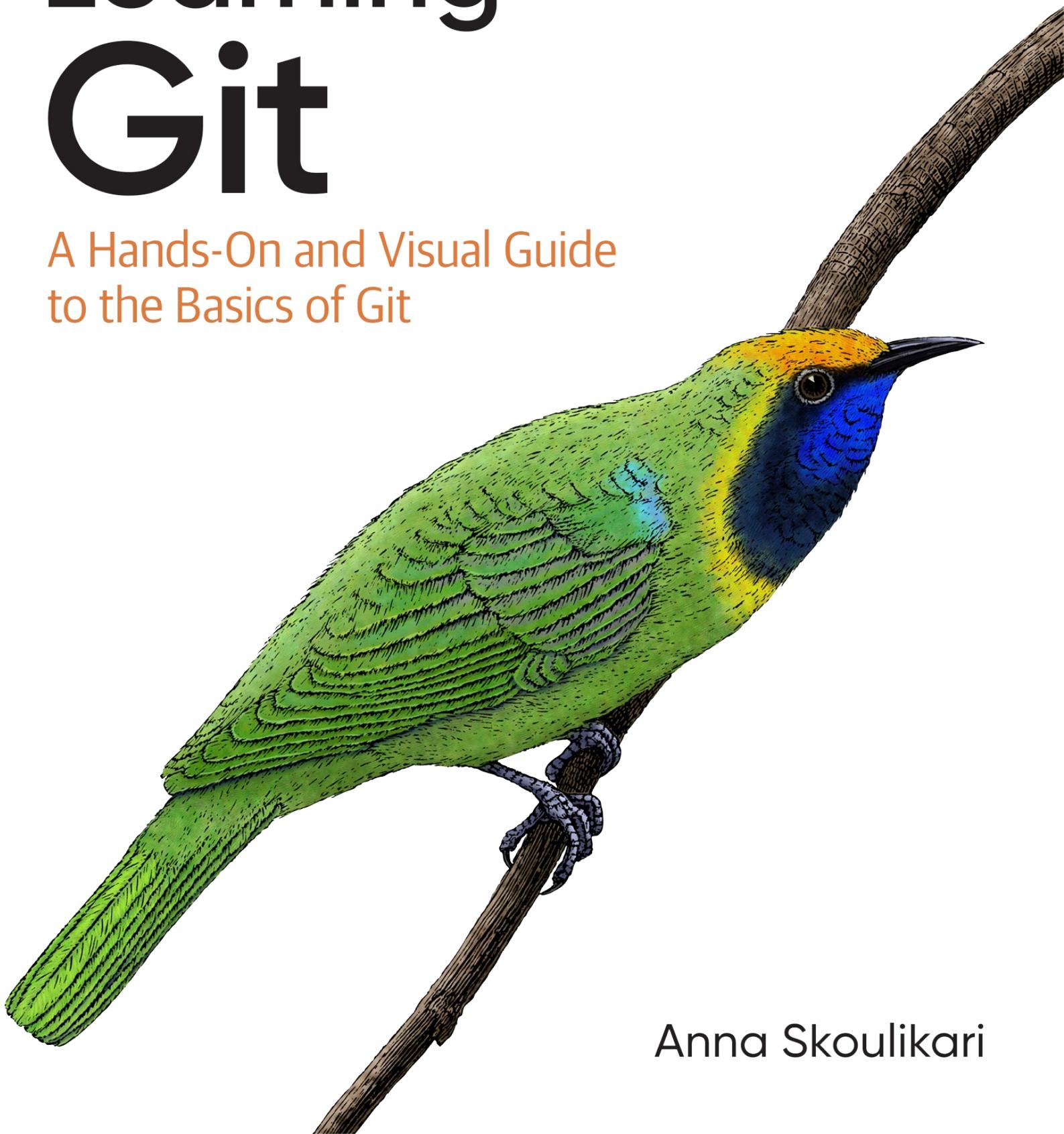


O'REILLY®

Learning Git

A Hands-On and Visual Guide
to the Basics of Git



Anna Skoulikari

Learning Git

A Hands-On and Visual Guide to the Basics of Git

Anna Skoulikari

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Learning Git

by Anna Skoulikari

Copyright © 2023 Anna Skoulikari. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North,
Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Development Editor: Shira

Evans

Acquisition Editor: Melissa

Duffield

Production Editor: Christopher

Faucher

Copyeditor: Rachel Head

Proofreader: Piper Editorial

Consulting, LLC

Indexer: nSight, Inc.

Cover Designer: Karen

Montgomery

Interior Designers: Ron

Bilodeau and

Monica Kamsvaag

Illustrator: Kate Dullea

May 2023: First Edition

Revision History for the First Edition:

2023-05-16 First Release

See <https://www.oreilly.com/catalog/errata.csp?isbn=0636920728078> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc.
Learning Git and related trade dress are trademarks of O'Reilly Media, Inc.

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 O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

Although the publisher and author have used reasonable care in preparing this book, the information it contains is distributed "as is" and without warranties of any kind. This book is not intended as legal or financial advice, and not all of the recommendations may be suitable for your situation. Professional legal and financial advisors should be consulted, as needed. Neither the publisher nor the author shall be liable for any costs, expenses, or damages resulting from use of or reliance on the information contained in this book.

978-1-098-13391-7

[LSI]

[*Preface*]

I never thought I would write a book teaching Git. But through a fortunate series of events, I found myself with a creative idea for how I could teach this technology in a simple way.

My journey started when I attended a coding bootcamp to learn web development. The teachers at the bootcamp briefly introduced Git to the students, but given that all our projects were done individually, we didn't have to use it extensively.

After the coding bootcamp, I got a job as a junior frontend developer working on a website in a big company. My real Git learning journey began on the first day of my new job. In those first months, working as part of a team in a large company, I realized I was terrified of it. Any time I had to do something that seemed remotely complicated using Git, I thought I was going to destroy the repository or seriously mess something up.

To be able to work properly with my coworkers, I decided to teach myself the ins and outs of Git. But as I read through various online resources, it quickly became clear to me that most of the material out there was not designed for people who were just starting out. Once I understood the basics, an idea started to form in my mind of how I could teach this technology in a simpler way using visuals and colors.

I ended up creating an online course that I uploaded to the web. While working on the course, in the back of my mind, I thought to myself that someday I might write a book about it as well.

I got a lot of positive feedback about the course, and finally, in the summer of 2021, I decided it was time to get started on that project. The book you're

reading now is the product of that decision, and I hope it helps you on your Git learning journey!

Who This Book Is For

This book is for anyone who wants to learn the basics of how Git works. It is especially designed for individuals that are just getting started learning technical skills, or that work in nontechnical roles but need to use Git to collaborate with their technical counterparts. Some examples of individuals that may benefit from this book include (but are not limited to) coding bootcamp students, computer science students, technical writers, product managers, designers, junior developers, data scientists, and self-taught programmers.

The book is written for people with no experience using Git, as well as those with a bit of experience using Git. If you have no experience with Git, that's not a problem since this book starts from zero. We'll begin with installing Git and how to use the command line, and build from there.

If you already have some experience using Git or the command line, the first chapter may be a bit of review. However, I encourage you not to skip it because it sets up the Rainbow project that you will be using throughout the rest of the book.

Using This Book

This book is a hands-on learning experience, where you will be carrying out exercises on your computer while learning the basic concepts of Git.

Throughout the book, you will come across two projects: the Rainbow project and the Book project.

The Rainbow project is a hands-on project that you will work on by going through the exercises in the book. It is a simplified project that is intended only for learning purposes. The Book project is an imaginary project that I'll use to demonstrate how certain features of Git might be used for a more realistic project. Let's take a closer look at each of these, and at the way the book is structured.

[NOTE]

Don't worry if this preface contains terminology that you are not yet familiar with, like *repository* and *commit*. I'll explain all of these concepts in the chapters to come.

THE RAINBOW PROJECT

To learn the basics of Git, throughout this book you are going to be working on the Rainbow project. To follow along, you should read the book from [Chapter 1](#) to [Chapter 12](#) in a linear fashion, and you should complete each and every exercise on your computer. For example, exercises in [Chapter 4](#) will assume that you've already completed the exercises in Chapters [1](#), [2](#), and [3](#).

Repositories

On a basic level, a *repository* is a copy of a Git project. At first, you will create one local repository called `rainbow` to work on the Rainbow project. Subsequently (in [Chapter 7](#)), you will create a remote repository called `rainbow-remote`. And finally, in [Chapter 8](#), you will simulate that you are collaborating with a friend on the Rainbow project, and you will create a second local repository called `friend-rainbow`. From [Chapter 8](#) onward, whenever a reference is made to your “friend” doing something, you will have to carry out the action in the `friend-rainbow` repository.

[NOTE]

When I refer to the Rainbow project with a capital R, I’m referring to the entire project that starts off with one repository and in the end contains three repositories. When I refer to the `rainbow` project directory or the `rainbow` repository (with a lowercase r), I’m referring to the specific local repository that is part of the Rainbow project.

Commits

In the Rainbow project, you are going to create and edit files to list the colors of the rainbow, and some colors that are not part of the rainbow. This is not meant to be a realistic example of a project version controlled by Git. It’s a simplified project that enables you to focus on learning instead of building something complicated.

Throughout the book, I’ll use diagrams to illustrate what is happening in the Rainbow project. Every time you add a color to the Rainbow project, you are going to make a *commit* in a repository. A commit basically represents a version of your project. In the diagrams, the commit will be represented by a circle in the color you added, and we will use the name of the color as the

name of the commit as well. For example, the first color that you will add to your Rainbow project is red, so the circle that will represent that commit will be colored red, and we will refer to it as the red commit.

To make the book accessible to readers with color vision deficiency, I'll include the name of the commit (or an abbreviation of the name) in the diagrams. See [Figure P-1](#) for an example of the red commit.



FIGURE P-1

An example of a commit with its full name

[Table P-1](#) contains a list of all the commits you will make in the Rainbow project in this book, showing their full names and their abbreviations.

TABLE P-1. Full list of commits made in the Rainbow project

COMMIT FULL NAME	COMMIT ABBREVIATION
red	R
orange	O
yellow	Y
green	G
blue	B
brown	Br

COMMIT FULL NAME	COMMIT ABBREVIATION
merge commit 1	M1
indigo	I
violet	V
merge commit 2	M2
gray	Gr
black	Bl
rainbow	Ra
pink	P
merge commit 3	M3

[Figure P-2](#) is a diagram of all the commits.

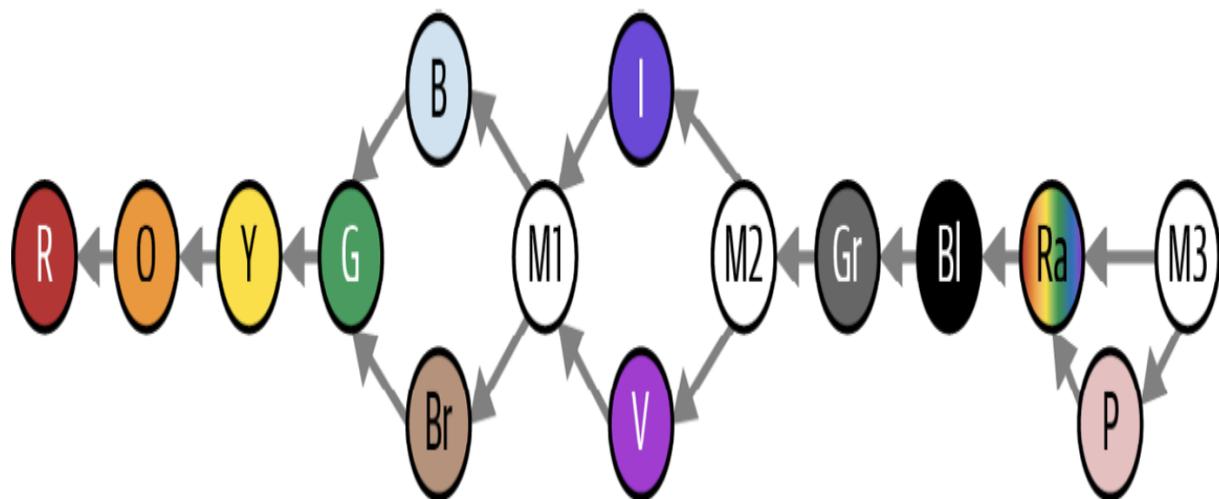


FIGURE P-2

The 15 commits you will make in the Rainbow project by the end of this book

APPENDIXES

While the book is designed to be read from [Chapter 1](#) to [Chapter 12](#) in a linear way, there may be some situations in which you want or need to start off from a specific chapter. For example:

- You have gone through the exercises in the entire book once and you want to review from a specific chapter onward.
- Something went wrong in the Rainbow project in a previous chapter that you were not able to troubleshoot, and you want to continue from a new chapter afresh.

In this case, you can use the instructions in [Appendix A](#) to re-create what the Rainbow project should look like at the start of the chapter you want to begin from.

[Appendix B](#) contains a quick-reference guide to the commands introduced in each chapter.

[Appendix C](#) is a guide to the visual language used in the diagrams in the book.

THE BOOK PROJECT

The Book project is an imaginary project that I will use to demonstrate how Git can be used for more realistic projects. For this project, we'll pretend that I'm writing a book and I want to use Git to version control the files. The book will consist of 10 chapters represented by 10 text files, one for each chapter: `chapter_one.txt`, `chapter_two.txt`, and so on. At times, I will also simulate what it would be like to work on the Book project with a coauthor and/or an editor. These discussions will take place in *Example Book Project* sections.

You will *not* actively work on or build the Book project. It will only be used to provide further examples and descriptions of how certain features of Git are used.

Apart from the Example Book Project sections in the book, you will also come across some other sections. We'll look at those next.

SECTIONS IN THE BOOK

Here's a quick guide to the different types of sections you will encounter in this book:

Example Book Project

As mentioned previously, Example Book Project sections provide additional context and examples about using Git features and commands, based on the Book project.

Follow Along

Follow Along sections present numbered lists of steps that you should carry out on your computer. If a step includes a command in bold, then you must enter and execute that command in the command line. Sample

output is provided for all commands that produce output. This output is based on the Rainbow project that I worked on while creating this book; it was generated by the macOS operating system, but the output of Git commands should be the same on Microsoft Windows. When there are significant differences between the output of a command on Microsoft Windows and macOS, this is stated in the text.

Save the Command

Save the Command sections introduce useful commands, some of which you will use in the Follow Along sections.

A full list of all important commands grouped by chapter is also available as a reference in [Appendix B](#).

Visualize It

Visualize It sections show diagrams of what is happening in the Rainbow project. Two important diagrams that are used in these sections (as well as Figures) are the *Git Diagram* and the *Repository Diagram*. I'll introduce the Git Diagram in [Chapter 2](#), and we will start using the Repository Diagram in [Chapter 4](#). Every diagram in the book is built step by step, with explanations in the text.

A summary of the visual language used throughout this book is also available as a reference in [Appendix C](#).

Note

Note sections provide useful information related to the material in the text.

THE LEARNING GIT REPOSITORY

While I aim to provide most of the information that you need in order to go through the *Learning Git* experience within the book itself, there are some

technologies and processes that change too often to document in the book. I've created the public Learning Git repository (<https://github.com/gitlearningjourney/learning-git>) on GitHub to provide up-to-date information about these technologies and processes. Among other things, the repository contains:

- Information about downloading Git
- Links to resources related to working with hosting services
- Information about setting up HTTPS or SSH access to remote repositories

Throughout the book, I will mention when you may need to consult the Learning Git repository for more guidance.

WHAT THIS BOOK ISN'T

This is not a reference book. I won't cover every single Git command (trust me, there are a lot!). It also isn't an advanced guide to Git. There are a lot of features of Git that won't be discussed in this book because they are not necessary to carry out the basic actions I want to teach you to perform. I have been very selective in what I have included in the book. My aim is to give you a clear mental model of the basics of Git so that you can go on to learn about any additional features you need with a solid foundation in place.

This book also won't tell you what your Git workflow should be or how you should use the features of Git. As much as possible, I try to stay away from giving any opinions on these matters, and instead I focus on teaching you how the tool works. Depending on your individual context and preferences, the tool can be used in different ways.

Where possible, this book aims not to be prescriptive. For example, you are allowed to use whichever text editor or hosting service you prefer to carry out the exercises in the book.

CHAPTER SUMMARY

The book is split into two logical parts. In the first part of the book (Chapters [1](#) through [5](#)), you will learn about working with local repositories on your computer. In Chapters [6](#) through [12](#), you will additionally learn about working with hosting services and remote repositories. Following is a brief summary of what we'll cover in each chapter:

- In [Chapter 1](#), you'll get ready to work on a project using Git by installing it, learning some command line basics, preparing Git settings, making one of the project directories you will use throughout the rest of the book, and preparing a text editor.
- In [Chapter 2](#), you'll turn your project directory into a Git repository. I'll introduce the Git Diagram that represents the different areas of Git, including the working directory, staging area, commit history, and local repository. At the end of the chapter you'll create the first file in your project directory.
- In [Chapter 3](#), you'll learn about and carry out the two main steps to make the first commit in your local repository.
- In [Chapter 4](#), you'll learn about branches: what they are, how to make them, how to switch branches, and how to identify which branch you're on.
- In [Chapter 5](#), you'll learn about the two types of merges and carry out a fast-forward merge.
- In [Chapter 6](#), you'll prepare to work with remote repositories by choosing a hosting service and setting up authentication details to connect to these repositories over either HTTPS or SSH.

- In [Chapter 7](#), we'll discuss the different ways to work with local and remote repositories. You'll learn how to create a remote repository and upload data to it.
- In [Chapter 8](#), we'll start simulating what it's like to work with others on a Git project. In order to do so, you'll create a second local repository; you'll pretend that it belongs to a friend of yours who will be helping you on the Rainbow project and is on their computer. In the process, you'll learn about cloning remote repositories and fetching data.
- In [Chapter 9](#), you'll carry out a three-way merge and learn about the difference between fetching data and pulling data.
- In [Chapter 10](#), we'll go over an example of resolving merge conflicts during a three-way merge.
- In [Chapter 11](#), you'll learn about rebasing. This is an alternative way of incorporating changes from one branch to another, as opposed to merging.
- In [Chapter 12](#), you'll learn about pull requests (also known as merge requests) and how they facilitate collaboration in Git projects.
- The book also includes three appendixes, whose contents are discussed in [“Appendices” on page xv](#).

Conventions Used In This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, and emphasis.

Constant width

Used for commands, command output, filenames, directory names, branch names, repository names, and other text on screen.

Constant width bold

Shows commands or other text that should be typed or entered by the user.

<Constant width angle brackets>

Shows text that should be replaced with user-supplied values or by values determined by context.

O'Reilly Online Learning

For more than 40 years, O'Reilly Media has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform.

O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <https://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-889-8969 (in the United States or Canada)

707-829-7019 (international or local)

707-829-0104 (fax)

support@oreilly.com

<https://www.oreilly.com/about/contact.xhtml>

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *<https://oreil.ly/learning-git>*.

Visit *<http://oreilly.com>* for news and information about our books and courses.

Find us on LinkedIn: *<http://linkedin.com/company/oreilly-media>*

Follow us on Twitter: *<http://twitter.com/oreillymedia>*

Watch us on YouTube: *<http://youtube.com/oreillymedia>*

Acknowledgments

I'd like to thank all the people who have supported me throughout the process of writing this book. This includes friends and family, some of whom were forced to user test earlier versions of the book. It also includes the entire O'Reilly team that has supported me throughout this creative project, and finally all the technical reviewers and user testers that read various iterations of the book and provided me with priceless feedback. Oh, and my partner, who was patient as I delayed my publication date multiple times!

Git and the Command Line

In this chapter, I will introduce what Git is and why we use it, and you'll make sure you have it installed on your computer. You will also set some Git configuration variables (aka *settings*). I'll introduce the graphical user interface and the command line, which are the two tools you will use to interact with Git and the hands-on Rainbow project. To get you comfortable working in the command line, we will go over how to carry out some basic actions such as viewing the current directory location, navigating into and out of directories, and making directories. Finally, at the end of the chapter, you will prepare the text editor you will use to work on the Rainbow project in [Chapter 2](#).

If you already have some experience working in the command line, then you may already know some of the information in this chapter. However, I don't recommend skipping the chapter because it prepares the setup you will use for the rest of the book.

[NOTE]

To understand how to use this book, you must have read "[Using This Book](#)" on page [xii](#). If you have not read that section, I strongly recommend that you go back and do that now.

What Is Git?

Git is a technology that can be used to track changes to a project and to help multiple people to collaborate on a project. At a basic level, a project version controlled by Git consists of a folder with files in it, and Git tracks the changes that are made to the files in the project. This allows you to save different versions of the work you’re doing, which is why we call Git a *version control system*.

Git was created by Linus Torvalds to version control the work done on a large software development project called the Linux kernel. However, since Git can track changes to all sorts of files, it can be used for a wide variety of projects.

Git is a powerful technology, and the abundance of features it provides—as well as the fact that it was originally designed to be used in the command line—means that using it is a bit more complicated than just selecting File → Save on your computer.

To summarize, Git is a version control system that you can download onto your computer that allows you to track the history of a project and collaborate with other people. Next, let’s take a look at [Example Book Project 1-1](#) to see an example of how I might use Git for my Book project.

