, 125-126
capture HTML tag example, 124-125

P

packaging, automating, 72, 78
parameters and arguments, 155-156
parentheses (( )), capture groups, 121-122
parseInt
defining, 234
older numbering systems, 234
strings, converting to numbers, 234
partial paths. *See* relative (partial) paths
pattern-matching, regular expressions, 113-114
performance, planning for, 204-205
Perl, 29
permissions, 51, 207
persistent (long term) storage, 104
personalizing
kittenbook, 243
adding functionality to Facebook, 244
restyling Facebook, 243-244
sharing your version of kittenbook, 245
Pinterest, 244
phone number validation project, 116-124
PHP, 29
pictures. *See* images
Pinterest, personalizing, 244
planning, 199
architectures, 200-201
breakdowns, exploring, 202-203
draw diagrams, 201-202
weaknesses, exploring, 202-203

balance, 206
capabilities and limitations, designing to, 206-207
constraints, identifying and creating, 206
limitations and capabilities, designing to, 206-207
permissions, 207
best practices, 183
design step, 199-200
extensibility, designing for, 203
iterative planning, 203
permissions, 207
priorities, 204
deadlines, 205
performance, 204-205
scalability, 205
security, 205
user experience, 204
specifications, 216-217
concatenating files, 74-75
copying files, 75
error-checking files, 75-76
JSHint, 75-76
loading, 76
packaging extensions, 78
watching tasks, 79-80
plug-ins (Grunt), 73-74
plus sign (+)
addition operator, 136
regular expressions, 111-112
pop method, adding items to arrays, 93
previous directory, returning to, 56
primitive data types, 84
Boolean data types, 84-87
character data types, 88
number data types, 88-89
priorities, planning, 204
deadlines, 205
performance, 204-205
scalability, 205
security, 205
user experience, 204
processors
CPU, 49
executing source code, 34
programming
abstractions, 48
architectures, 200-201
breakdowns, exploring, 202-203
design patterns, 268-270
draw diagrams, 201-202
weaknesses, exploring, 202-203
automated testing, 211
failures, 218-219
functions, 217
kittenbook, 215-218
"spying" on functions, 219-220
unit tests, 212-217
balance, designing for, 206
capabilities and limitations, designing to, 206-207
constraints, identifying and creating, 206
limitations and capabilities, designing to, 206-207
permissions, 207
best practices, 182
documentation, 182-183
planning, 183
testing, 183
certifications
Coursera, 252-253
Udacity, 252-253
code encapsulation, 158
dividing jobs per function, 159-162
one job per function, 158-159
coding vs, 5
communal builds, 181-182
computers' understanding of code, 177
consistency in coding, 177
conventions, establishing, 173-174, 178-180
"hacky" code, 174-175
high-quality code, importance of, 174-175
"maintainable" code, writing, 175-176
price of poor coding, 175
standards, 174, 178-180
"tech debt," 175
debugging, 222-223
breakpoints, 224-226
call stacks, 231

code inspection via Console (Chrome Dev Tools), 228-231
errors, 223-224
logs, 224
repetition, importance of, 232
root causes, determining, 231
"stepping through" code, 229-231
design patterns
Mediator design pattern, 269
Pub Sub design pattern, 268-269
Singleton design pattern, 270
design step, 199-200
documentation,
audience, writing for an, 196-197
best practices, 182-183
bugs, finding, 191
comments, usefulness of, 192-193
decision-making process, documenting, 196-197
diagrams, 192-193
Grunt, 194
JSDoc, 194
kittenbook, 192-194
learning via documentation, 192
Markdown files, 193
memory versus documentation, 191-192
obvious, documenting the, 189
outdated documentation, 190-191
resources, documenting, 197
self-documenting code, 187-188
"sneaky" code, 188
teaching via documentation, 197
DRY concept, 165-166
evolution of, 48
formatting code, 176-177
consistency in coding, 177
whitespace in code, 178
GIGO programming, 37
defensive coding, 37
sanitization, 37-38
validation, 37-38
"hacky" code, 174-175
high-quality code, importance of, 174-175
inspecting code via Console (Chrome Dev Tools), 228-231
"maintainable" code, 175-176
OOP, 265-266
classes, 266
inheritance, 266-267
instances, 267
open source software, benefits from, 181
communal builds, 181-182
faster builds, 180-181
solo builds, recognizing the need for, 182
organizing code, code encapsulation, 158-162
outdated documentation, 190-191
permissions, 207
planning, 199
architectures, 200-203
balance, 206-207
best practices, 183
breakdowns, exploring, 202-203
design step, 199-200
draw diagrams, 201-202
extensibility, designing for, 203
iterative planning, 203
priorities, 204-205
specifications, 200, 216-217
weaknesses, exploring, 202-203
priorities, 204
deadlines, 205
performance, 204-205
scalability, 205
security, 205
user experience, 204
reusing code, 163
DRY concept, 165-166
general problem solving, 163
generalized functions, 163-164
"robust" code, 151
scope, variables/functions and
global variables, 167-168
local variables, 168
lookups, 168-171
restricting via functions, 169
solo builds, recognizing the need for, 182
"spaghetti" code, 203
specifications, 216-217
"stepping through" code, 229-231
"syntatic sugar," 140
"tech debt," 175