Example Book Project 1-1

Suppose I am writing a book, and I want to use Git to version control all the files in my Book project. Every time I make changes to the book, I can save a version of it using Git. For example, suppose I make changes to the book on Monday, Wednesday, and Friday, and I save one version on each of those days. This means I have at least three versions of my project. A version of a project in Git is called a *commit*. In [Chapter 2](#), you will learn more about commits. For now, all you need to know is that in my example I have at least three commits.

These three commits allow me to look at the different versions of the book that I had at the end of my Monday work session, my Wednesday work session, and my Friday work session. Git also allows me to compare any of those commits (or saved versions of my project) to one another to check what changed between the different versions. This illustrates how Git helps me track the history of my project.

Now, suppose I decide to work on my Book project together with a coauthor. Git allows me and my coauthor to work on the same project at the same time and combine our work when we are ready. For example, I can work on chapter 1 and my coauthor can work on chapter 2, and when we're ready, we can combine the work we have done.

If we get an editor to review the book, they can also make edits to all the chapters of the book we have written, and we can integrate those changes into the main version of the book as well. This illustrates how Git is a useful tool for collaboration.

Next, let's learn about some of the other tools you will use in this learning experience and how you will be interacting with Git.

The Graphical User Interface and the Command Line

The two main ways to interact with a computer are by using the graphical user interface or the command line.

The *graphical user interface* (GUI) is the set of graphical representations of objects (icons, buttons, etc.) that allows you to interact with your computer. You can think of it as the point-and-click interface. For example, the folders represented by folder icons on your desktop are part of your computer's GUI.

The command line—also known as the command line interface (CLI), terminal, or shell—is a place where you can type text-based commands to interact with your computer.

The default way to work with Git is through the command line. However, there are also ways in which you can work with Git using a GUI: for example, by using a Git GUI client or a text editor that has Git integrations. This means you can carry out Git actions by clicking buttons and selecting options instead of entering commands in the command line.

In this book you will learn how to use Git in the command line, because this allows you to build a solid mental model of how it works and gives you access to all of its functionality. You will use your computer's GUI only for other actions, for example to look at files in the filesystem or to work in your text editor to manage your files. We'll take a closer look at the command line in the next section.

[NOTE]

Throughout this book, I will provide some specific instructions for macOS and Microsoft Windows users. If you are a Linux user, I will assume you already know some command line basics.

Opening a Command Line Window

To use the command line, you must open a command line window using a command line application. At any given point in time, in a command line window, you are in one particular directory, which we refer to as the *current directory*. A directory, for our purposes, is the same thing as a folder.

When you open a command line window there will be a command prompt in the upper-left corner. This is a short piece of text whose exact contents will differ depending on your operating system and computer settings. By default, however, the command prompt indicates the directory location in the command line (in other words, your current directory). When you open a new command line window, the directory location starts off at the current user directory (also known as the home folder), which is represented by the tilde sign (~). This directory location will be the only important part of the command prompt that you will have to identify for the exercises in this book. After the command prompt, there is a cursor that indicates where you are typing in the command line.

See [Figure 1-1](#) for an annotated example of a generic command prompt. In the examples in this book, we use a dollar sign (\$) at the end of the command prompt, but this is just one example of how a command prompt may end. Your command prompt may end with a different character or symbol.

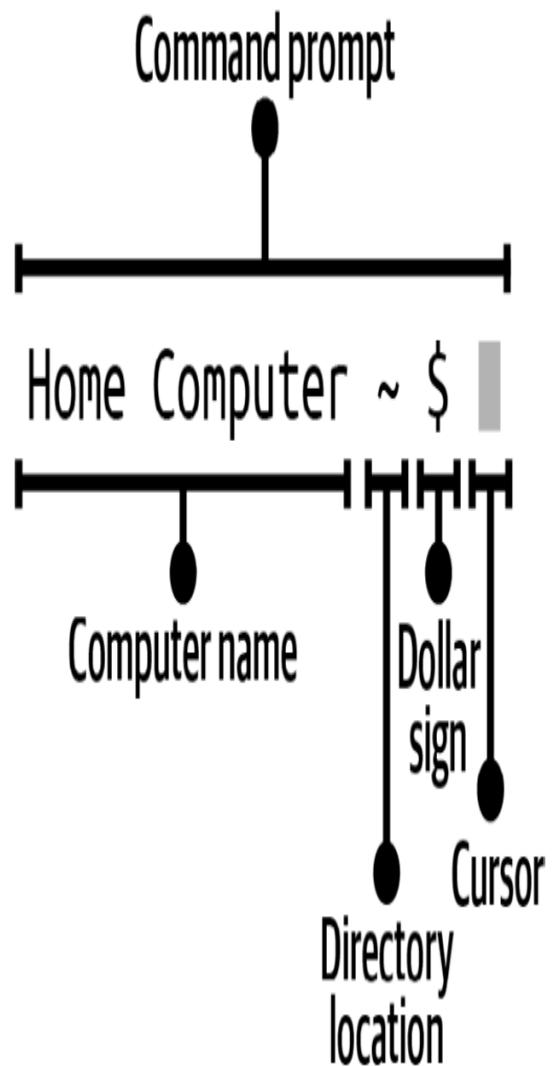


FIGURE 1-1

An example of a command prompt

The command line application you will use to complete the exercises in this book will depend on the operating system you are using:

- macOS—the command line application is called Terminal.
- Microsoft Windows—the command line application is called Git Bash.
It will only be available if you have Git installed on your computer.

[NOTE]

If you are a Microsoft Windows user and you don't have Git installed, then you will need to go to the Learning Git repository (<https://github.com/gitlearningjourney/learning-git>) and follow the steps to install Git for Microsoft Windows in order to have access to Git Bash before continuing with the rest of this chapter.

If you are a macOS user and you don't have Git installed, then you may continue with the rest of this section. You will install Git in "[Installing Git](#)" on page 7.

To open a command line window, you may use the search function on your computer to look up the command line application, select it, and open it. Go to [Follow Along 1-1](#) to open a command line window and view the command prompt.

[FOLLOW ALONG 1-1]

1

Use your command line application to open a command line window.

2

Look at the command prompt in your command line window.

What to notice:

- The command prompt indicates the directory location.

Now that you have opened a command line window, let's cover how you will execute your first command in the command line.

Executing Commands in the Command Line

At the end of the command prompt in your command line window is the *cursor* that indicates where you will type commands. In the Terminal (macOS), by default, the cursor is on the same line as the command prompt, while in Git Bash (Microsoft Windows) it is on the line below. After you type a command, to execute it you have to press the Enter (Return) key.

If a step in a Follow Along section in this book includes a command in **bold** after a dollar sign (\$), then you must execute it in the command line. If the command produces output, it will be shown below the command (not in bold). If the command is supposed to be executed in a directory other than the current user directory, the directory location will be indicated before the dollar sign.

[Figure 1-2](#) shows an example of what it will look like when you execute a command in a Follow Along. In this case, I am executing the `pwd` command while in the `desktop` directory. You'll learn what the `pwd` command does later in this chapter; for now, take a look at [Figure 1-2](#) to identify where it shows the directory location, the command to execute, and the output.

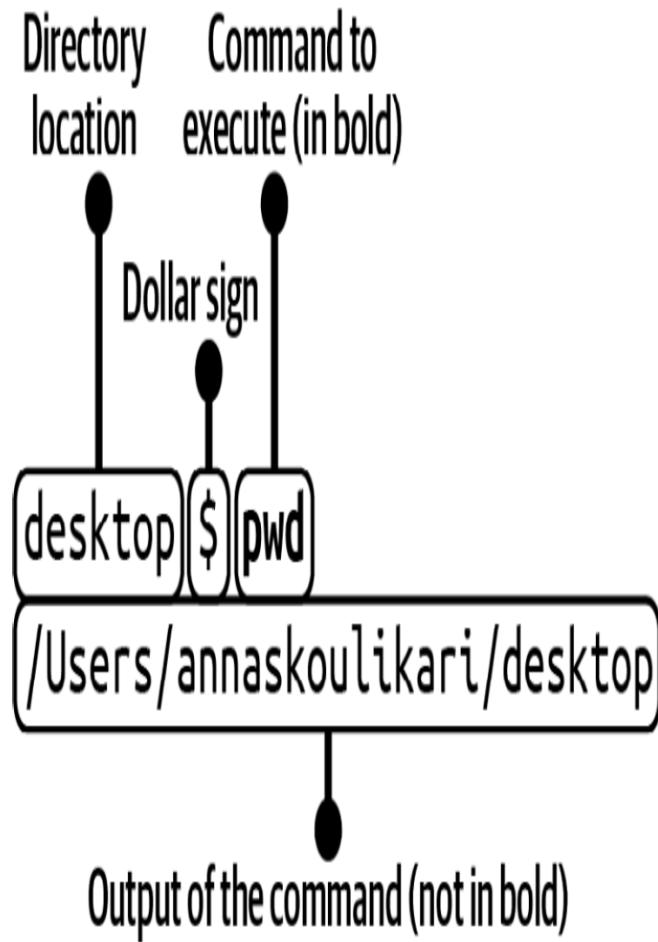


FIGURE 1-2

How to execute a command included in a Follow Along section

In the print version of this book, some longer commands are wrapped because of page width constraints. If you see a command line like the following:

```
rainbow $ git remote add origin https://github.com/gitlearningjourney/rainbow-remote.git
```

you should enter the entire command on one line, with a single space before the wrapped part (in this case, the URL). Reminders will be included in the

appropriate places. If you’re reading a version of this book that does not suffer from this line length constraint, you can ignore these reminders.

COMMAND OUTPUT

Some commands produce output, and some don’t. For commands that produce output, I will provide sample output that is based on the Rainbow project I worked on in creating this book. This output was produced by the macOS operating system, but the output for Git-related commands does not differ between operating systems. In the few cases where non-Git-related command output differs between operating systems in a significant way, I will point it out in the text.

If, while carrying out the exercises in a Follow Along section, you see output that looks drastically different than the output in this book or you get an unexpected error, then you may have done something different than what the instructions indicate, and you may need to review the Follow Along steps.

[NOTE]

It is possible that Git will undergo updates in the future that may affect output in small ways. If I become aware of any significant changes, I will aim to document these situations on the errata page in the Learning Git repository (<https://github.com/gitlearningjourney/learning-git>).

From now on, whenever you encounter a command in a Follow Along exercise, you should enter the command in bold in the command line and execute it.

EXECUTING THE FIRST COMMAND IN THE COMMAND LINE

The first command you will practice executing in the command line is the `git version` command. If you have Git installed on your computer, then it will provide the version number that you have installed. If you don't have Git installed on your computer, then the output will provide you with a message indicating that it is not installed.

To have access to all the commands used in the exercises in this book, I recommend you have a version of Git greater than 2.28. Go to [Follow Along 1-2](#) to check if you have Git installed and what version it is.

[FOLLOW ALONG 1-2]

```
1 $ git version  
git version 2.35.1
```

What to notice:

- If you have Git installed, you see the version of it installed on your computer.
- If you don't have Git installed, you will instead see an error message.

If the `git version` command output indicates that you have a version of Git installed that is greater than 2.28, then you may skip the “Installing Git” section and go to the “Command Options and Arguments” section.

If the `git version` command output indicates you don't have Git installed on your computer or that the version of Git is older than 2.28, then you should

continue onto the next section to install an up-to-date version of Git on your computer.

Installing Git

If you don't yet have Git (version 2.28 or greater) installed, go to [Follow Along 1-3](#). Otherwise, go to the next section.

[FOLLOW ALONG 1-3]

1

Go to the Learning Git repository (<https://github.com/gitlearningjourney/learning-git>) and follow the steps to download Git for your operating system.

Now that you've installed an up-to-date version of Git on your computer, let's continue learning a bit more about the commands you will be using in the Follow Along sections in this book.

Command Options and Arguments

Sometimes you will use commands with options and/or arguments. *Options* are settings that change the behavior of a command. An option follows a single dash (-) or a double dash (--).

Arguments are values that provide information to the command. They will be denoted by angle brackets (<>), indicating that these items should be replaced with user-supplied values. In the exercises, you will have to pass in a value for the argument, without including the angle brackets.

An example of a command with an option and an argument that you will use is `git commit -m "<message>"`. In this example, `-m` is the option and `<message>` is the argument, as shown in [Figure 1-3](#). We will cover what this command does in [Chapter 3](#).

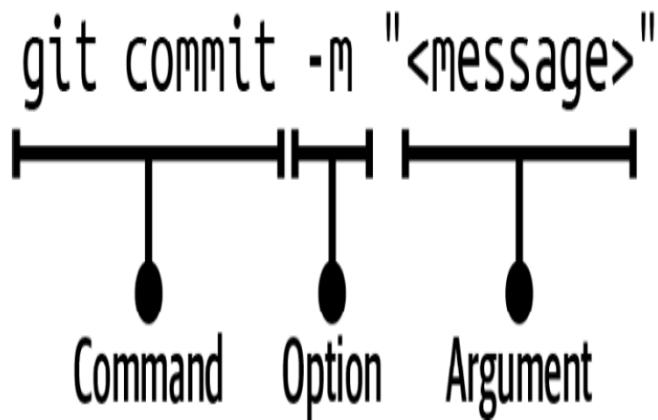


FIGURE 1-3

An example of a command with an option and an argument

In addition to entering commands in the command line, it is also important to learn how to clear them.

Clearing the Command Line

Every time you enter a command in the command line window, it will be listed directly under the previous command you entered (or its output). After you've entered a lot of commands, the command line window gets quite cluttered. To clear the contents of the command line window, you can use the `clear` command.

[SAVE THE COMMAND]

`clear`

Clear the command line window

Go to [Follow Along 1-4](#) to practice using the `clear` command.

[FOLLOW ALONG 1-4]

1 \$ clear

What to notice:

- The command line window is clear.

We've covered entering commands and clearing them from the command line window. Now it's time to prepare the next tool that will help you in your learning journey, the filesystem window.

Opening the Filesystem Window

You will use a filesystem application to open a filesystem window, which is part of the GUI. Throughout your learning journey, you will interact with both the filesystem window and the command line window. Therefore, it is useful to have both windows open side-by-side on your computer screen.

The filesystem application you will use will depend on your operating system:

- macOS—the filesystem application is Finder.
- Microsoft Windows—the filesystem application is File Explorer (or Windows Explorer).

Go to [Follow Along 1-5](#) to open a filesystem window.

[FOLLOW ALONG 1-5]

- 1 Find your filesystem application and open up a filesystem window next to your command line window.

Now that you have both windows open, let's get back to some command line basics.

Working with Directories

As mentioned previously, at any given point in time in a command line window you are in one particular directory (the current directory).

Assuming you haven't changed any default settings, when you first open the command line application you will start off in the current user directory (your home folder), indicated by the tilde sign (~) in the command prompt.

When you navigate to other directories in the command line, the command prompt will change to indicate the directory you are in. You can also use the `pwd` command, which stands for “print working directory,” to see the path to the current directory.

[SAVE THE COMMAND]

`pwd`

Show the path to the current directory

Go to [Follow Along 1-6](#) to practice using the `pwd` command.

[FOLLOW ALONG 1-6]

1 \$ `pwd`
/Users/annaskoulikari

What to notice:

- You are in your current user directory.

[NOTE]

In Follow Along 1-6, the output of the `pwd` command for Microsoft Windows users will be similar to `/c/Users/annaskoulikari`, which is slightly different from the output for macOS users. Keep this in mind when going through the rest of the exercises in this chapter.

The output of the `pwd` command prints the path to the current directory. In [Follow Along 1-6](#), `/Users/annaskoulikari` is an example of a path. My name is Anna Skoulikari and `annaskoulikari` is my username on my computer. `Users`

and `annaskoulikari` are two directories. Directories in a path are separated by a slash (`/`). The `annaskoulikari` directory is inside the `users` directory.

Knowing the directory location in a command line window is useful because many commands show you information about or affect the current directory when you execute them. It also helps you with navigating through your filesystem, which we will cover later in this chapter.

Now that we have covered how to identify your current directory, let's explore how to view the actual contents of a directory.

VIEWING THE CONTENTS OF DIRECTORIES

You can view the contents of a directory in the GUI and the command line window. But before we get to that, I want to mention that there are two types of files and directories that exist in the filesystem: visible files and directories and hidden files and directories. *Visible* files and directories are always visible in the filesystem. *Hidden* files and directories are visible only in the filesystem if you change your settings to view them. They are often files or directories that store information that we, as users, don't need to access, such as application configurations and various system settings.

I do not recommend modifying or deleting hidden files or directories, unless you really know what you're doing. Once you change your settings to view hidden files and directories, they appear partially transparent (grayed out). Their names often start with a dot (`.`).

There are some important hidden files and directories that you will want to be aware of in your Git learning journey, so you need to know how to view them, both in the GUI and the command line.

In the GUI, to view hidden files and directories in a filesystem window you have to explicitly make them visible:

- macOS—to toggle between viewing and hiding hidden files and directories, press Cmd-Shift-dot.
- Microsoft Windows—you must alter the filesystem settings to view hidden files and directories. Consult online resources if necessary for step-by-step instructions for your computer.

Go to [Follow Along 1-7](#) to view the hidden files and directories in your filesystem.

[FOLLOW ALONG 1-7]

1

Make hidden files and directories visible in your filesystem.

In the command line, to view a list of the visible files and directories in the current directory you use the `ls` command (which stands for “list”).

To view both visible and hidden files and directories in the current directory you use the `ls` command with the `-a` option: `ls -a`.

[SAVE THE COMMAND]

ls

List visible files and directories

ls -a

List hidden and visible files and directories

Go to [Follow Along 1-8](#) to practice using these commands to list different kinds of files.

[FOLLOW ALONG 1-8]

1 \$ ls

Applications	Downloads	Music
Desktop	Library	Pictures
Documents	Movies	Public

2 \$ ls -a

.	.config	Desktop
..	.local	Documents
..CFUserTextEncoding	.npm	Downloads
.DS_Store	.ssh	Library
.Trash	.viminfo	Movies
.bash_history	.vscode	Music
.bash_sessions	.yarnrc	Pictures
.bashrc	Applications	Public

What to notice:

- The names of many hidden files and directories start with a dot (.).
- The visible and hidden files and directories shown in the output in this book will be different from the ones on your computer because the contents of everyone's computers are different.

Now that we have covered how to identify the current directory and view its contents in the command line, let's explore how to move between directories.

NAVIGATING INTO AND OUT OF A DIRECTORY

In the GUI, to go into a directory you can double-click on it. In the command line, to go into a directory you use the `cd` command, which stands for “change directory,” and pass in either the name of the directory or the path to the directory.

[SAVE THE COMMAND]

`cd <path_to_directory>`

Change directory

Go to [Follow Along 1-9](#) to practice by navigating into the `desktop` directory.

[FOLLOW ALONG 1-9]

1 \$ cd desktop

2 desktop \$ pwd
/Users/annaskoulikari/desktop

What to notice:

- In step 1, the `cd` command does not produce any output.
- In step 2, the command prompt and the `pwd` command output indicate that you are in the `desktop` directory.

Earlier in this chapter, I mentioned that the command prompt shows the directory location. In [Follow Along 1-9](#), notice that the command prompt updates to show that your current directory is the `desktop` directory. By default, the way the directory location is presented differs depending on the operating system you use:

- macOS—in the Terminal, the name of the current directory is displayed.
- Microsoft Windows—in Git Bash, the path to the current directory is displayed.

[NOTE]

Navigating into and out of directories in the command line does not affect what you are viewing in the filesystem. For example, navigating into the `desktop` directory in the command line will not automatically cause your filesystem to display the contents of the `desktop` directory.

In the GUI, to go back to the parent directory you can select the back button. In the command line, to go back to the parent directory you can pass in two dots (...) to the `cd` command. Two dots represents the parent directory of the current directory. Go to [Follow Along 1-10](#) to try this out.

[FOLLOW ALONG 1-10]

```
1 desktop $ cd ..  
2 $ pwd  
/Users/annaskoulikari
```

What to notice:

- In step 2, the command prompt and the `pwd` output indicate that you are back in the current user directory.

Now that we have covered how to navigate into and out of directories, let's go over how to create a new one.

CREATING A DIRECTORY

In the GUI, you can create a directory by right-clicking or selecting the relevant menu option or icon. In the command line, to create a directory you will use a command called `mkdir`, which stands for “make directory.” The directory will be created inside the current directory when you execute the command.

[SAVE THE COMMAND]

`mkdir <directory_name>`

Create a directory

To keep things simple, avoid including spaces in your directory names. If a directory name contains spaces, then you have to make modifications to how you use certain commands in the command line, which makes your tasks more complicated.

[NOTE]

In general, when working in the command line you should avoid using spaces in the names of any files, directories, or other things you create because it can cause complications when you use commands.

As mentioned in the Preface, throughout this book you will work on one project in which you will create and edit files to list the colors of the rainbow, as well as some colors that are not part of the rainbow. This is not

a realistic example of a project typically version controlled with Git; it's a simplified example that will allow you to focus on learning how Git works.

Since the main objective of the sample project is to list the colors of the rainbow, you will give your project directory the name `rainbow`. You will create this project directory in the `desktop` directory so that you can easily see it from the desktop on your computer screen. Go to [Follow Along 1-11](#) to create your directory.