testing
automated testing, 211-220
best practices, 183
"crazy" changes, making while testing, 210-211
"dogfooding," 211
integration tests, 221-222
Jasmine testing library, 215-220
manual testing, 209-211
repetition, importance of, 232
test-driven development, 220
testing while working, 210
unit tests, 212-217
version control, 255-256
collaboration and, 256-260
conflict resolution and, 258-260
Git version control system, 260-265
merging changes, 258-260
reasons for using, 256-257
reviewing code, 260
Subversion version control system, 260
updating projects, 258-260
visualizing, 256
whitespace in code, 178
programming languages, 26-27
Bash, 27
C, 27
C++27
C#27-28
comments, 41-42
compiled languages, 31
dynamically/statically typed languages, 96
installing, 32-33
interpreted languages, 31
Java, 28
JavaScript, 28
Objective-C, 28-29
Perl, 29
PHP, 29
Python, 29-30
for loops, 178
whitespace in code, 178
R, 30
Ruby, 30
runtime environments, 32-33
source code's transformation to binary, 31
statically/dynamically typed languages, 96
Swift, 30
setting value of, 41
storing, 42-44
variables, 41-42
VBA, 30-31
Project Euler, 249
projects
choosing, 246-247
directories, creating, 12
new projects, finding, 246-247
open source projects, 247-249
updating, version control and, 258-260
prompts
command line prompt, 54-56
name prompt and test files, 15-16
prompt function, 25-26, 156
Pub Sub design pattern, 268-269
push method, removing items from arrays, 93
push/pull in Git, 263-264
Python, 29-30
for loops, 178
whitespace in code, 178

Q - R

question mark (?), regular expressions, 111, 121-122
questions (error-handling)
answering, 241-242
phrasing a question, 241
referencing your personal code, 241

R, 30
RAM (Random Access Memory), short-term memory, 49
README.md and kittenbook, 193-192
refactoring code and regular expressions, 127
registering tasks, 76-78
regular expressions, 90
\ (backslash), 112-113, 119
[ ] (brackets), 115
^ (caret), 116
{ } (curly braces), 113
+ (plus sign), 111

? (question mark), 111
\B, 119
\b, 119
\D, 119
\d, 119
\n, 119
\r, 119
\s, 119
\t, 119
\W, 119
\w, 119
limitations of, 121
list of acceptable characters, 115
ranges, 115
* (star), 112
code refactoring, 127
data extraction, 128
defining, 109-110
escaping, 112-113
as Find feature, 109-110
flags, 126
global flags, 126
ignore case flags, 126
multiline flags, 127
greed and, 114
grep (findstr) command, 127
groups
(?:) groups, 121-122
capture groups, 122-126
JavaScript, 110
negation in, 116
pattern-matching, 113-114
phone number validation project, 116-124
regexpal, 116
Regexper visualization tool, 114
repetition, 111-114
shortcuts, 119-121
syntax of, 110
testing, 110
uses for, 127-128
validation, 128
visualizing, 114
relational databases, 104
many-to-many relationships, 106
one-to-many relationships, 106
one-to-one relationships, 106
relationships (overview of), 105-106
SQL, 106-107
tables, 105
relative (partial) paths, 57
removing items from arrays, 93
renaming files, 58-59
repositories
adding files to, 261-262
cloning, 261
updating via Git, 263-264
resources
books, 252
documenting, 197
error-handling
blogs, 237, 241-242
forums, 239
friends, 237-238
StackOverflow, 238
online resources
codeacademy, 251
Coursera, 250-253
free online education, 249-251
Google, 249
Khan Academy, 251
Treehouse, 253
tutorials, 251
Udacity, 250-253
tutorials, 251
return statements
functions and, 156-157
loops and, 157
reusing code, 163
DRY concept, 165-166
general problem solving, 163
generalized functions, 163-164
reviewing code, 260
rm (del) command, 59
"robust" code, 151
Ruby, 30
runtime environments, 32-33