[FOLLOW ALONG 1-11]

```
1 $ cd desktop  
2 desktop $ mkdir rainbow  
3 desktop $ ls  
rainbow
```

What to notice:

- In step 3, the `ls` output will show the `rainbow` project directory you just created along with any other directories or files you have in your `desktop` directory. These are not displayed in the sample output, as the contents on your computer will differ from mine.

If you look on the desktop of your computer, you should see the `rainbow` project directory you just created, as shown in [Figure 1-4](#).

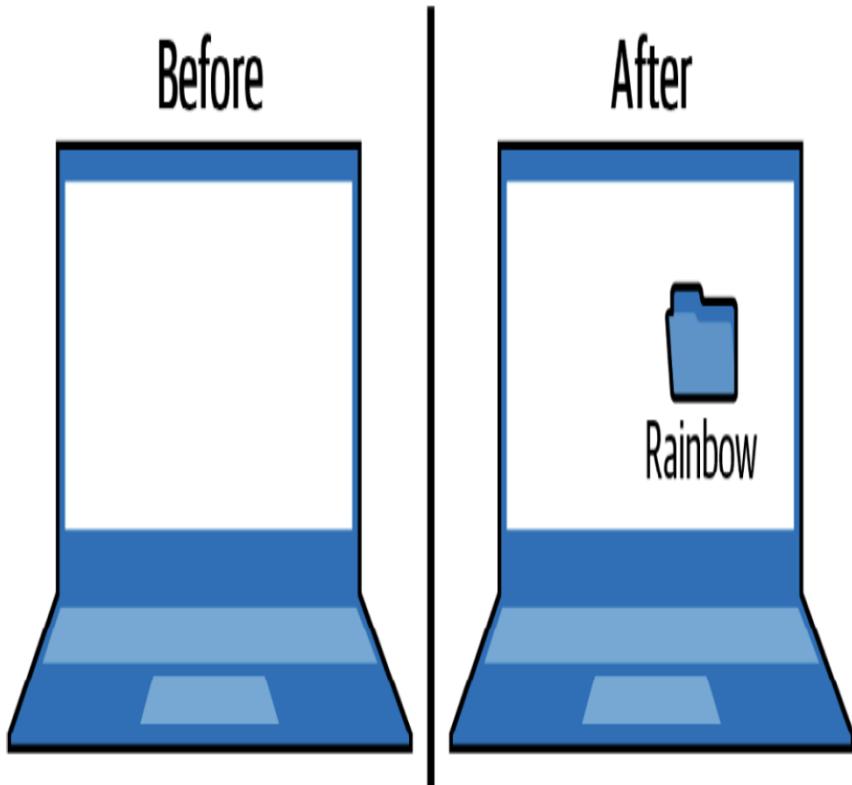


FIGURE 1-4

An example of your desktop before and after you create the `rainbow` project directory

Next, you'll notice that just because you created the `rainbow` project directory, this does not mean you navigated into it in the command line. Go to [Follow Along 1-12](#) to explicitly navigate into the `rainbow` project directory in the command line.

[FOLLOW ALONG 1-12]

1 desktop \$ **cd rainbow**

2 rainbow \$ **pwd**
/Users/annaskoulikari/desktop/rainbow

What to notice:

- In step 2, the command prompt and the output from the **pwd** command indicate you are in the **rainbow** directory.

You have created and navigated into the **rainbow** directory. Now, what happens if you close the command line window and open it back up again?

Closing the Command Line

By default, if you close the command line window and then open it up again, the directory location will reset to the current user directory.

Therefore, you will have to navigate to the directory you want to work in again. Go to [Follow Along 1-13](#) to test this out.

[FOLLOW ALONG 1-13]

1 rainbow \$ **pwd**
/Users/annaskoulikari/desktop/rainbow

2 Close your command line window and then open a new command line window.

3 \$ **pwd**
/Users/annaskoulikari

4 \$ **cd desktop**

5 desktop \$ **cd rainbow**

6 rainbow \$ **pwd**
/Users/annaskoulikari/desktop/rainbow

At this point, we have covered several command line basics. Next, you will set some basic Git configurations.

Setting Git Configurations

Git configurations are settings that allow you to customize how Git works. They consist of variables and their values, and they are stored in a couple of different files. To work with Git, you must set a few configuration variables related to user settings.

Before setting any variables, you'll check if a global Git configuration file exists in your filesystem and, if so, which variables have been set. To do so you will use the `git config` command, passing in the `--global` and `--list` options.

Note that the `git config` command with the `--global` option is an example of a command where it does not matter what your current directory is when you execute it; it will only ever show information about or change information in the global Git configuration file. This is a hidden file called `.gitconfig` that is usually created in the current user directory.

[SAVE THE COMMAND]

`git config --global --list`

List the variables in the global Git configuration file and their values

Go to [Follow Along 1-14](#) to try this out.

[FOLLOW ALONG 1-14]

```
1 $rainbow $ git config --global --list
fatal: unable to read config file '/Users/annaskoulikari/.gitconfig': No
such file or directory
```

What to notice:

- The output shown here indicates what you will see if you have never set any variables in your global Git configuration file. In this case, you will get an error indicating that the file does not exist. If you have set variables in your global Git configuration file, then your output will display the variables you have set and their values.

The two variables we are interested in for this book are `user.name` and `user.email`. Every time someone saves a version of a project (or in other words, makes a commit), Git will note the name and email address of the individual and associate it with that saved version. The `user.name` and `user.email` variables are used to set the name and email address that will be saved for the commits you make. This means that you can see who worked on what in a Git project. Configuring these variables is required in order to work with Git. Keep in mind that anyone who is able to see a list of your commits in any project will be able to view your email address, so make sure to use an address you don't mind other people viewing.

To set these variables in your global Git configuration file, you pass them as arguments to the `git config` command, entering your desired values inside quotation marks (remember, you should not include the angle brackets around the values).

[SAVE THE COMMAND]

`git config --global user.name "<name>"`

Set your name in the global Git configuration file

`git config --global user.email "<email>"`

Set your email address in the global Git configuration file

If in the output of [Follow Along 1-14](#) these variables are set to the values you want them to be, then you may skip [Follow Along 1-15](#). If these variables do not appear or are not set to the values you want them to be, decide what name and email address you want to associate with the work you do on the Rainbow project, and go to [Follow Along 1-15](#). Be sure to swap in the values you want to use in place of my username and email address.

[FOLLOW ALONG 1-15]

1

```
rainbow $ git config --global user.name "annaskoulikari"
```

2

```
rainbow $ git config --global user.email "gitlearningjourney@gmail.com"
```

3

```
rainbow $ git config --global --list
```

```
user.name=annaskoulikari
```

```
user.email=gitlearningjourney@gmail.com
```

What to notice:

- In step 3, in the `git config --global --list` output, the `user.name` and `user.email` variables are set to the values you entered.

Now that you've installed Git and prepared your user settings, the final tool you need to work on a Git project is a text editor.

Preparing a Text Editor

A Git project consists of files and directories that are version controlled. Git can version control all kinds of file types. In the Rainbow project, you will use a text editor to work with simple text files (that have an extension of `.txt`).

A *text editor* is a program that allows a user to edit plain text. You will need a text editor to carry out many of the Follow Along exercises in this book. A text editor is different from a word processor, which is mainly used to edit rich text. Examples of word processors are Microsoft Word and Google Docs; these cannot be used to manage the files of a Git project. *Rich text* is text with styles attached to or embedded within it. If you can see that text is bold or italicized, then that is rich text.

Some text editors make it a lot easier to work on Git projects than others. This book is written in a way that allows you to use whichever text editor you prefer. If you already have a text editor installed that you have used to work on Git projects, then feel free to use that for the exercises in this book. If you're not sure which text editor to use, I highly recommend Visual Studio Code (<https://code.visualstudio.com>): it's a popular text editor, and it is the one I used when working on the Rainbow project in this book. Go to [Follow Along 1-16](#) to prepare a text editor.

[FOLLOW ALONG 1-16]

1 Choose your preferred text editor. If you don't already have one installed, then download a text editor.

2 Open the `rainbow` project directory in a text editor window.

Integrated Terminals

Some advanced text editors (also referred to as *integrated development environments*, or IDEs), such as Visual Studio Code, include a version of a command line within them, usually referred to as an *integrated terminal*, in which you can execute the commands you would normally use in your command line window.

When you use an integrated terminal it may be easier to manage your screen space, because the terminal is already part of the text editor window. However, this depends on personal preference. If your text editor has an integrated terminal, you may choose to use that instead of a separate command line window to execute the Git commands in the rest of the exercises in this book. Either will work just fine, so it is up to you to decide what you prefer.

In the rest of this book, whenever I want to refer to the place where you need to execute commands I will refer to the command line window; however, keep in mind that this also includes the integrated terminal.

Now, with Git installed and your text editor ready to go, you are ready to start working on the Rainbow project!

Summary

In this chapter, you learned a little about Git and saw that it is a useful tool for tracking the history of a project and collaborating with others. You prepared to work on a project using Git by installing an up-to-date version on your computer and setting some basic Git configuration variables.

You also learned some command line basics, such as how to view the contents of directories, how to navigate into and out of them, and how to make them in the first place. Along the way, you created the project directory that you will be using throughout this learning journey, called `rainbow`.

Finally, you prepared a text editor so that you can create and edit files for any Git project you want to work on.

Now you're ready to move on to [Chapter 2](#), where you will actually turn your `rainbow` project directory into a Git repository and start learning about the most important areas when working with Git.

Local Repositories

In the last chapter, you learned some command line basics and you prepared to work with Git by downloading it and configuring some settings.

In this chapter, you will turn the `rainbow` project directory you created in [Chapter 1](#) into a Git repository. You will also learn about four important areas when working with Git: the working directory, the staging area, the commit history, and the local repository. To help you visualize how each of these areas works together, we will build a Git Diagram that will include a representation of each area.

Finally, at the end of this chapter you will add the first file to the `rainbow` project directory, and in the process learn about untracked and tracked files. Let's get started!

Current Setup

At the start of this chapter you should have:

- Downloaded or updated Git on your computer. (You will need a version equal to or greater than 2.28.)
- Created an empty project directory called `rainbow` on your desktop.

- Opened a command line window and navigated into the `rainbow` directory.
- Decided on a text editor to use and opened the `rainbow` project directory in a text editor window.
- Set the `user.name` and `user.email` global Git configuration variables to your name and email address.

Introducing Repositories

A *repository* (also known as a *repo*) is how we refer to a project version controlled by Git. In reality, there are two types of repositories:

- A *local repository* is a repository that is stored on a computer.
- A *remote repository* is a repository that is hosted on a hosting service.

A *hosting service* is a company that provides hosting for projects using Git. As of the time of writing, the main hosting services are GitHub (<https://github.com>), GitLab (<https://about.gitlab.com>), and Bitbucket (<https://bitbucket.org/product>).

In the first part of this book, up through [Chapter 5](#), you will learn about and work only with local repositories. In the second part of the book, from [Chapter 6](#) onward, you will learn how to work with remote repositories as well.

Now that we've distinguished between local and remote repositories, you'll learn about initializing a local repository.

Initializing a Local Repository

A local repository is represented by a hidden directory called `.git` that exists within a project directory. It contains all the data on the changes that have

been made to the files in a project.

To turn a project directory into a local repository you have to *initialize*, or create, the repository. When you initialize a repository, the `.git` directory is automatically created inside the project directory. Because the `.git` directory is a hidden directory, you won't be able to see it unless you explicitly make hidden files and directories visible.

[NOTE]

You should never touch any files or directories inside your `.git` directory. Doing this could have undesirable consequences for your repository. You should never delete this directory unless you want to delete your repository.

In [Follow Along 2-1](#), you will make sure you have your settings configured to be able to view hidden files and directories. (If you need a reminder of how to do this, refer to [“Viewing the Contents of Directories” on page 10](#).) You will then check both in your filesystem window and your command line window to see if there are any visible or hidden files or directories in the `rainbow` project directory.

[FOLLOW ALONG 2-1]

1

Make hidden files and directories visible in your filesystem.

2

Open the `rainbow` project directory in a filesystem window and look at its contents. There should be no visible or hidden files or directories.

3

`rainbow $ ls -a`

.

..

What to notice:

- In step 2, in the filesystem window, you can see that the `rainbow` project directory is empty.
- In step 3, in the command line window, you see that the `rainbow` project directory is empty.

To prepare to build the Git Diagram, we will create a representation of the empty `rainbow` project directory as seen in [Visualize it 2-1](#).

[VISUALIZE IT 2-1]



Project directory: rainbow

A representation of the empty `rainbow` project directory that will contain the Git Diagram

To initialize a Git repository, you use the `git init` command. Your current directory must be the project directory you want to turn into a repository when you execute this command.

Normally Git users initialize a Git repository by using just the `git init` command with no additional options; however, in the Rainbow project, you

will initialize the repository by using the `git init` command with the `-b` option (which is short for `--initial-branch`) and pass in the name `main`. We will cover what branches are in more depth in [Chapter 4](#). For now, all you need to know is that by default Git will create a branch called `master` when you initialize a new local repository. From Git version 2.28 onwards, the name of the initial branch is configurable. I have chosen to use the name `main` instead of `master` in this book, because “master” is not considered inclusive terminology. We will discuss this topic more in [“A Bit of Git History: master and main” on page 48](#).

[NOTE]

If you decide that you want the initial branch in all of your repositories to have a name other than “master,” then you may set a variable called `init.defaultBranch` in your global configuration file. The process is similar to how you set the `user.name` and `user.email` variables in [“Setting Git Configurations” on page 17](#). If the `init.defaultBranch` variable is defined, you can initialize Git repositories with the `git init` command, and the initial branch will have the name that is defined in the configuration.

[SAVE THE COMMAND]

`git init`

Initialize a Git repository

`git init -b <branch_name>`

Initialize a Git repository and set the name for the initial branch to be
`<branch_name>`

Go to [Follow Along 2-2](#) to turn your `rainbow` project directory into a Git repository.

[FOLLOW ALONG 2-2]

1 To see the `.git` directory being created, make sure you have a view of the contents of your `rainbow` project directory in a filesystem window with hidden files and directories enabled.

2 `rainbow $ git init -b main`
Initialized empty Git repository in
`/Users/annaskoulikari/desktop/rainbow/.git/`

3 Go to the `rainbow` project directory in the filesystem window and look at the `.git` directory that was just created. Open the `.git` directory to view the contents inside.

What to notice:

- Git created the `.git` directory inside the `rainbow` project directory.

The first area for which we will add a representation in the Git Diagram is the local repository, represented by the `.git` directory itself. This can be seen in [Visualize it 2-2](#).

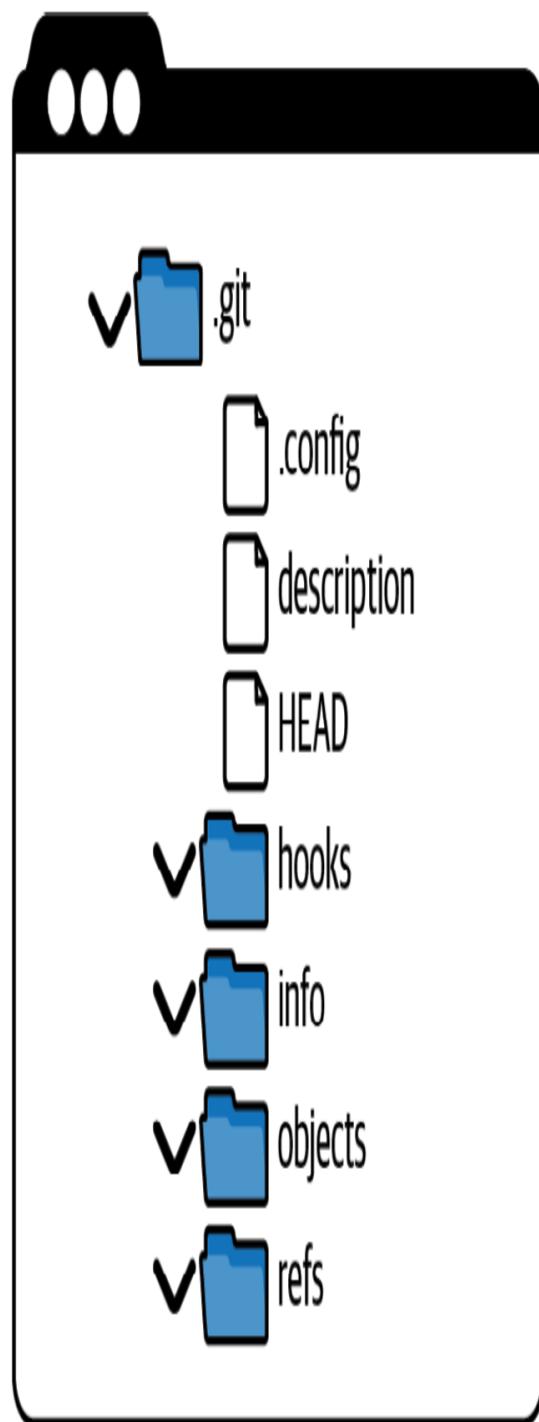
[VISUALIZE IT 2-2]



The Git Diagram with a representation of the local repository

Inside the `.git` directory are various files and directories. Some of them represent the areas of Git that you are going to learn about next. In the following section, we will add representations of these to the Git Diagram as well. As you continue working in the `rainbow` repository, more files and directories will be created in the `.git` directory; you'll learn about some of these as we go along.

[Figure 2-1](#) shows an example of the contents of a `.git` directory in a newly initialized repository.



The contents of a `.git` directory right after executing the `git init` command

FIGURE 2-1

Next, let's explore the different areas you'll interact with when working with Git.

The Areas of Git

There are four important areas to be aware of when you are working with Git:

- Working directory
- Staging area
- Commit history
- Local repository

You learned about the local repository in the previous section. In this section we will cover the rest of the areas and how they all relate to one another, and we will continue building the Git Diagram. Let's start with the working directory.

INTRODUCING THE WORKING DIRECTORY

The *working directory* contains the files and directories in the project directory that represent one version of a project. It is sort of like a workbench. It is where you add, edit, and delete files and directories.

Let's take a look at [Example Book Project 2-1](#) to explore how files are managed in the working directory.

Example Book Project 2-1

Suppose the Book project that I am working on has 10 chapters, and I have 10 text files, one for each chapter: `chapter_one.txt`, `chapter_two.txt`, and so on.

To add each of these chapter files to my project, I would create these files in the working directory.

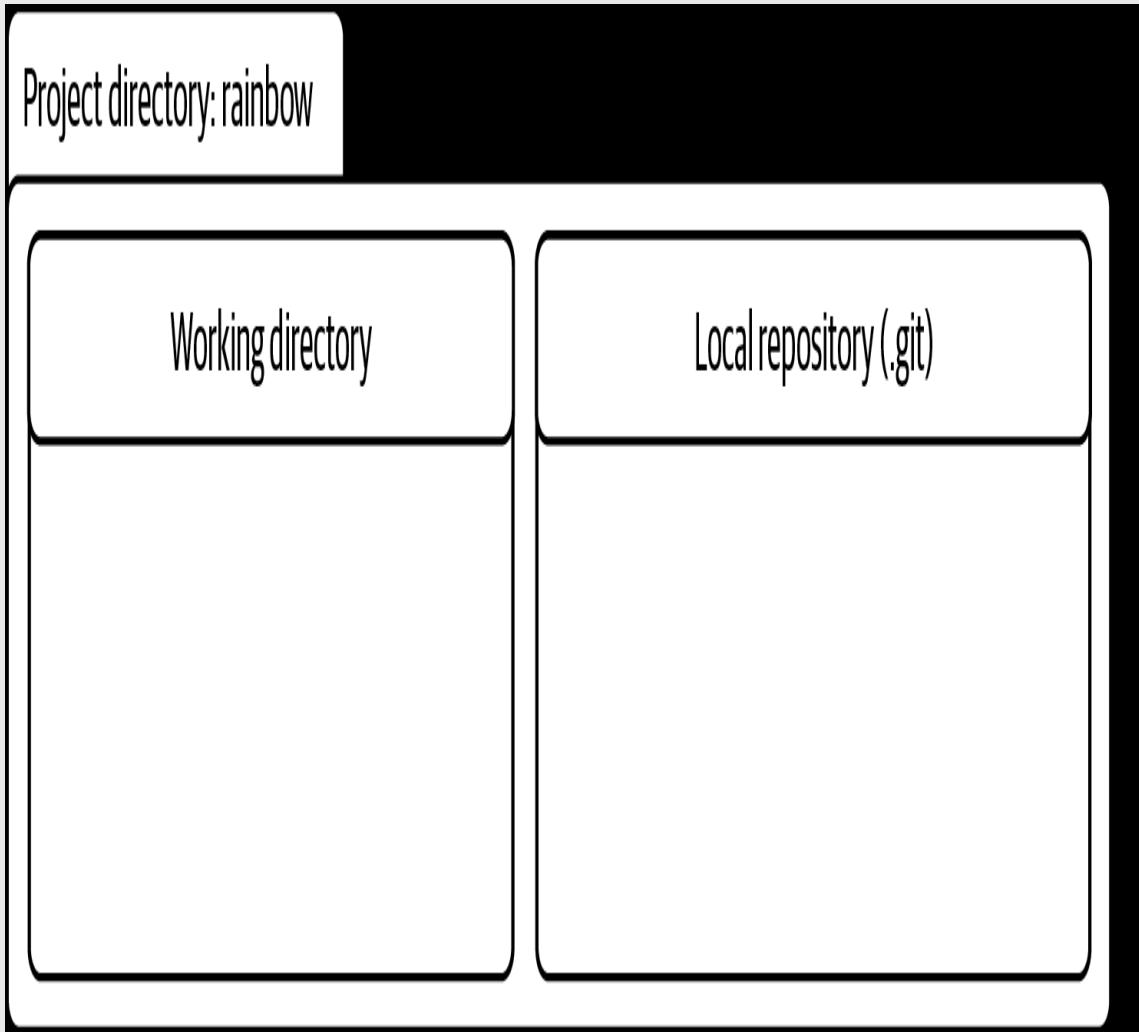
If I wanted to make any changes to the content of those chapters, I would start by editing the files in the working directory.

And finally, if I decided I wanted to remove an entire chapter of my book, the first step would be to delete the corresponding file in the working directory.

From [Example Book Project 2-1](#), you can conclude that the working directory is where you make all the modifications to the content of a project.

In the case of the `rainbow` repository, the working directory is currently empty. To continue building the Git Diagram, we will add a representation of the working directory to it. This is illustrated in [Visualize it 2-3](#).

[VISUALIZE IT 2-3]



The Git Diagram with a representation of the working directory and local repository

In Visualize It 2-3, the `rainbow` project directory contains the local repository within it. However, it is important to know that when people refer to a project version controlled with Git, they will typically refer to the project directory as the repository. For example, in our case, we will make references to “the `rainbow` repository.”

Within the local repository, there are two important areas we want to explore further: the staging area and the commit history. We'll take a look at those next and also discuss the concept of a commit in a little more detail.

INTRODUCING THE STAGING AREA

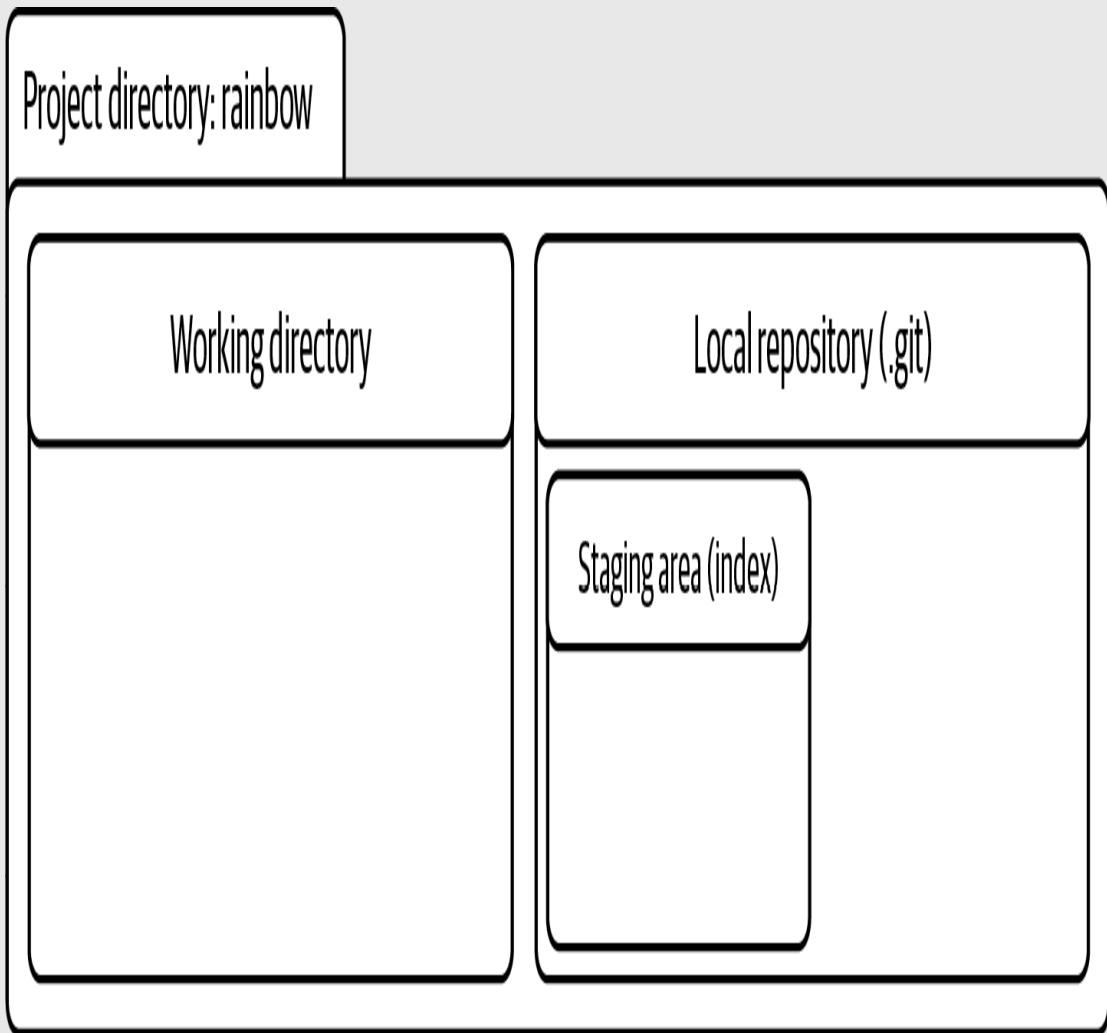
The *staging area* is similar to a rough draft space. It is where you can add and remove files, when you are preparing what you want to include in the next saved version of your project (your next commit). The staging area is represented by a file in the `.git` directory called `index`.

[NOTE]

The `index` file is created only if you have added at least one file to the staging area in your project. In the `rainbow` project directory, you have not yet added any files to the staging area; therefore, the `index` file is not yet visible in the `.git` directory. In [Chapter 3](#), you will add a file to the staging area and you will see the `index` file being created.

You will learn more about why the staging area is so helpful when you practice adding a file to it in [Chapter 3](#). For now, we will add a representation of it inside the local repository in the Git Diagram, as shown in [Visualize it 2-4](#).

[VISUALIZE IT 2-4]



The Git Diagram with a representation of the working directory, staging area, and local repository

The Git Diagram now includes representations of the working directory, the staging area, and the local repository. The final area I want to introduce is the commit history, but before I do that, let's properly explore the concept of a commit.

WHAT IS A COMMIT?

A *commit* in Git is basically one version of a project. You can think of it as a snapshot of a project, or a standalone version of a project that contains references to all the files that are part of that commit.

Every commit has a *commit hash* (sometimes called a *commit ID*). This is a unique 40-character hash composed of letters and numbers that acts like a name for the commit, providing a way to refer to it.

An example of a commit hash is `51dc6ecb327578cca503abba4a56e8c18f3835e1`.

In reality, you only need the first seven characters of a commit hash to refer to a commit. So, for the example hash just given, you can just use `51dc6ec` to refer to the commit.

[NOTE]

Since commit hashes are unique, your commit hashes in the Rainbow project will be different than the ones shown in this book. Keep this in mind while going through the Follow Along exercises in the book.

Now that you have an idea of what commits are, let's introduce the final area that we will add to our Git Diagram, the commit history.

INTRODUCING THE COMMIT HISTORY

The *commit history* is where you can think of your commits existing. It is represented by the `objects` directory inside the `.git` directory. To understand the commit history in depth we would have to dive into the internals of Git, but that is a complex topic that you don't need to know about when learning the basics of how to use Git. For our purposes, all you need to know is that every time you make a commit, it is saved in the commit history.

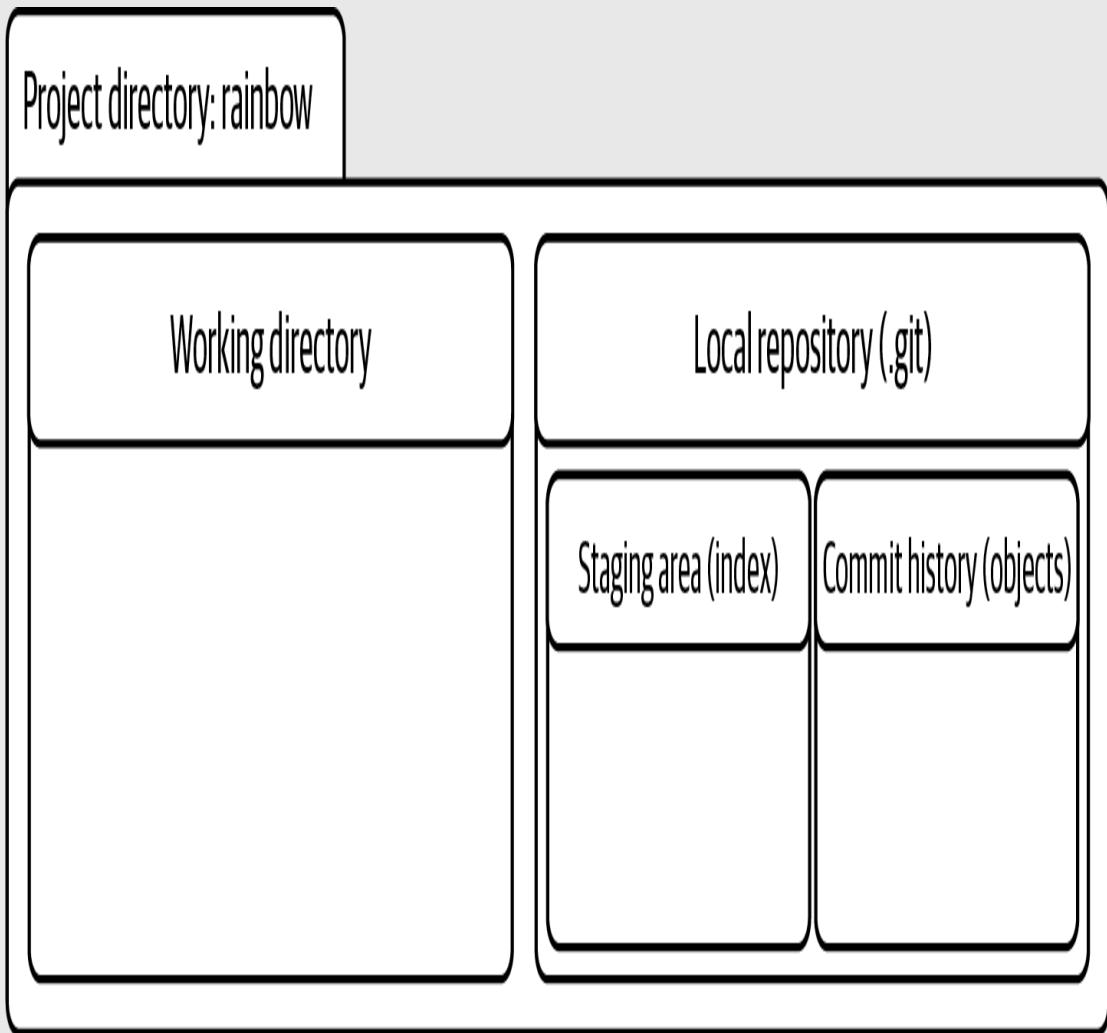
Go to [Follow Along 2-3](#) to identify the commit history in the `rainbow` repository.

[FOLLOW ALONG 2-3]

- 1 In your filesystem window, inside your `rainbow` project directory, look inside the `.git` directory and find the `objects` directory.

In [Visualize it 2-5](#), we add a representation of the commit history to the Git Diagram inside the local repository. This completes the Git Diagram.

[VISUALIZE IT 2-5]



The complete Git Diagram with representations of the working directory, staging area, commit history, and local repository

Now that we have a complete Git Diagram showing the most important areas when working with Git, let's add the first file to the Rainbow project and use a text editor to edit it.

Adding a File to a Git Project

As mentioned earlier in the book, throughout this learning journey you will work on a project in which you list the colors of the rainbow and some colors that are not part of the rainbow. Every time you add a color, you will make a commit to keep track of how the project progresses.

Your first step will be to create a file called `rainbowcolors.txt`, in which you will list the colors. As mentioned in “[Creating a Directory” on page 14](#), it is important that the filename does not include spaces. Then, using the text editor that you prepared in [Chapter 1](#), you will add “Red is the first color of the rainbow.” on line 1 in that file. Go to Follow Along 2-4 to create and edit your file.

[FOLLOW ALONG 2-4]

1 Use your text editor to create a file called `rainbowcolors.txt` inside your `rainbow` project directory.

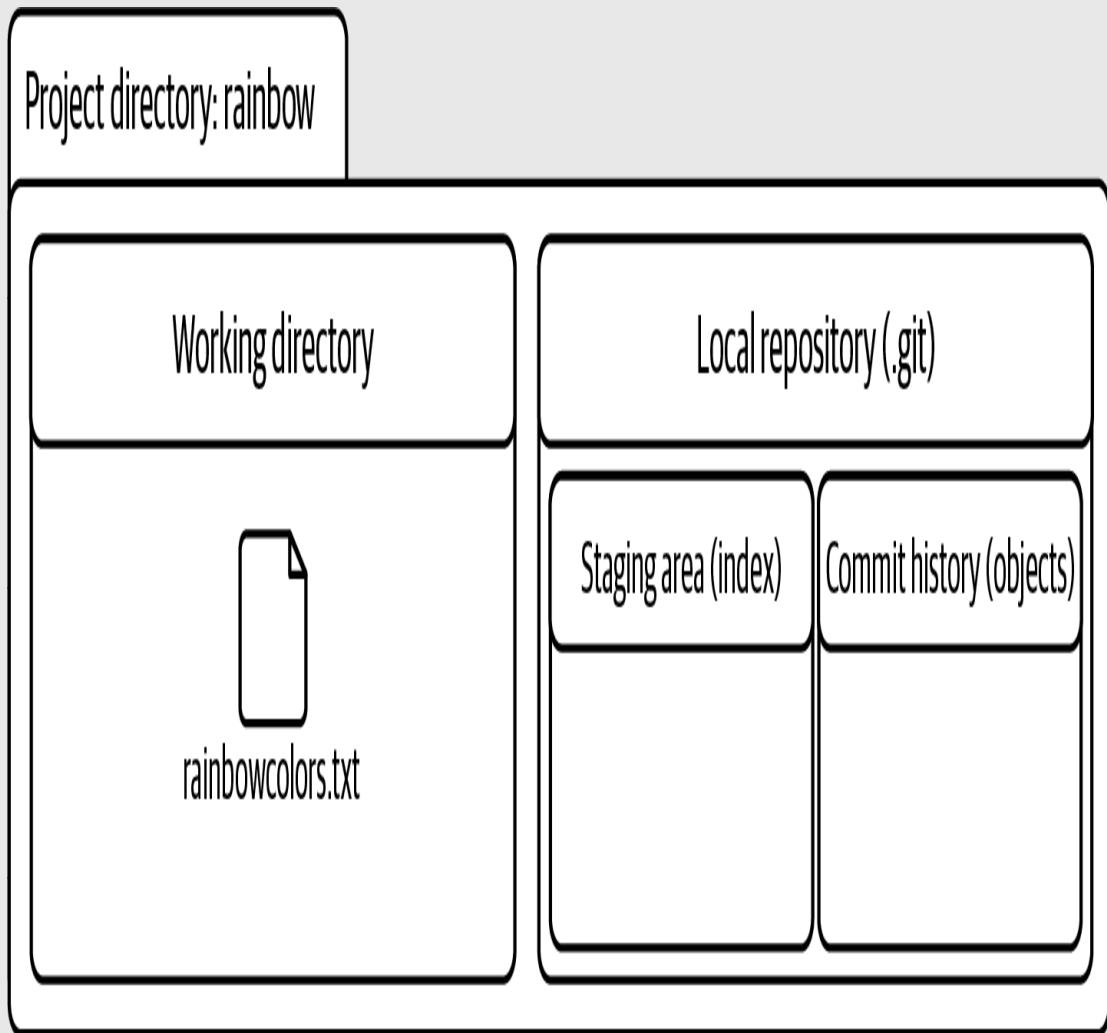
2 On line 1, add “Red is the first color of the rainbow.” and save the file.

What to notice:

- The `rainbowcolors.txt` file is inside the `rainbow` project directory; therefore, it is in the working directory.

Even though the `rainbowcolors.txt` file is in the working directory, it is not part of your repository. It has not been added to the staging area and it has not yet been included in a commit in the commit history. We illustrate this in Visualize It 2-6.

[VISUALIZE IT 2-6]



The `rainbow` project directory after you add the `rainbowcolors.txt` file to the working directory

Since the `rainbowcolors.txt` file is not yet in your repository, it is an untracked file. An *untracked file* is a file in the working directory that Git is not version controlling. It has never been added to the staging area and it has never been included in a commit; therefore, it is not part of the repository.

Once you add a file to the staging area and include it in a commit, the file becomes a *tracked file*. This is a file that is version controlled (in other words, a file that Git tracks).

Every new file in a project version controlled by Git needs to be explicitly added to the staging area and then included in a commit in order to become a tracked file. You will carry out these steps in [Chapter 3](#).

[NOTE]

In the Preface, I mentioned that creating and editing files listing the colors of the rainbow and some colors that are not part of the rainbow is not a very realistic example of a project version controlled with Git. Recall that the aim of this project is to keep the learning journey simple, so that you can focus on how Git works instead of the contents of the files you are editing. The changes you will be making to files in real projects will look very different.

Summary

In this chapter, you turned your `rainbow` project directory into a repository by initializing the repository. We also built a Git Diagram visualizing the four important areas of Git: the working directory, the staging area, the commit history, and the local repository.

You added your first file to your `rainbow` project directory, and you learned about the important distinction between untracked files and tracked files.

In the next chapter, we will walk through the steps involved in making a commit. Along the way, we will touch upon how each of the areas in the Git Diagram is involved in the committing process.

Making a Commit

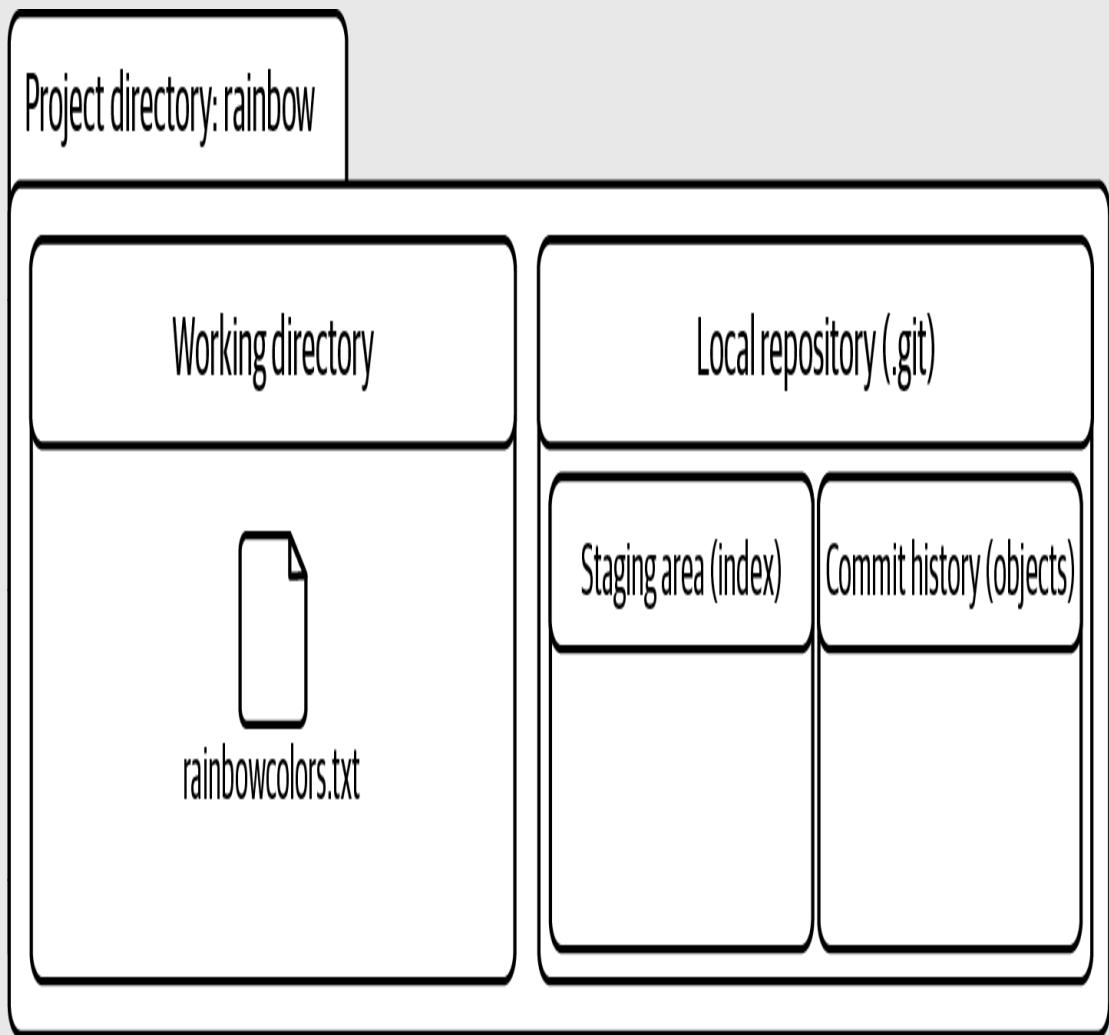
In the last chapter, you learned about the different areas you will interact with when working with Git: the working directory, the staging area, the commit history, and the local repository . We built a Git Diagram of these areas, and you ended the chapter by adding the first file to your `rainbow` project directory.