S

sanitization and programming, 37-38
scalability, planning for, 205
scope, 166-167
global variables, 167-168
local variables, 168

lookups, 168-171
restricting via functions, 169
searching, 234
backward, 236
blogs as a programming resource, 237
parseInt as a search tool, 234
resources, identifying quality, 236-237
right terms, finding, 235
deciphering error messages, 236
"leveling up," 235-236
security
permissions, 207
planning for, 205
self-documenting code, 187-188
sets, 99
setTimeout, 149
short integers, 88-89
short-term memory, 49
Singleton design pattern, 270
"sneaky" code, 188
software
automated testing, 211
functions, 217
kittenbook, 215-218
unit tests, 212-217
defensive coding, 37
defining, 23-24
dependencies, 68
first instance of, 24
frameworks, importance of understanding, 26
GIGO programming, 37
defensive coding, 37
sanitization, 37-38
validation, 37-38
Grunt, installing, 66-69
input
from the Clock, 35
from external devices, 36
functions, 34-35
GIGO programming, 37-38
from the hard drive, 35
from the Internet, 36
need for input, 34
origins of, 35-36
from other programs, 35
receiving input, 36
life cycle of, 24
node.js, installing, 64-66
NPM, installing, 66
open source software, benefits from, 181
communal builds, 181-182
faster builds, 180-181
solo builds, recognizing the need for, 182
operation of, 24
origin of, 24
output
GIGO programming, 37-38
types of, 36
source code, 25
Bash, 27
C, 27
C#, 27-28
C++, 27
comments, 41-42
compiled languages, 31
executing via processors, 34
installing programming languages, 32-33
instructions, source code as a set of, 25-26
interpreted languages, 31
Java, 28
JavaScript, 28
Objective-C, 28-29
Perl, 29
PHP, 29
programming languages, 26-31
prompt function, 25-26
Python, 29-30
R, 30
Ruby, 30
runtime environments, 32-33
source code's transformation to binary, 31
Swift, 30
variables, 41-44
VBA, 30-31
state, 38-40
testing
automated testing, 211-220
"crazy" changes, making while testing, 210-211

"dogfooding," 211
manual testing, 209-211
testing while working, 210
unit tests, 212-217
solo builds, recognizing the need for, 182
source code, 25
comments, 41-42
computers' understanding of code, 177
defensive coding, 37
evolution of, 48
executing via processors, 34
GIGO programming, 37
defensive coding, 37
sanitization, 37-38
validation, 37-38
input
from the Clock, 35
from external devices, 36
functions, 34-35
GIGO programming, 37-38
from the hard drive, 35
from the Internet, 36
need for input, 34
origins of, 35-36
from other programs, 35
receiving input, 36
instructions, source code as a set of, 25-26
output
GIGO programming, 37-38
types of, 36
programming languages, 26-27
Bash, 27
C, 27
C#, 27-28
C++27
compiled languages, 31
installing, 32-33
interpreted languages, 31
Java, 28
JavaScript, 28
Objective-C, 28-29
Perl, 29
PHP, 29
Python, 29-30
R, 30
Ruby, 30
runtime environments, 32-33
Swift, 30
VBA, 30-31
prompt function, 25-26
source code's transformation to binary, 31
variables, 41-42
setting value of, 41
storing, 42-44
"spaghetti" code, 203
specifications
software design and, 200
testing and, 216-217
in arrays, 98
databases, 104-106
long term (persistent) storage, 104
in objects, 98
in strings, 97-98
"spying" on functions, 219-220
SQL (Structured Query Language), 106-107
StackOverflow as an error-handling resource, 238
stacks, 99-100
standards (coding), establishing, 174, 178-180
star (*)
multiplication operator, 136
regular expressions, 112
state, 38-40
statically/dynamically typed languages, 96
"stepping through" code, 229-231
storing data
strings, 15, 89-90
converting to numbers via parseInt, 234
creating
in Java, 96
in JavaScript, 96
key-value pairs, storing in strings, 97-98
URL, 97-98
head tags, 13
HTML functionality and JavaScript, 13-15
JavaScript functionality and HTML, 13-15
name prompt, 15-16
"stuffed duck, the," 240