In this chapter, you will go through the process of making a commit in the Rainbow project and observe how each of the areas in the Git Diagram is involved. I will also introduce two important commands that will assist you in your day-to-day work with Git. The first command will allow you to check the state of your working directory and staging area, and the second will allow you to view a list of commits.

Current Setup

You now have a project directory called `rainbow` that has a `.git` directory inside it, and you also have one file, called `rainbowcolors.txt`, in your working directory. The staging area and commit history are empty because you have not yet made any commits in the `rainbow` repository. This can all be seen in [Visualize it 3-1](#).

[VISUALIZE IT 3-1]



The `rainbow` project directory at the start of [Chapter 3](#) has one untracked file in the working directory

Why Do We Make Commits?

In [Chapter 2](#), you learned that a commit basically represents one version of a project. Every time you want to save a new version of a project, you can make a commit.

Committing is important because it allows you to back up your work and avoid the frustration of losing unsaved work. Once you've made a commit, that work is saved, and you'll be able to go back and look at that commit to see what your project looked like at that point in time.

In terms of when to make commits, there is no strict rule. This can depend on many factors, such as whether you're working on a project on your own or with other people, and what type of project you're working on (e.g., whether you're writing code that needs to compile or documentation for a feature). Finally, if you're working in a team, it may depend on the workflow your team uses and what conventions that team has agreed to (for example, how they use branches, which we'll discuss in [Chapter 4](#)).

A common adage in the world of Git is “commit early, commit often.” If you're just getting started with Git, then I think this is good advice. At this stage, it is preferable to have too many commits than too few commits. Also, once you have a good grasp of Git basics, you can learn about additional tools that Git provides for cleaning up commits.

The Two Steps to Make a Commit

Now that we have touched upon why to commit, let's go over how to commit. Making a commit is a two-step process:

1. Add all the files you want to include in the next commit to the staging area.
2. Make a commit with a commit message.

Throughout this process you will be interacting with the four different areas of Git introduced in [Chapter 2](#): the local repository, the working directory, the staging area, and the commit history.

A useful command in the committing process is the `git status` command. Among other things, it tells you the state of the working directory and the staging area. This is useful because in a project with many files it is easy to lose track of what state the files in your working directory are in (i.e., which ones you have edited) and which files you've added to the staging area.

[SAVE THE COMMAND]

git status

Show the state of the working directory and the staging area

In a project with many files, it can be hard to remember which files are untracked, which files are tracked, and which files you've edited. In the Rainbow project at the moment, you only have one file, but let's take a look at [Example Book Project 3-1](#) to consider the case of a project with more files.

Example Book Project 3-1

The Book project I'm working on consists of 10 files, one for each chapter in the book. After working on the book for a while in between commits, I may forget which chapter files I have edited. I know I worked on chapters 2 and 3, but did I make any changes in chapter 4? If so, I don't want to lose them.

In this case, the `git status` command can help me by telling me which chapter files I've edited, which ones I've added to the staging area, and which ones I have yet to add.

The `git status` command only provides information; it never actually changes anything in your repository. Feel free to use it at any time while you're working on a Git project to learn more about the state of your working directory and staging area.

Go to [Follow Along 3-1](#) to practice using the `git status` command and check on the state of the working directory and staging area in the `rainbow` repository. To carry out the steps in this Follow Along, you must be in the `rainbow` project directory in the command line, as indicated by the command prompt.

[FOLLOW ALONG 3-1]

```
1 rainbow $ git status
On branch main
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
  rainbowcolors.txt
nothing added to commit but untracked files present (use "git add" to
track)
```

What to notice:

- The `git status` output informs you there are `No commits yet`. In other words, the commit history does not contain any commits at this time.
- The `rainbowcolors.txt` file is an untracked file.
- Git gives you the instructions that you need to add the untracked file to the staging area: `use "git add <file>..." to include in what will be committed.`

In [Chapter 2](#), I mentioned that `rainbowcolors.txt` is an untracked file and that for it to become a tracked file it must be added to the staging area and included in a commit. So, let's carry out the first step in the process to make a commit, which is to add any files you want to include in the commit to the staging area.

ADDING FILES TO THE STAGING AREA

To add files to the staging area, you use the `git add` command. If you only want to add individual files that you have edited to the staging area, then you can pass in the filename or filenames to the `git add` command as arguments. To add all the files you have edited or changed in your working directory, you can use the `git add` command with the `-A` option (which stands for “all”). This may be helpful if you have edited many files and don’t want to write out each individual filename in the command line.

[SAVE THE COMMAND]

`git add <filename>`

Add one file to the staging area

`git add <filename> <filename> ...`

Add multiple files to the staging area

`git add -A`

Add all the files in the working directory that have been edited or changed to the staging area

As mentioned in [Chapter 2](#), the staging area allows you to choose which updated files (or changes) will be included in your next commit. The general rule is to group related changes together. This allows you to keep your commits more organized. As you will see in the next section, every commit also has an associated commit message. This can be used to provide a description of what was updated in a specific commit.

This first step of the committing process allows you to be very specific about what you include in a commit. This means that you can edit many

files in your project, but you don't have to save them all in one commit. Let's explore this in [Example Book Project 3-2](#).

Example Book Project 3-2

The Book project that I'm working on has 10 chapters represented by 10 files. Imagine a scenario where I work on chapters 1, 2, and 3 (therefore, I edit `chapter_one.txt`, `chapter_two.txt`, and `chapter_three.txt`).

If I decide that the work I have done on chapter 2 is ready to be committed (saved) but the work on chapters 1 and 3 is not something I want to include in my next commit, then I can add the updated version of the chapter 2 file to the staging area *without* adding the updated versions of the chapter 1 and chapter 3 files to the staging area. This means that *only* the changes in the chapter 2 file will be included in my next commit and officially “saved” in the local repository.

[Example Book Project 3-2](#) illustrates how the staging area gives you a lot of control over what the saved versions of your project (your commits) look like.

In the Rainbow project, the `rainbowcolors.txt` file is the first file you will add to the staging area. This means that when you add this file, the `index` file (which represents the staging area in the `.git` directory) will be created. As you learned in [Chapter 2](#), this file does not exist until you add a file to the staging area. The `index` file is a binary file, which means the actual contents look like gibberish to a human and are not easily understandable. For our purposes, we only need to understand that it represents the staging area. Go

to [Follow Along 3-2](#) to add the `rainbowcolors.txt` file to the staging area and create the `index` file.

[FOLLOW ALONG 3-2]

1 To see the `index` file being created, make sure to have a view of your `rainbow` project directory in the filesystem window with the hidden files and directories enabled and the contents of the `.git` directory visible.

2 `rainbow $ git add rainbowcolors.txt`

3 `rainbow $ git status`

On branch main

No commits yet

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

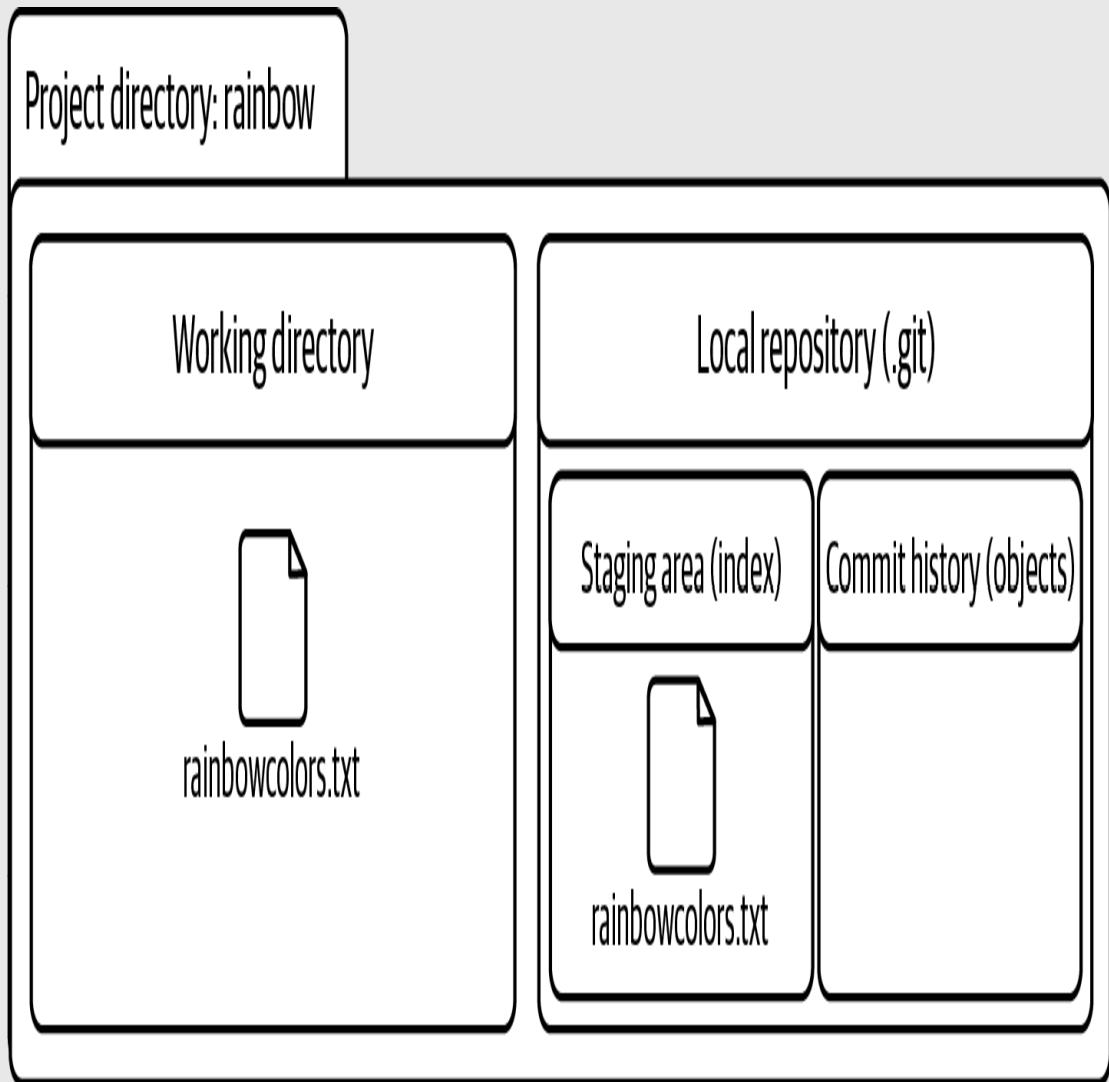
`new file: rainbowcolors.txt`

4 Using your filesystem window, look at the contents of the `.git` directory and identify the newly created `index` file.

What to notice:

- You added the `rainbowcolors.txt` file to the staging area, and in step 3 it is listed under the `Changes to be committed:` section. This is illustrated in Visualize It 3-2.

[VISUALIZE IT 3-2]



The `rainbow` project directory after you add the `rainbowcolors.txt` file to the staging area

The `rainbowcolors.txt` file is now both in the working directory and in the staging area. This is because the `git add` command does not *move* a file from the working directory to the staging area. It *copies* the file from the working directory into the staging area.

With the `rainbowcolors.txt` file in the staging area, you are now ready to move on to the second step in the committing process, which is to actually make a commit with a commit message.

MAKING A COMMIT

It's important to note that *commit* is both a verb and a noun. In Git, the verb *to commit* means to save something, and the noun (*a commit*) means a version of our project. So, to make a commit means to save a version of a project.

To make a commit, you will use the `git commit` command and pass in the `-m` option (which stands for “message”), typing in a message inside quotation marks. The message should usually be a brief description of the changes you made in this version of the project.

[SAVE THE COMMAND]

`git commit -m "<message>"`

Create a new commit with a commit message

Let's consider a sample commit message in [Example Book Project 3-3](#).

Example Book Project 3-3

In my Book project, if I worked on chapter 2, added just that file to the staging area, and wanted to make a commit only with the updates to chapter 2, I might use the commit message “Updated chapter 2”.

Keep in mind that different individuals and teams may have different rules about what to include in a commit message. If you’re working on a Git project with other people, you should check with your collaborators so you’re sure what they expect you to include in these messages.

For the Rainbow project, you will simply be passing in the name of the color you add to the project as the commit message, so that we can easily represent the commits in the Visualize It diagrams. Since the first color you added to the list of rainbow colors was red, in [Follow Along 3-3](#) you will make a commit with the commit message “red”.

[FOLLOW ALONG 3-3]

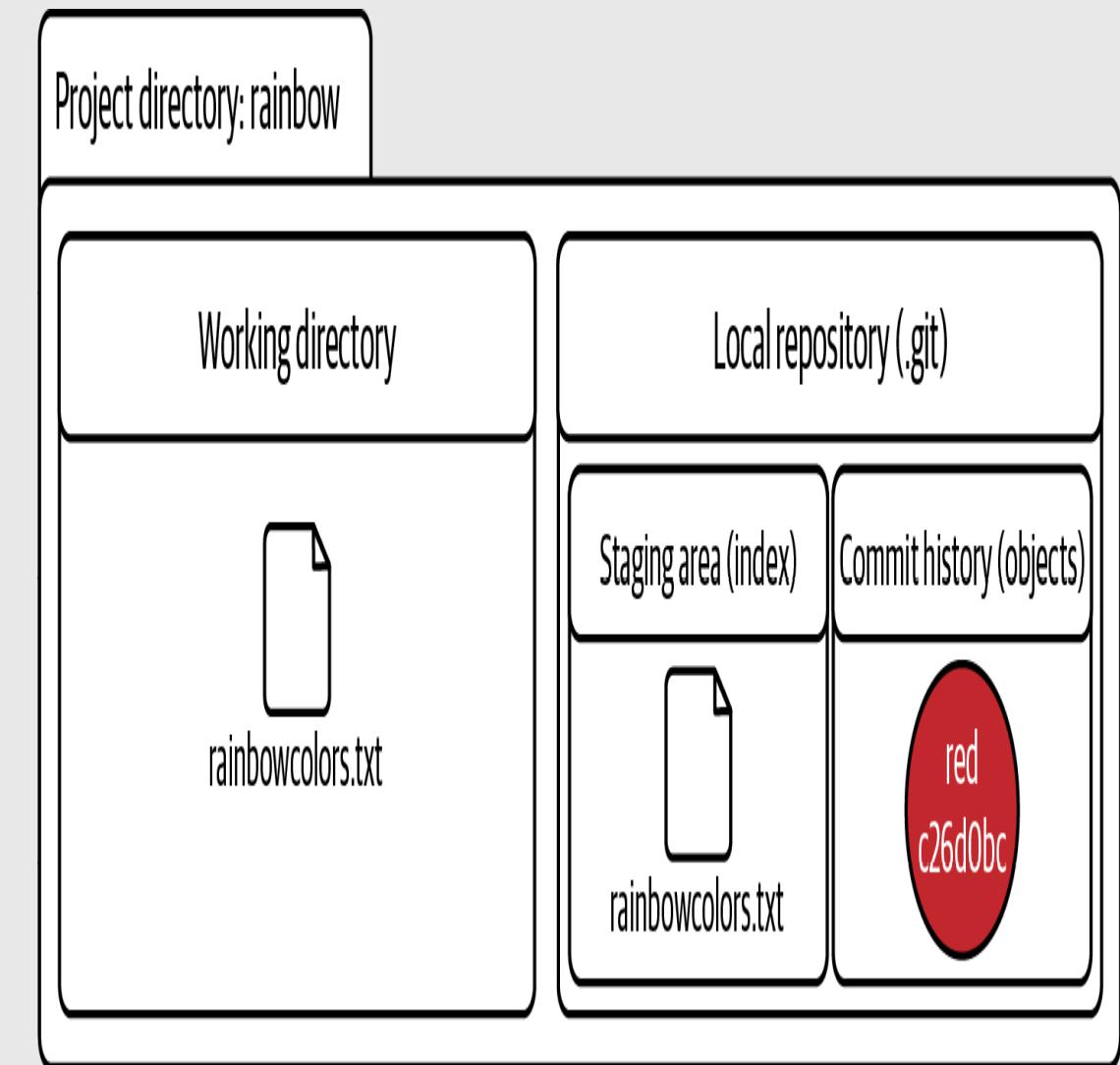
```
1 rainbow $ git commit -m "red"  
[main (root-commit) c26d0bc] red  
 1 file changed, 1 insertion(+)  
  create mode 100644 rainbowcolors.txt
```

What to notice:

- The output of the `git commit` command shows the first seven characters of the commit hash for the red commit, which is `c26d0bc` in this book. The first seven characters of your commit hash will be different, since, as you learned in [“What Is a Commit?” on page 28](#), commit hashes are unique.

In [Visualize it 3-3](#) you can see the red commit in the commit history, with the first seven characters of the commit hash.

[VISUALIZE IT 3-3]



The `rainbow` project directory after you make the red commit

You also learned in the previous chapter that when a new file that was untracked is added to the staging area and included in a commit, it becomes a tracked file, because Git now knows about it. Therefore, the `rainbowcolors.txt` file is now a tracked file.

The rest of the output following the `git commit` command is not really important for our purposes. It simply gives more information about what changed in this commit, and in my experience it's not something most Git users need to examine in detail.

Now that you've made the first commit in the `rainbow` repository, let's take a look at the information you can find about this commit in the commit history.

Viewing a List of Commits

To see a list of commits in the commit history, you use the `git log` command. The `git log` command lists the commits in a local repository in reverse chronological order. It displays four pieces of information about each commit:

1. Commit hash
2. Author name and email address
3. Date and time commit was made
4. Commit message

[SAVE THE COMMAND]

`git log`

Show a list of commits in reverse chronological order

[NOTE]

If the output of the `git log` command goes beyond the size of the command line window, to view the rest of the commits you must press Enter (Return) or use the down arrow. To exit the command, you must enter Q. Right now you have only one commit, so your `git log` output will be very short. However, as you add commits to the Rainbow project, your `git log` output will get longer and you will need to use these keys.

Go to [Follow Along 3-4](#) to practice listing commits.

[FOLLOW ALONG 3-4]

```
rainbow $ git log
1 commit c26d0bc371c3634ab49543686b3c8f10e9da63c5 (HEAD -> main)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 09:23:18 2022 +0100

    red
```

What to notice:

- The `git log` output shows that at the moment you have only one commit, the red commit.
- The full commit hash of the red commit in this book is `c26d0bc371c3634ab49543686b3c8f10e9da63c5`. The commit hash in your project will be different.

- The author of the red commit should match the name and email address you provided in [Chapter 1](#) when you set the `user.name` and `user.email` Git configuration variables.
- The date and time at which the commit was made are displayed.
- Below the date is the commit message, which in this case is `red`.

In the Visualize It diagrams, I will represent commits as circles. Each commit will be displayed in the color used in the commit message, and labeled with its full name or abbreviation (e.g., R for the red commit). Keep in mind that these circles each represent an individual commit that you can see listed in the `git log` output. This is made clear in [Figure 3-1](#).

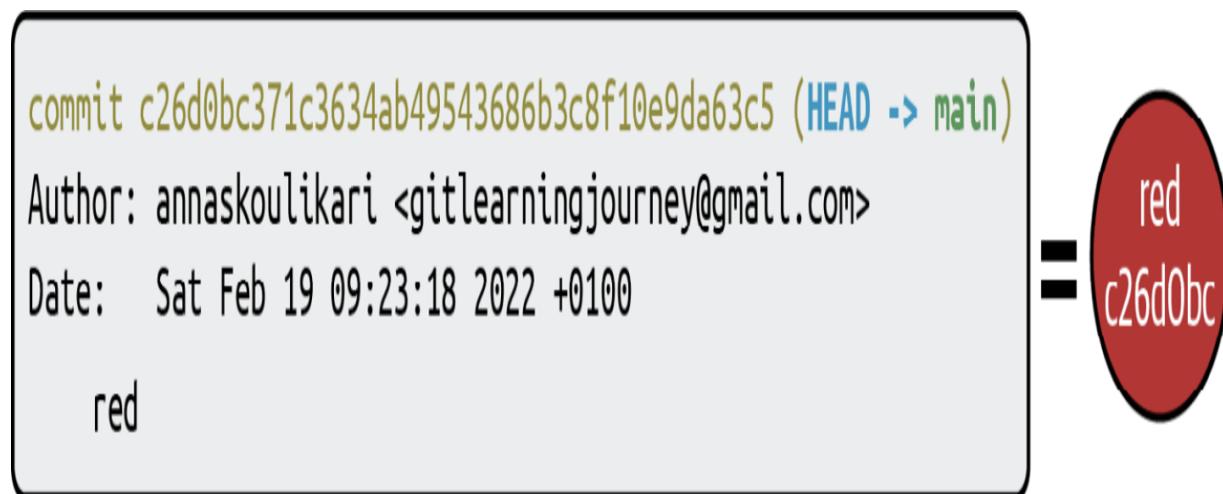


FIGURE 3-1

In this book, commits are represented by circles in the diagrams

There's one more thing that shows up in the `git log` output in [Follow Along 3-4](#) that we have not covered yet. Next to the commit hash, inside the parentheses, you can see the text `HEAD -> main`. `main` is a branch; you'll learn

what branches are and how to work with them in the next chapter, and you'll also learn what `HEAD` is.