Sublime Text text editor, 9-11
subtraction (-) operator, 136
Subversion version control system, 260
Swift, 30
switch statements, 139-140
symbolic links, 51-52
"syntatic sugar," 140
syntax highlighting and text editors, 6

T

Tab key, autocompleting commands in command line, 57
tables, relational databases, 105
tags (head), test files, 13
teaching and error-handling, 241
teaching via documentation, 197
"tech debt," 175
terminal, opening, 54
ternary operators, 136-137
test files, creating, 12-13
testing
- automated testing, 211
  - *failures, 218-219*
  - *functions, 217*
  - *kittenbook, 215-218*
  - *"spying" on functions, 219-220*
  - *unit tests, 212-217*
- automating, 71-72
- best practices, 183
- failures, 218-219
- functions, 219-220
- integration tests, 221-222
- Jasmine testing library, 215-220
- kittenbook, 215-218
- manual testing, 209
  - *"crazy" changes, making while testing, 210-211*
  - *"dogfooding," 211*
  - *testing while working, 210*
- regular expressions, 110
- repetition, importance of, 232
- test-driven development, 220
- unit tests, 212-217

text editors
- choosing, 5-11
- core features, 6-8
- Eclipse text editor, 10
- extensibility, 8
- Gedit text editor, 9-10
- IDE, 8
- IntelliJ text editor, 11
- monospace type font, 6
- Notepad++ text editor, 9
- Sublime Text text editor, 9-11
- syntax highlighting, 6
- text completion, 7-8
- TextMate text editor, 9
- Vim text editor, 10
- Visual Studio text editor, 11
- Xcode text editor, 11

TextMate text editor, 9
timeouts, 149
tracking changes to files via Git, 261-262
Treehouse, 253
trees, 100-101
troubleshooting
- breakdowns, exploring during planning, 202-203
- debugging, 222-223
  - *breakpoints, 224-226*
  - *call stacks, 231*
  - *code inspection via Console (Chrome Dev Tools), 228-231*
  - *errors, 223-224*
  - *finding bugs via documentation, 191*
  - *logs, 224*
  - *repetition, importance of, 232*
  - *root causes, determining, 231*
  - *"stepping through" code, 229-231*
- infinite loops, 145-146
- manifest.json files, 19
- try/catch and, 150-151
- weaknesses, exploring during planning, 202-203

true/false and if statements, 139
try/catch, 150-151
tutorials, 251

U

Udacity, 250-253
unary operators, 134
Undo actions, 149

unit tests, 212-217
Unix time, 90
updating
- Chrome extension, 20
- projects, version control, 258-260
- repositories via Git, 263-264

URL (Uniform Resource Locators), 97-98
user accounts and permissions, 51
user experience, planning for, 204

V

validation
- phone number validation project, 116-124
- programming and, 37-38
- regular expressions and, 128

variables, 16, 41-42
- adding functionality to, 16
- assigning functions to, 164
- global variables, 167-168
- JavaScript, setting the value of variable, 41
- local variables, 168
- lookups, 168-171
- naming, 168
- scope, 166-167
  - *global variables, 167-168*
  - *local variables, 168*
  - *lookups, 168-171*
- storing, 42-44
- "working" variables, 16

VBA (Visual Basic for Applications), 30-31
version control, 255-256
- collaboration and, 256-260
- conflict resolution and, 258-260
- Git version control system, 260-261
  - *adding files to repositories, 261-262*
  - *branches, 265*
  - *cloning repositories, 261*
  - *committing changes to files, 262*
  - *kittenbook, 265*
  - *push/pull in, 263-264*
  - *tracking changes to files, 261-262*
  - *workflow of, 264*
- merging changes, 258-260
- reasons for using, 256-257
- reviewing code, 260
- Subversion version control system, 260
- updating projects, 258-260
- visualizing, 256

Vim text editor, 10
Visual Studio text editor, 11
visualizing regular expressions, 114

W

watching tasks, 79-80
weaknesses, exploring during planning, 202-203
web pages, creating, 12-13
websites, creating, 247
when actions, 147
- cron jobs, 148
- events, 147-148
- timeouts, 149

when to ask questions, determining (error-handling), 240
while loops, 145, 146-147
- break statements, 146
- infinite loops, 145-146

whitespace in code, 178
"working" variables, 16
"wraparound date," and Unix time, 90

X - Y - Z

Xcode text editor, 11

Y2K and Unix time, 90