Summary

This chapter introduced the two steps of making a commit, namely adding files to the staging area and making a commit with a commit message, and you went ahead and made your first commit in the `rainbow` repository. To assist in the process I introduced the `git status` command, which provides you with information about the state of the files in the working directory and staging area.

You also learned how to list the commits in a local repository using the `git log` command. In the `git log` output, you saw a mention of `main` in parentheses next to the commit hash. In [Chapter 4](#), you'll learn that `main` is a *branch*, and we will explore why and how branches are used in Git.

Branches

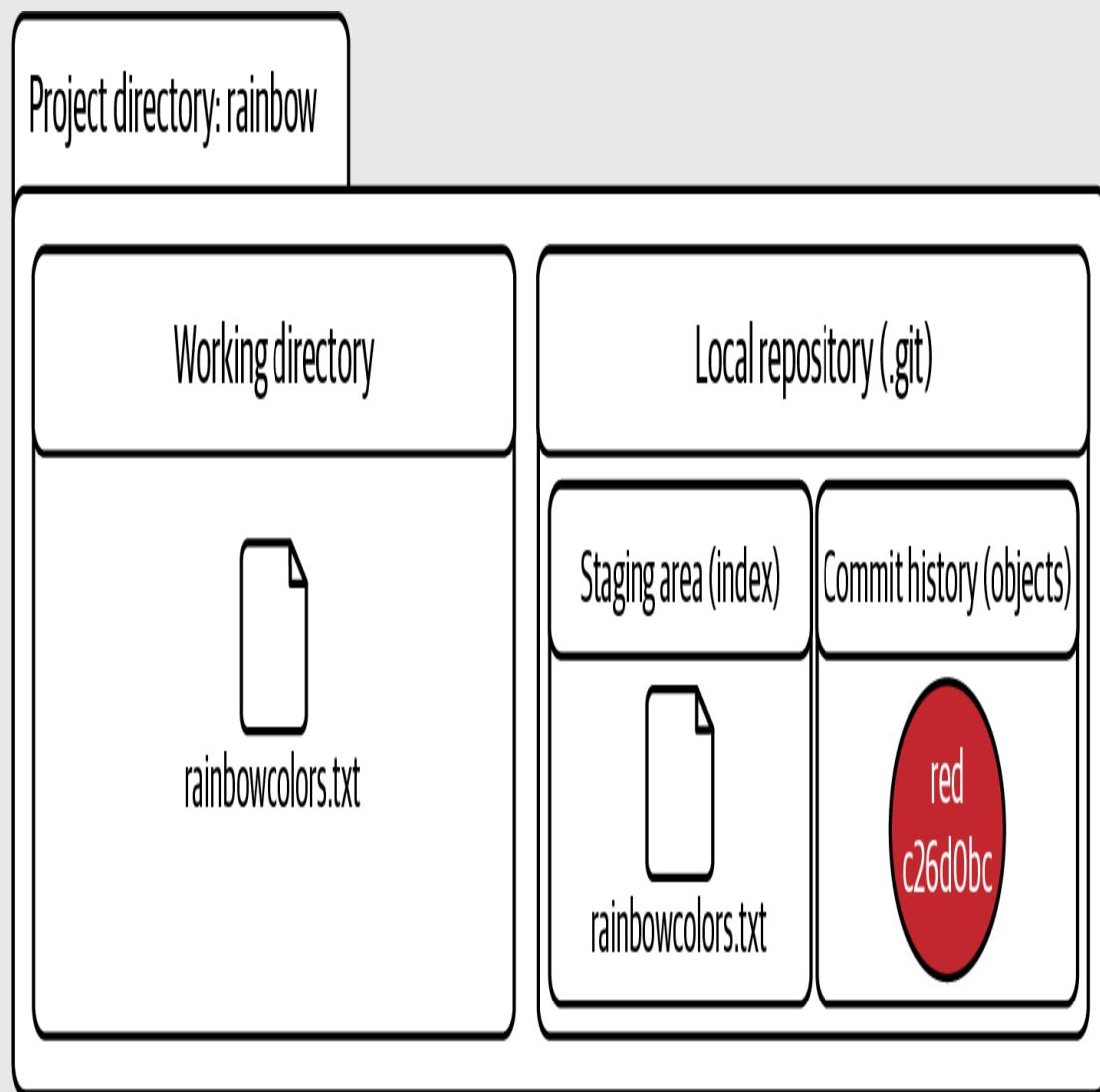
In the last chapter, you learned about the process of making a commit and you made your first commit in the `rainbow` repository.

In this chapter, you'll learn what branches are and why we use them. You will continue making commits in the `rainbow` repository and see how this affects the branches in your project. Finally, you will make a new branch and learn how to switch (or change) onto it. In addition, in the process of making more commits in the `rainbow` repository you will learn about the concept of unmodified and modified files and how commits are linked to one another.

State of the Local Repository

In [Chapter 2](#), we built a Git Diagram with the four important areas of Git: the working directory, the staging area, the commit history, and the local repository. [Visualize it 4-1](#) uses the Git Diagram to show the state of the `rainbow` repository at the start of this chapter.

[VISUALIZE IT 4-1]



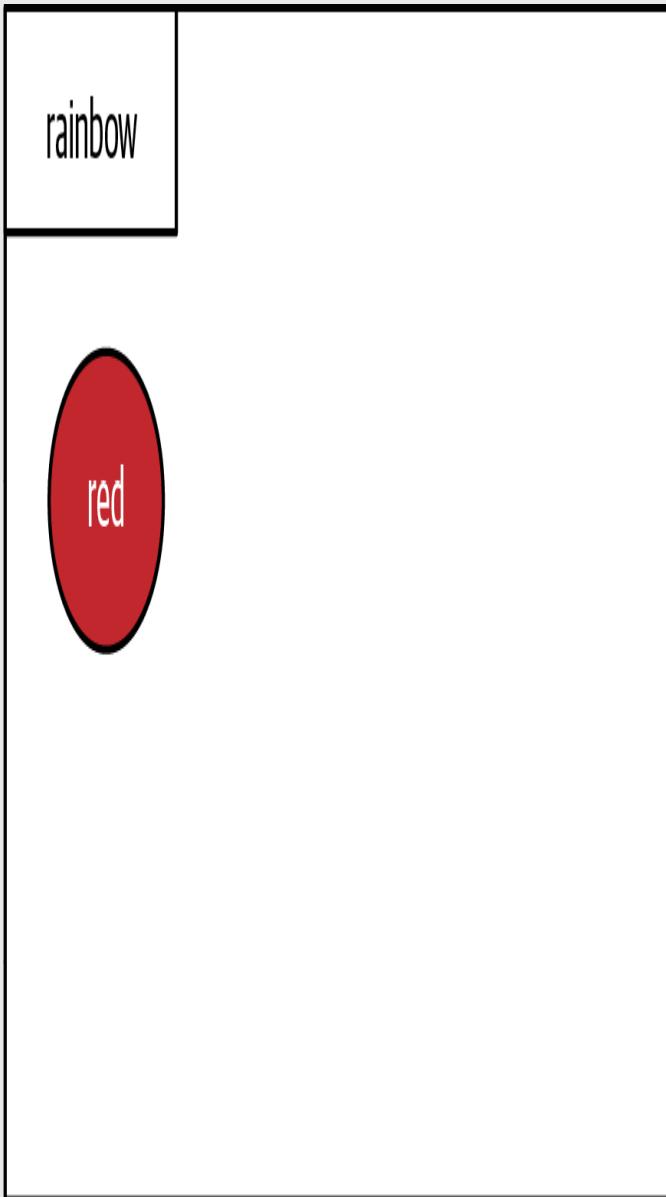
The Git Diagram showing the state of the `rainbow` repository at the start of [Chapter 4](#), with one commit, the red commit

[NOTE]

From this point forward, the Visualize It diagrams will only display the name of the color in the commit, or an abbreviation of the name. I will no longer include the first seven characters of the commit hash.

To focus on the commit history, I'm now going to introduce a new diagram called the Repository Diagram. The Repository Diagram includes only a representation of the commit history of a repository and the relevant branches and references. The local repository is represented by a rectangle, with the name of the repository in the top-left corner. [Visualize it 4-2](#) shows the current state of the `rainbow` repository in the form of a Repository Diagram.

[VISUALIZE IT 4-2]



The Repository Diagram showing the current state of the `rainbow` repository with one commit, the red commit

Why Do We Use Branches?

Before we get into the specifics of branches in Git, I want to explain why they’re so useful. There are two main reasons to use branches:

- To work on the same project in different ways.
- To help multiple people work on the same project at the same time.

You can think of a branch like a line of development. A Git project can have multiple branches (or lines of development). Each of these branches is a standalone version of the project. Different Git projects can use branches in different ways, depending on the needs of the people working on the project.

One common pattern for working with branches is to have one official primary line of development—the main or primary branch—and off of that to create secondary branches, called *topic branches* or *feature branches*, that are used to work on just a specific part of the project. These topic branches are short-lived; they are ultimately combined or incorporated back into the primary branch and then deleted. The two processes you can use to integrate one branch into another are called merging and rebasing. We will cover these in more depth in Chapters [5](#), [9](#), [10](#), [11](#), and [12](#) (yup, they are big topics!).

This pattern of having one primary line of development is the approach you will take in the Rainbow project. To make this a little more concrete, let’s take a look at [Example Book Project 4-1](#) to see how branching might be used in the Book project.

Example Book Project 4-1

Suppose the official branch of my Book project is the `main` branch. I don't want to add work to the official line of development until my editor has reviewed and approved it. So, each time I work on a chapter, I can make a secondary branch, work on that branch, and then submit it to be reviewed by my editor. Once they've approved the work that I have done in that secondary branch, I can combine that branch into the `main` branch.

If at some point I decide to work on the book with a coauthor, we can each work on our own secondary branches. Then, only when the work on a given secondary branch has been approved by both the other author and the editor will it be considered ready to be combined with the `main` branch.

Now that you have an idea of why we use branches, let's get a clearer sense of how they work in Git.

WHAT EXACTLY ARE BRANCHES IN GIT?

Branches in Git are movable pointers to commits. When you list the commits in a local repository using the `git log` command, you can see information about which branches point to which commits. Go to [Follow Along 4-1](#) to see this explicitly in the `rainbow` repository.

[FOLLOW ALONG 4-1]

1

```
rainbow $ git log
commit c26d0bc371c3634ab49543686b3c8f10e9da63c5 (HEAD -> main)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 09:23:18 2022 +0100

red
```

What to notice:

- In the `git log` output, next to the commit hash inside the parentheses you see `HEAD -> main`.

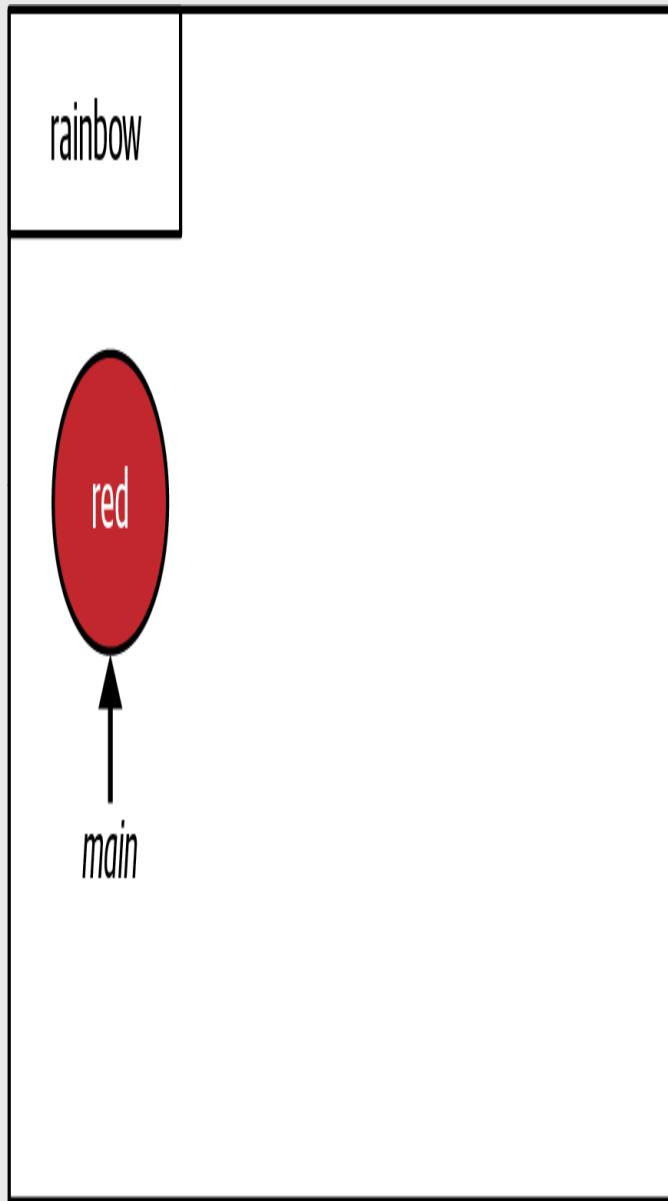
The branch or branches that appear inside the parentheses next to a particular commit hash in the `git log` output are the branches that point to that commit.

[NOTE]

`HEAD` (in capital letters) is not a branch. We'll discuss what it is in "["What Is HEAD?" on page 55.](#)

In the `rainbow` repository, the `main` branch points to the red commit. This is illustrated in [Visualize it 4-3](#).

[VISUALIZE IT 4-3]



The `main` branch points to the red commit in the `rainbow` repository

Later in this chapter, you will continue making commits on the `main` branch, and you will see how the `main` branch will move to point to the new commits.

[NOTE]

The `main` branch in the `rainbow` repository is a local branch. In the first part of this book, when you are working only with a local repository in the Rainbow project, you will work only with local branches. In the second part of the book, once we introduce remote repositories, you will learn about the concept of remote branches and remote-tracking branches.

To get a better idea of the concept of branches as movable pointers to commits, go to [Follow Along 4-2](#).

[FOLLOW ALONG 4-2]

- 1** In a filesystem window, go to the `rainbow` project directory.
- 2** Reveal all the hidden files and directories in the `rainbow` project directory. For more information on how to view hidden files and directories, see ["Viewing the Contents of Directories" on page 10](#).
- 3** Within the `rainbow` project directory in your filesystem window, go to `.git > refs > heads > main`.
- 4** Open the `main` file. On macOS, the file will automatically open in a basic text editor calledTextEdit. On Microsoft Windows, you can choose to open the file with a basic text editor called Notepad.
You will only useTextEdit and Notepad to view the contents of some of the files in the `.git` directory. These text editors are separate from the text editor you are using to manage the files in the Rainbow project.

What to notice:

- In step 4, inside the `main` file you will see the commit hash for the red commit in your `rainbow` repository.

In step 3 of [Follow Along 4-2](#), in order to get to the `main` file, you navigated into the `.git` directory, the `refs` directory, and then the `heads` directory. The term “refs” stands for “references.” The `heads` directory stores a file for each local branch in your local repository. At the moment you only have one local branch, the `main` branch, so there is only one file in this directory. You can think of that file as storing the “head” of that branch; in other words, the latest commit on that branch.

Now that you understand a bit about what branches are and how they work, let’s take a quick look at naming conventions for the primary branch.

A BIT OF GIT HISTORY: MASTER AND MAIN

Normally when you initialize a local repository using just the `git init` command with no options, behind the scenes Git creates a branch called `master`. However, “master” is not considered inclusive terminology, so in recent years, a large part of the Git community has decided to transition to using `main` (or other names) as the default branch name.

This is why in [Chapter 2](#) when you initialized the `rainbow` repository, you used the `git init` command with the `-b` option and passed in the value `main`. The actual word *main* is not special in any way; you could have chosen to give the first branch any other name by passing in another value to the `git init -b` command. When you made the first commit in the `rainbow` repository,

this updated the `main` branch to point to the first commit. In the Rainbow project, the `main` branch is the primary line of development.

In your Git learning journey you will come across many learning resources that still refer to the `master` branch. It is important to understand that there is nothing special about this branch; it's just the default name for the first branch created in Git.

At this point, you're ready to add a new color to the `rainbowcolors.txt` file and make another commit. But before we get to that, I'd like to introduce a few additional states that a file in a Git project can be in.

Unmodified and Modified Files

In [Chapter 2](#), I introduced the concept of untracked files and tracked files. Git knows about the `rainbowcolors.txt` file because it has been included in a commit, and therefore it is a tracked file.

Tracked files in the working directory can be in one of two states.

Unmodified files are files in the working directory that have not been edited since the last commit. Once a file in the working directory has been edited (and saved in the text editor), it becomes a *modified file*. Since your last commit, you have not edited the `rainbowcolors.txt` file; therefore, it is an unmodified file.

[NOTE]

For Git to know that a file has been edited, the file *must* have been saved in the text editor. If you have made edits to a file but you have not saved those changes in your text editor, then Git will view it as an unmodified file.

In [Chapter 3](#) you learned about the `git status` command, which shows you the state of the working directory and the staging area and the difference between the two. The `git status` command actually shows a list of all *modified* files and tells you whether or not they have been added to the staging area. It does not, however, list unmodified files.

Go to [Follow Along 4-3](#) to use the `git status` command to see that the `rainbowcolors.txt` file is an unmodified file, then edit and save the file and use `git status` again to see how its state changes.

[FOLLOW ALONG 4-3]

1 rainbow \$ `git status`

On branch main

nothing to commit, working tree clean

2 In the `rainbow` project directory in your text editor, open the `rainbowcolors.txt` file, add “Orange is the second color of the rainbow.” on line 2, and save the file.

3 rainbow \$ `git status`

On branch main

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

modified: `rainbowcolors.txt`

no changes added to commit (use "git add" and/or "git commit -a")

What to notice:

- In step 1, the `rainbowcolors.txt` file is an unmodified file. It is not yet listed in the `git status` output.
- In step 3, the `rainbowcolors.txt` file is a modified file. It is now listed in the `git status` output.
- The `rainbowcolors.txt` file is not staged for commit; in other words, it has not been added to the staging area.

You just saw how the `rainbowcolors.txt` file went from being an unmodified file to a modified file when you edited it and saved your changes. See [Figure 4-1](#) for an example of what your `rainbowcolors.txt` file should look like now.

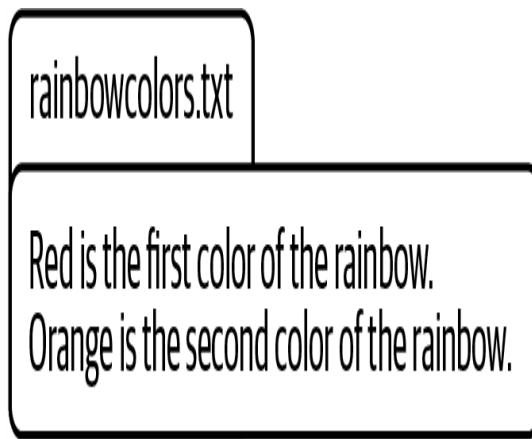


FIGURE 4-1

The `rainbowcolors.txt` file after you add a sentence about the color orange

In [Follow Along 4-4](#), you will add the `rainbowcolors.txt` file to the staging area so that it can be included in your next commit, and you will observe how the `git status` output changes.

[FOLLOW ALONG 4-4]

1 rainbow \$ **git add rainbowcolors.txt**

2 rainbow \$ **git status**

On branch main

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

modified: rainbowcolors.txt

What to notice:

- The `rainbowcolors.txt` file is staged for commit; in other words, it has been added to the staging area.

You have just completed the first step of the committing process, which is to add files to the staging area. Next, you'll make another commit and see how that affects your `main` branch.

Making Commits on a Branch

You are ready to make your second commit in the `rainbow` repository. This time you will add the color orange, so the commit message will be “orange”. Go to [Follow Along 4-5](#) to make your commit.

[NOTE]

If the output of the `git log` command extends beyond the size of the command line window, you must press Enter (Return) or use the down arrow to view the rest of the commits. To exit the command you must type Q.

[FOLLOW ALONG 4-5]

```
1 rainbow $ git commit -m "orange"
[main 7acb333] orange
 1 file changed, 2 insertions(+), 1 deletion(-)

2 rainbow $ git log
commit 7acb333f08e12020efb5c6b563b285040c9dba93 (HEAD -> main)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 09:42:07 2022 +0100

    orange

commit c26d0bc371c3634ab49543686b3c8f10e9da63c5
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 09:23:18 2022 +0100

    red
```

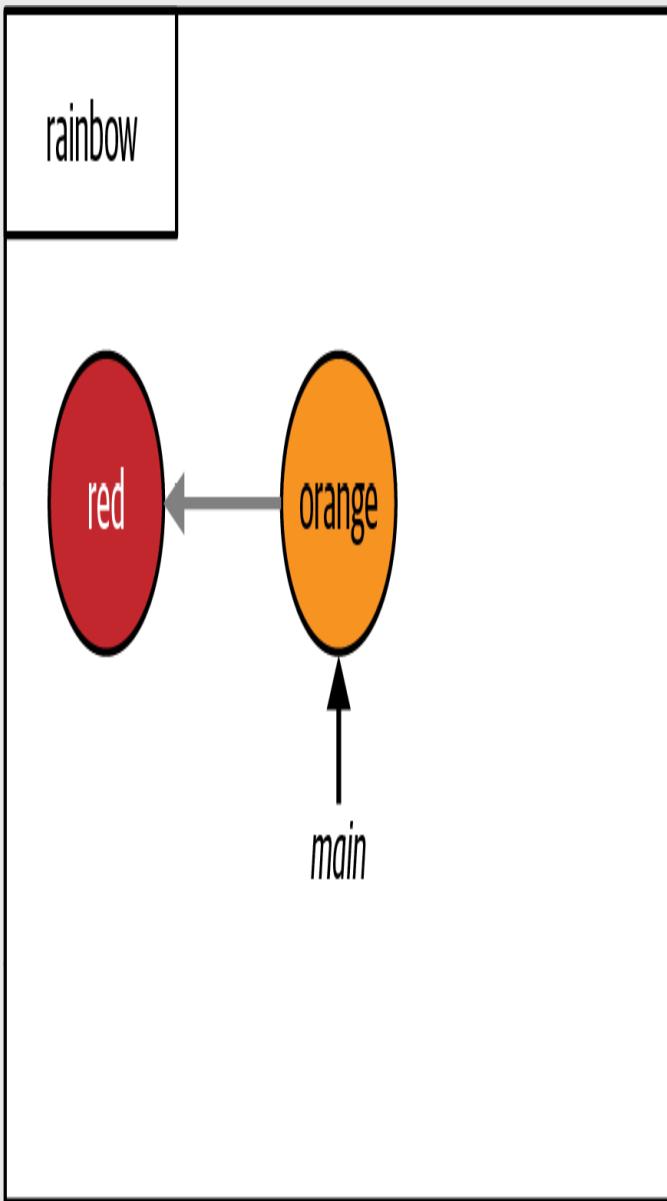
What to notice:

- You made a new commit, the orange commit. In the `rainbow` repository in this book the commit hash for the orange commit is `7acb333f08e12020efb5c6b563b285040c9dba93`. Your commit hash will be different.

- The text `HEAD -> main` appears in parentheses next to the orange commit.

[Visualize it 4-4](#) shows the state of the `rainbow` repository after [Follow Along 4-5](#).

[VISUALIZE IT 4-4]



The `rainbow` repository after you make the orange commit

What to notice:

- There is a second commit, the orange commit.
- The orange commit points back to the red commit.
- The `main` branch points to the orange commit.

In [Visualize it 4-4](#), you can see that there is a gray arrow pointing from the orange commit back to the red commit. This gray arrow represents the parent link. Every commit, other than the very first one in a repository, has a parent commit (some commits can have more than one parent; we will cover that in [Chapter 5](#)). The parent commit of the orange commit is the red commit; this is why the orange commit points back to the red commit.

These parent links describe how commits are linked to one another. Understanding these links allows you to visualize the commit history and keep track of what work has been done on which branches.

[NOTE]

In the Visualize It diagrams, gray arrows are used to represent parent links and black arrows are used to represent branch pointers.

To check which commit is the parent of a given commit, you can use the `git cat-file` command with the `-p` option and pass in a commit hash: `git cat-file -p <commit_hash>`. Most Git users probably won't use this command in their day-to-day work, but since it's a good learning tool, let's see it in action.

Go to [Follow Along 4-6](#) to retrieve the commit hash of the parent of the orange commit. To do this, you'll need to pass in the orange commit's commit hash to the `git cat-file -p` command. The easiest way to retrieve the

commit hash for a particular commit is to look at the list of commits produced by the `git log` command and copy the entire commit hash in the command line. Alternatively, you can look back at the `git commit` output in [Follow Along 4-5](#) and pass in the first seven characters of the commit hash, which is shown there.

[FOLLOW ALONG 4-6]

1 Retrieve the commit hash for the orange commit (you can copy this from the `git log` output in Follow Along 4-5). You must pass this commit hash as an argument to the `git cat-file -p` command in step 2 of this Follow Along. You may copy and paste the entire commit hash or just enter the first seven characters, as shown here.

2 rainbow \$ `git cat-file -p 7acb333`
tree 407fe6a858cd7f157405e013a088fdc1c61f0a40
parent c26d0bc371c3634ab49543686b3c8f10e9da63c5
author annaskoulikari <gitlearningjourney@gmail.com> 1645260127 +0100
committer annaskoulikari <gitlearningjourney@gmail.com> 1645260127 +0100
orange

What to notice:

- In the `git cat-file -p` output, you can see that next to `parent` it references the commit hash of the red commit in this book. In your output, it will reference the commit hash of your red commit.

You've now made a second commit in your `rainbow` repository, and in [Visualize it 4-4](#) you observed that as you make commits on a branch, the

branch pointer moves to point to the latest commit. Next, let's go over how to make other branches so that you can work on other lines of development.

Creating a Branch

At the moment, you only have one local branch, called `main`, in your `rainbow` repository. To list the branches in a local repository, you can use the `git branch` command. To create a new branch, you can pass the name of a branch that doesn't exist yet to this command. Note that branch names cannot contain spaces.

[SAVE THE COMMAND]

`git branch`

List local branches

`git branch <new_branch_name>`

Create a branch

As with commit messages, as discussed in [Chapter 3](#), if you're working with other people on a project you should check if there are any existing rules that determine how to name your branches. Since there are no specific branch naming rules in the Rainbow project, you will use the generic name `feature` for the branch you will create in [Follow Along 4-7](#). Keep in mind, however, that in a real Git project the branch name would usually be more descriptive, relating to the feature or topic you're working on.

[FOLLOW ALONG 4-7]

1 rainbow \$ **git branch**
* main

2 rainbow \$ **git branch feature**

3 rainbow \$ **git branch**
feature
* main

4 rainbow \$ **git log**
commit 7acb333f08e12020efb5c6b563b285040c9dba93 (**HEAD -> main, feature**)

Author: annaskoulikari <gitlearningjourney@gmail.com>

Date: Sat Feb 19 09:42:07 2022 +0100

orange

commit c26d0bc371c3634ab49543686b3c8f10e9da63c5

Author: annaskoulikari <gitlearningjourney@gmail.com>

Date: Sat Feb 19 09:23:18 2022 +0100

red

5 Using your filesystem window, go to `.git > refs > heads` to see which files are in there now.

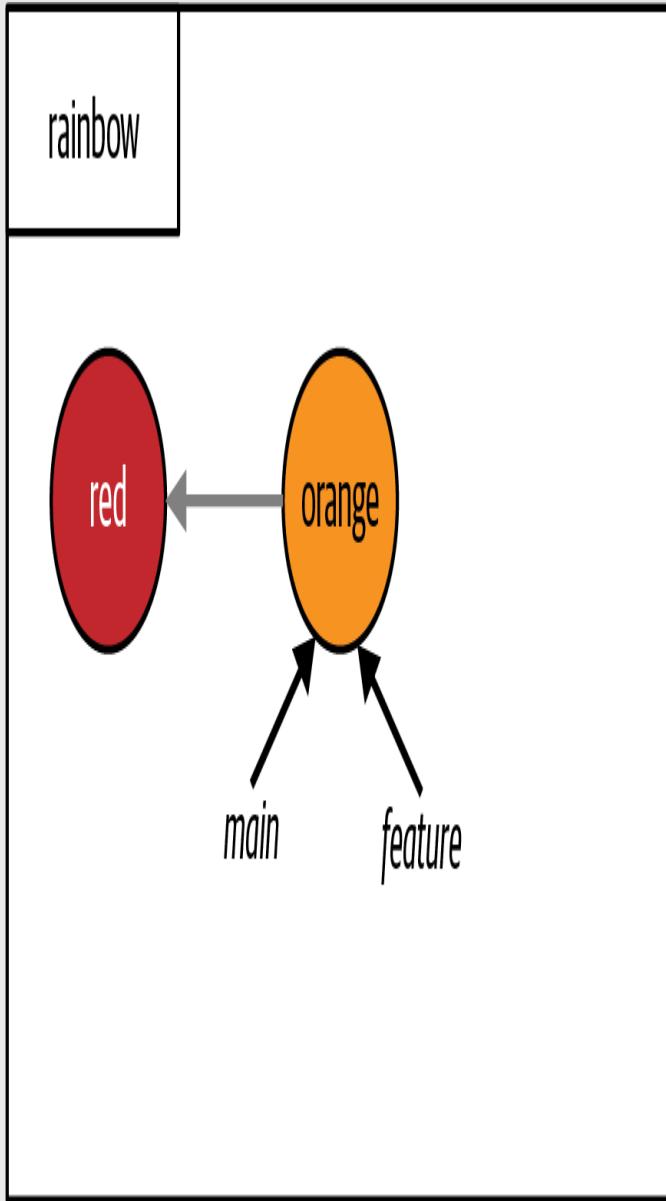
6 Open the `feature` file. The file should contain a commit hash.

What to notice:

- In step 2, you made a new branch called `feature` that points to the orange commit.

The state of the `rainbow` repository after you complete [Follow Along 4-7](#) is illustrated in [Visualize it 4-5](#).

[VISUALIZE IT 4-5]



The `rainbow` repository after you make the `feature` branch

In [Visualize it 4-5](#) you can see that there are now two arrows, representing the `main` and `feature` branches, pointing to the orange commit. A new branch will initially point to the commit that you were on when you made the

branch. In this case, you can say that you “made the `feature` branch off of the `main` branch.” That is why the `feature` branch and `main` branch both now point to the same commit.

In the `git log` output from [Follow Along 4-7](#), in the parentheses next to the orange commit, you should see `HEAD → main, feature`. As you know, `main` and `feature` are branches—but what is `HEAD`?

What Is HEAD?

At any given point in time, you are looking at a particular version of your project. Therefore, you are on a particular branch which is pointing to a commit. `HEAD` is simply a pointer that tells you which branch you are on. The name `HEAD` is always in capital letters, but this is simply a convention; it is not an acronym.

[NOTE]

There are times where you can be on a commit that is not pointed to by a branch. Git calls this “detached `HEAD` state.” We’ll explore this further in [“Checking Out Commits” on page 80](#).

Go to [Follow Along 4-8](#) to explore what `HEAD` is by taking a look at the `.git` directory.

[FOLLOW ALONG 4-8]

1

Using your filesystem window, go to `rainbow > .git > HEAD`.

2

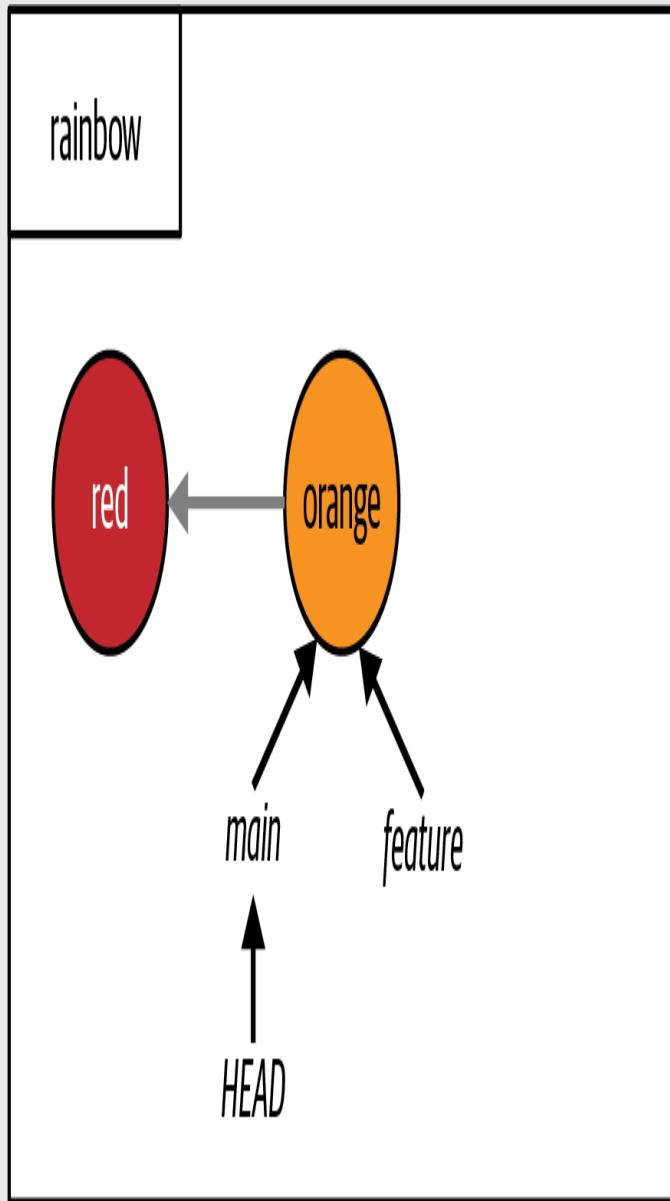
Open the `HEAD` file. The contents should be `ref: refs/heads/main`.

What to notice:

- The `HEAD` file contains `ref: refs/heads/main`, which is a reference to the `main` file that represents the `main` branch.

This is illustrated in [Visualize it 4-6](#).

[VISUALIZE IT 4-6]



In the `rainbow` repository, `HEAD` is pointing to `main`

In [Visualize it 4-6](#), an arrow shows `HEAD` is pointing to `main`. This indicates you're currently on the `main` branch.

[NOTE]

`HEAD` (in capital letters) should not be confused with the `heads` directory that can be found in `.git > refs > heads`. The `heads` directory stores a file for every local branch in your local repository, while `HEAD` indicates which branch you are on by referencing one of the files inside the `heads` directory. You can distinguish them because `HEAD` is always in capital letters.

Another way of knowing which branch you’re currently on is to look at the output of either the `git branch` command or the `git log` command. In the `git branch` output, the branch you are currently on will have an asterisk next to it. If you look at the output in [Follow Along 4-7](#), you can see the asterisk is next to the `main` branch. In the `git log` output, `HEAD` will point to the branch you are on inside the parentheses.

Now that you’ve created a new branch, you’re ready to start using it—but for the time being, you’re still on the `main` branch. Next, you will switch branches, moving the `HEAD` pointer to your new `feature` branch.

Switching Branches

To work on another branch (or line of development) in a Git project, you have to switch onto that branch. Another way of saying this in Git terminology is that you have to “check out” another branch.

You currently have two branches, `main` and `feature`. But as you’ve just seen, just because you make a branch in Git does not mean that you automatically switch onto that branch. You must explicitly instruct Git that you want to switch onto a branch. You can do this using either the `git switch` command

or the `git checkout` command, passing in the name of the branch that you want to switch onto.

[NOTE]

If you have a version of Git that is older than version 2.23, then you won't have access to the `git switch` command and you must use the `git checkout` command. The `git checkout` command is available to all Git users.

[SAVE THE COMMAND]

`git switch <branch_name>`

Switch branches

`git checkout <branch_name>`

Switch branches

The only purpose of the `git switch` command is to switch branches, while the `git checkout` command can do more things. We will talk more about `git checkout` in [“Checking Out Commits” on page 80](#).

From now on, in the Follow Alongs in this book you will use the `git switch` command, as this is the specialized command included in the latest versions of Git for this purpose. However, you can always choose to use the `git checkout` command instead, as they are equivalent.

The `git switch` (or `git checkout`) command does three things when used to switch branches:

1. It changes the `HEAD` pointer to point to the branch you are switching onto.
2. It populates the staging area with a snapshot of the commit you are switching onto.
3. It copies the contents of the staging area into the working directory.

[NOTE]

If you need a refresher on what the staging area and working directory are, refer back to [“The Areas of Git” on page 26](#).

In short, when you change branches you end up changing the commit that you’re looking at, provided that the two branches point to two different commits. At the moment, both of the branches you have in the `rainbow` repository point to the same commit, so only the first action will take place, and the commit you are on will not change. In [Chapter 5](#), you will go over an example where all three of the actions listed here take place.

Go to [Follow Along 4-9](#) to switch onto the `feature` branch.

[FOLLOW ALONG 4-9]

1 rainbow \$ **git branch**

feature

* main

2 rainbow \$ **git switch feature**

Switched to branch 'feature'

3 rainbow \$ **git branch**

* feature

main

4 rainbow \$ **git log**

commit 7acb333f08e12020efb5c6b563b285040c9dba93 (**HEAD -> feature, main**)

Author: annaskoulikari <gitlearningjourney@gmail.com>

Date: Sat Feb 19 09:42:07 2022 +0100

orange

commit c26d0bc371c3634ab49543686b3c8f10e9da63c5

Author: annaskoulikari <gitlearningjourney@gmail.com>

Date: Sat Feb 19 09:23:18 2022 +0100

red

5 Using your filesystem window, go to `rainbow > .git > HEAD`, and open the `HEAD` file in a new window. You will see that the contents of the file have changed and it now refers to the `feature` branch, `refs/heads/feature`.

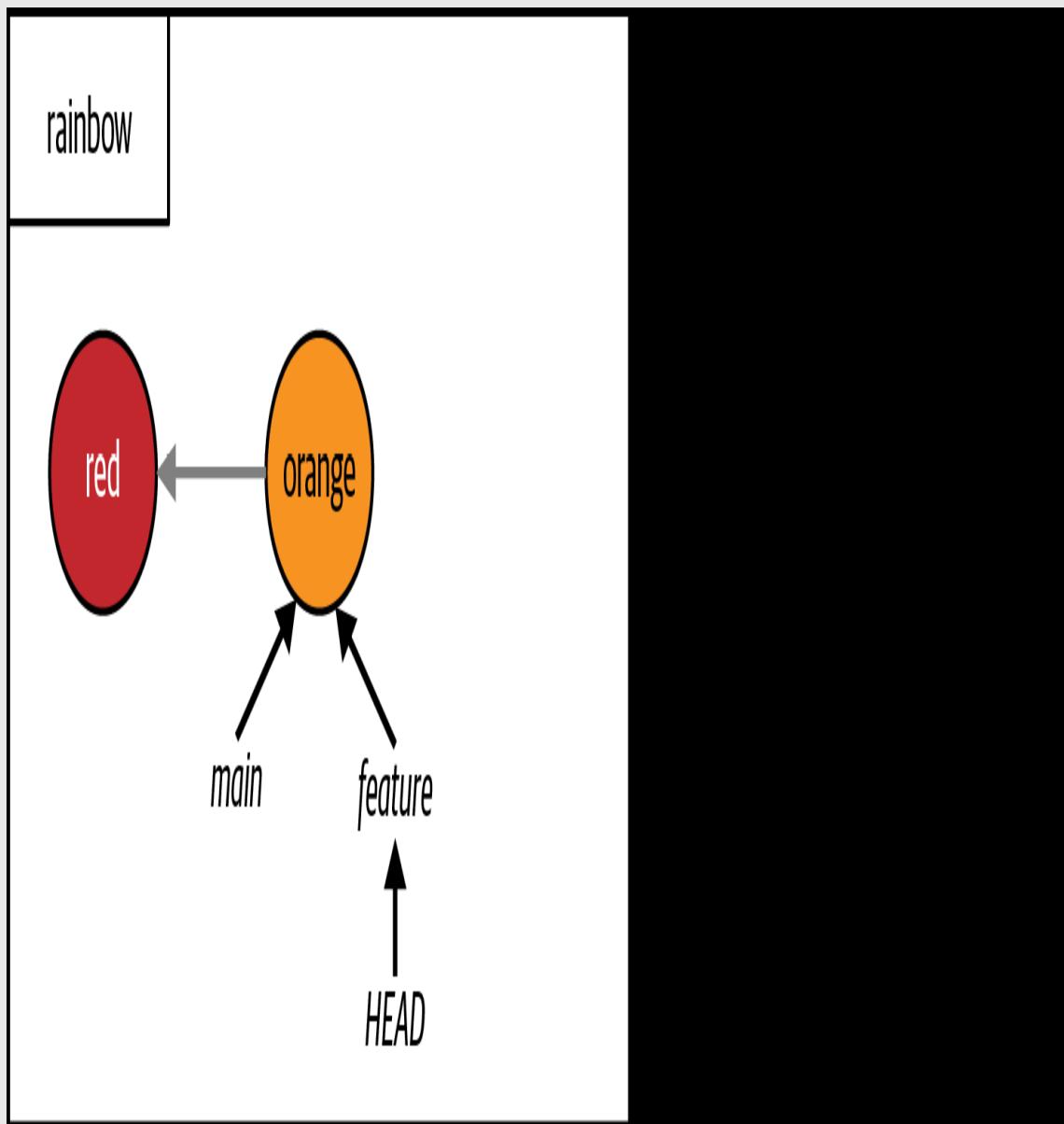
What to notice:

- In step 4, the `git log` output shows that `HEAD` points to the `feature` branch.

- You have switched onto the `feature` branch, so this is where you'll be working now.

These observations are illustrated in [Visualize it 4-7](#).

[VISUALIZE IT 4-7]



The `rainbow` repository after you switch onto the `feature` branch

As you can see, switching branches changed the contents of the `HEAD` file in the `.git` directory. This is illustrated in [Figure 4-2](#).

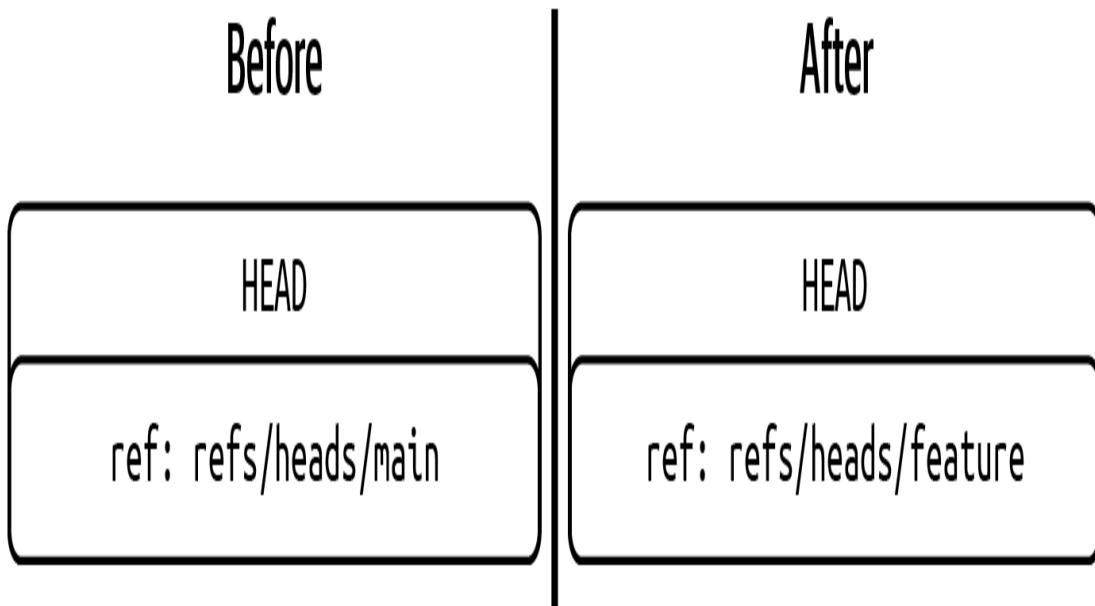


FIGURE 4-2

The contents of the `HEAD` file before and after you switch from the `main` branch onto the `feature` branch in the `rainbow` repository

Now that you have switched onto the `feature` branch, let's see what happens when you work on it.

Working on a Separate Branch

You are now on the `feature` branch, and in [Follow Along 4-10](#) you're going to add the color yellow to the `rainbow` repository.

[FOLLOW ALONG 4-10]

1 In the `rainbow` project directory in your text editor, in the `rainbowcolors.txt` file, add “Yellow is the third color of the rainbow.” on line 3. Then save the file.

2 `rainbow $ git add rainbowcolors.txt`

3 `rainbow $ git commit -m "yellow"`

`[feature fc8139c] yellow`
`1 file changed, 2 insertions(+), 1 deletion(-)`

4 `rainbow $ git log`

`commit fc8139cbf8442cdbb5e469285abaac6de919ace6 (HEAD -> feature)`

`Author: annaskoulikari <gitlearningjourney@gmail.com>`

`Date: Sat Feb 19 10:09:59 2022 +0100`

`yellow`

`commit 7acb333f08e12020efb5c6b563b285040c9dba93 (main)`

`Author: annaskoulikari <gitlearningjourney@gmail.com>`

`Date: Sat Feb 19 09:42:07 2022 +0100`

`orange`

`commit c26d0bc371c3634ab49543686b3c8f10e9da63c5`

`Author: annaskoulikari <gitlearningjourney@gmail.com>`

`Date: Sat Feb 19 09:23:18 2022 +0100`

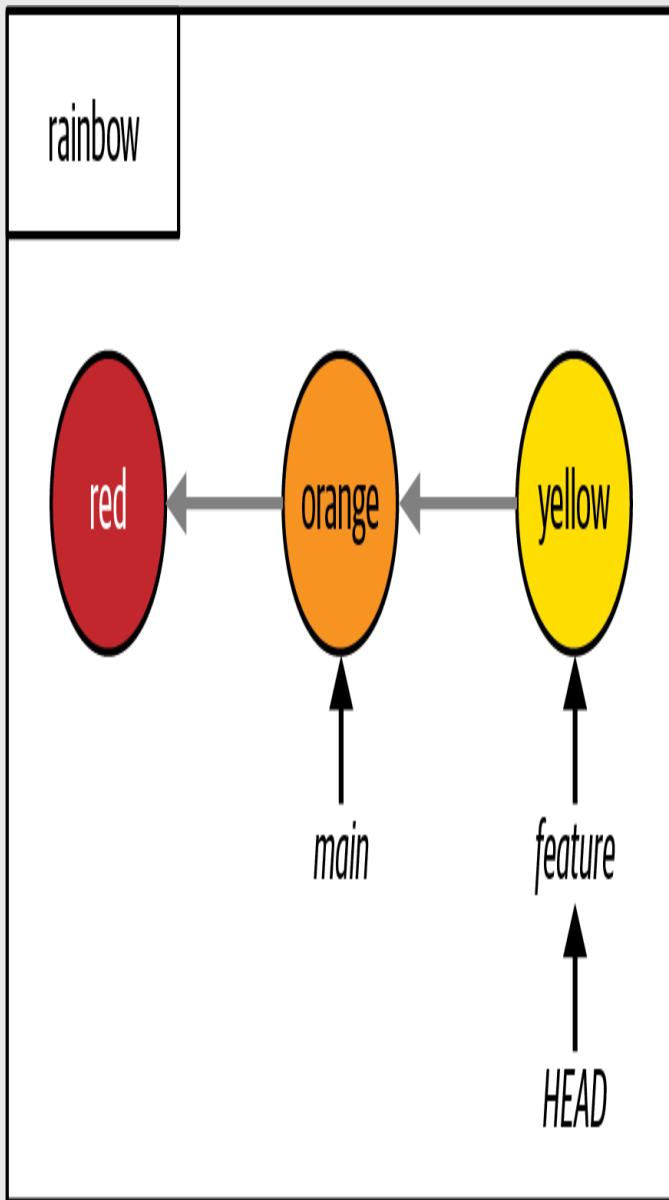
`red`

What to notice:

- The `feature` branch points to the latest commit, the yellow commit.
- The `main` branch still points to the orange commit.

These observations are illustrated in [Visualize it 4-8](#).

[VISUALIZE IT 4-8]



The `rainbow` repository after you make the yellow commit

As mentioned previously, when you make a commit, it is the branch you're currently on that updates to point to the new commit. The `main` and `feature` branches no longer point to the same commit because `feature` has updated to point to the new yellow commit. `HEAD` continues to point to the `feature` branch.

Summary

This chapter introduced the concept of branches as different lines of development and showed how, in practice, they are movable pointers to commits. We explored why you might want to use them, and you learned how to list, create, and change branches. We covered why the first branch in the `rainbow` repository is called `main`, and why in other learning resources teaching Git you may come across a branch called `master`. You also learned that `HEAD` is a pointer to the branch you're currently on.

While making more commits, you observed how tracked files change state from unmodified to modified when they are edited. You also observed that it's the branch you're on that moves to point to the latest commit you make in a local repository, and you saw how commits are linked to each other through parent links.

Now that we've covered how to use branches to work on different lines of development simultaneously, let's continue on to [Chapter 5](#) to start learning how to combine these different lines of development by merging.

Merging

In the last chapter, you learned about branches, and we discussed how they allow you to work on the same project in different ways and to collaborate with other people on a project.

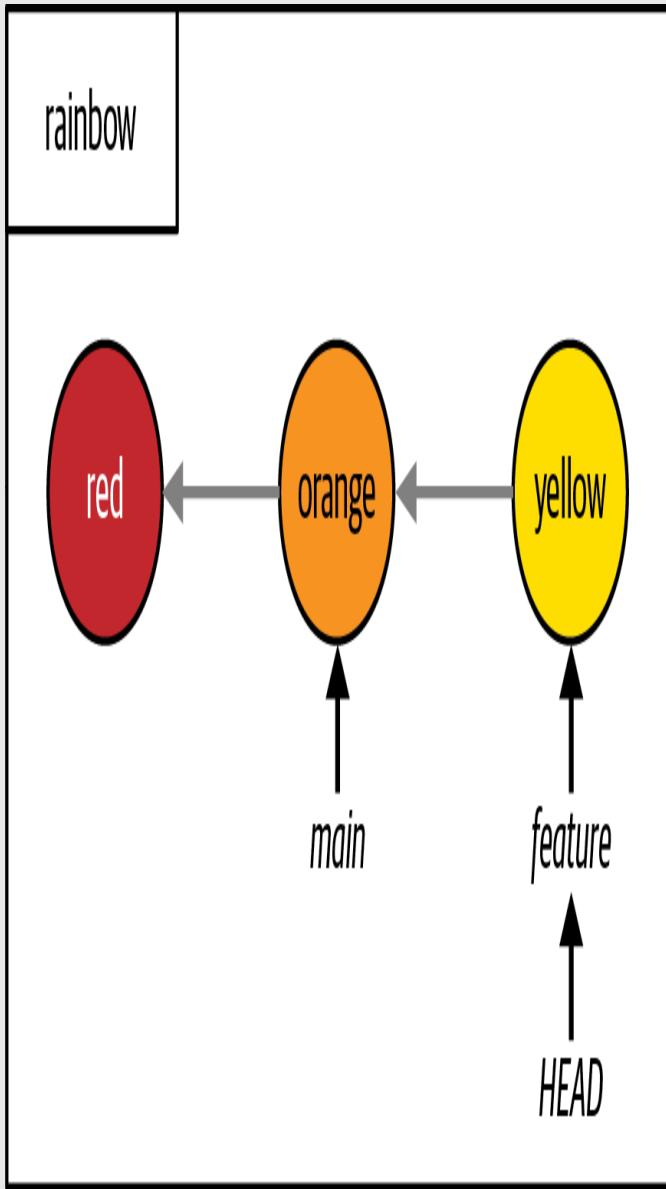
In this chapter, you are going to learn about integrating changes from one branch into another. In Git, there are two ways to do this: merging and rebasing. We will cover rebasing in [Chapter 11](#); for now we will focus on merging. This chapter will introduce the two types of merges (fast-forward merges and three-way merges), and you will carry out a fast-forward merge.

In the process, you will also learn about how Git protects you from losing any uncommitted changes, how changing branches may change the contents of the working directory, and how to check out commits directly.

State of the Local Repository

At the start of this chapter, you should have three commits and two branches in your `rainbow` repository, and you should be on the `feature` branch. The current state of the `rainbow` repository is illustrated in [Visualize it 5-1](#).

[VISUALIZE IT 5-1]



The `rainbow` repository at the start of [Chapter 5](#), with three commits and two branches

Introducing Merging

In [Chapter 4](#), you created your first branch, `feature`, and started working on that branch. Branches are a powerful feature of Git, and it's great that they

allow us to work on different parts of a project independently. But how can you then combine the work you have done with the `main` branch once you're ready?

Merging in Git is one way you can integrate the changes made in one branch into another branch. In any merge, there is one branch that you are merging, called the *source branch*, and one branch that you're merging into, called the *target branch*. The source branch is the branch that contains the changes that will be integrated into the target branch. The target branch is the branch that receives the changes and is therefore the only one that is altered in this operation. Let's take a look at how to use merging in

[Example Book Project 5-1.](#)

Example Book Project 5-1

Suppose that in my Book project, I decide together with my editor that every time I start to work on one of the chapters, I will make a secondary branch off the `main` branch. I also agree with my editor that I will merge the secondary branch into `main` only after they have reviewed my work on the branch.

For example, let's say that I decide to work on chapter 4. I can make a branch called `chapter_four` off the `main` branch, work on this branch, and then, when I think I've made significant headway, submit my work to my editor. Once they approve the changes, I can merge the `chapter_four` branch into the `main` branch in my local repository.

[Example Book Project 5-1](#) illustrates that when you use branches, it's also essential to learn how to merge your work back together. Now, let's explore the different types of merges that exist.

Types of Merges

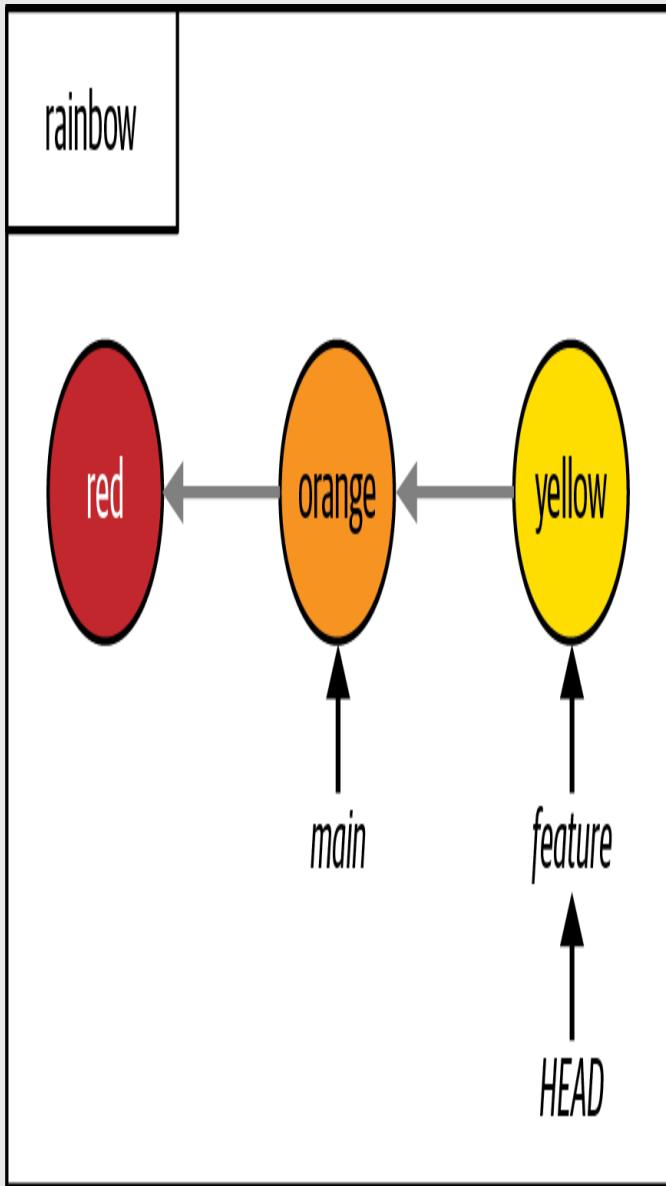
There are two types of merges:

- Fast-forward merges
- Three-way merges

The factor that determines which of these types of merges will take place when you merge the source branch into the target branch is whether the development histories of the two branches have diverged. A branch's development history can be traced by following the parent links of commits.

In [Chapter 4](#), I explained that in the Repository Diagrams a gray arrow pointing from one commit to another indicates a parent link (that is, the arrow points backward from the child commit to the parent commit). In [Visualize it 5-2](#), you can see that in the `rainbow` repository, the parent commit of the orange commit is the red commit. By the same logic, the orange commit is the parent commit of the yellow commit.

[VISUALIZE IT 5-2]



Parent links in the `rainbow` repository

The development history of a branch begins with the commit it points to, and extends backward through the chain of commits. In [Visualize it 5-2](#), you can see that the development history of the `main` branch is therefore

made up of the orange commit and the red commit, while the development history of the `feature` branch is made up of the yellow commit, the orange commit, and the red commit.

To illustrate the difference between fast-forward merges and three-way merges, let's start by going over an example of a fast-forward merge in [Example Book Project 5-2](#).

From [Example Book Project 5-2](#) we can see that a fast-forward merge is a type of merge that occurs when the development histories of the branches involved in the merge have not diverged—in other words, when it is possible to reach the target branch by following the parent links that make up the commit history of the source branch. During a fast-forward merge, Git takes the pointer of the target branch and moves it to the commit of the source branch.

Example Book Project 5-2

Suppose there are two commits on the `main` branch in my Book project repository, commits A and B, as seen in [Figure 5-1](#).

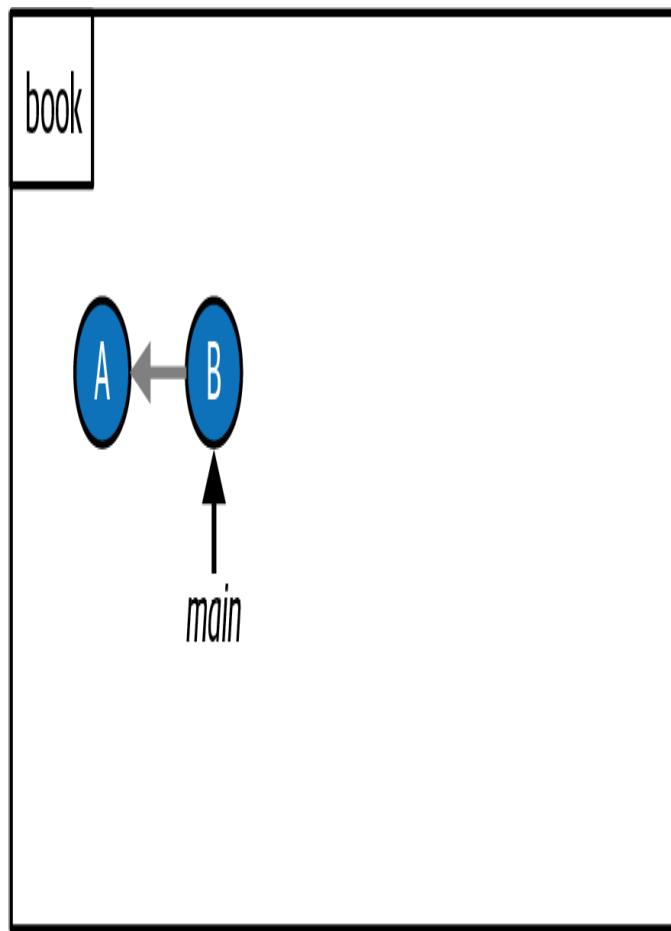


FIGURE 5-1

The commit history of the `book` repository with one branch, the `main` branch

Next, let's assume I create the `chapter_six` branch to work on chapter 6 of my book and I add commits C, D, and E to the `chapter_six` branch, as

seen in [Figure 5-2](#).

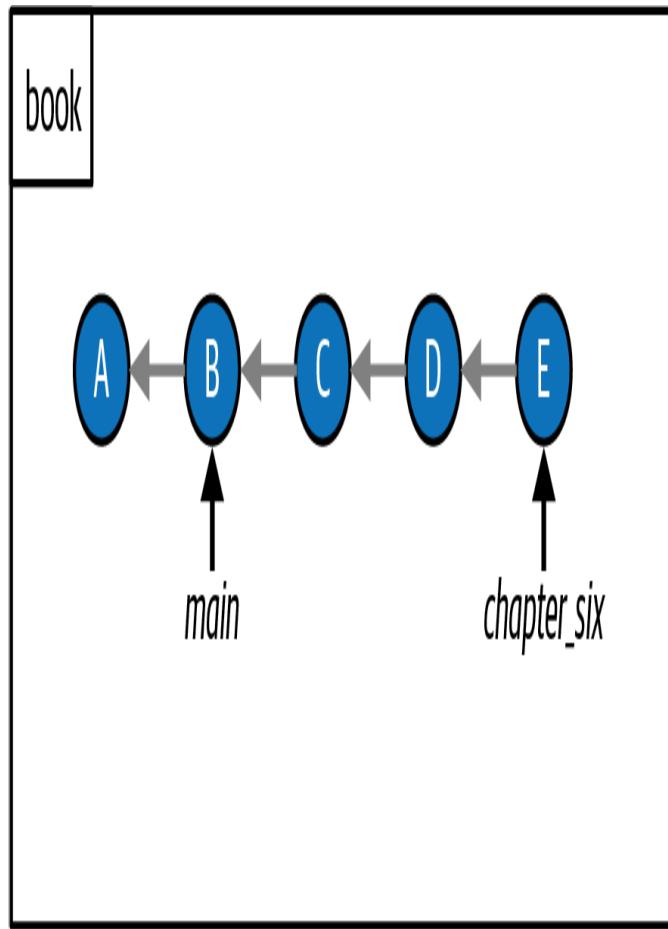


FIGURE 5-2

The commit history of the `book` repository after I work on the `chapter_six` branch

If I follow the parent links of the `main` branch backward, I can see that the branch is made up of commits A and B. In other words, the development history of the `main` branch consists of commits A and B. On the other hand, the development history of the `chapter_six` branch consists of commits A, B, C, D, and E.

If we can reach one branch through the commit history of another branch, we say that the development histories of the branches *have not diverged*. If I follow the parent links from the `chapter_six` branch, which points to commit E, backward, I reach the `main` branch, which points to commit B. Therefore, the `main` branch and the `chapter_six` branch have not diverged.

If I were to now merge the `chapter_six` branch into the `main` branch, a *fast-forward merge* would occur. During the fast-forward merge, the `main` branch pointer would move forward to point to the commit that the `chapter_six` branch points to, which is commit E, as seen in [Figure 5-3](#).

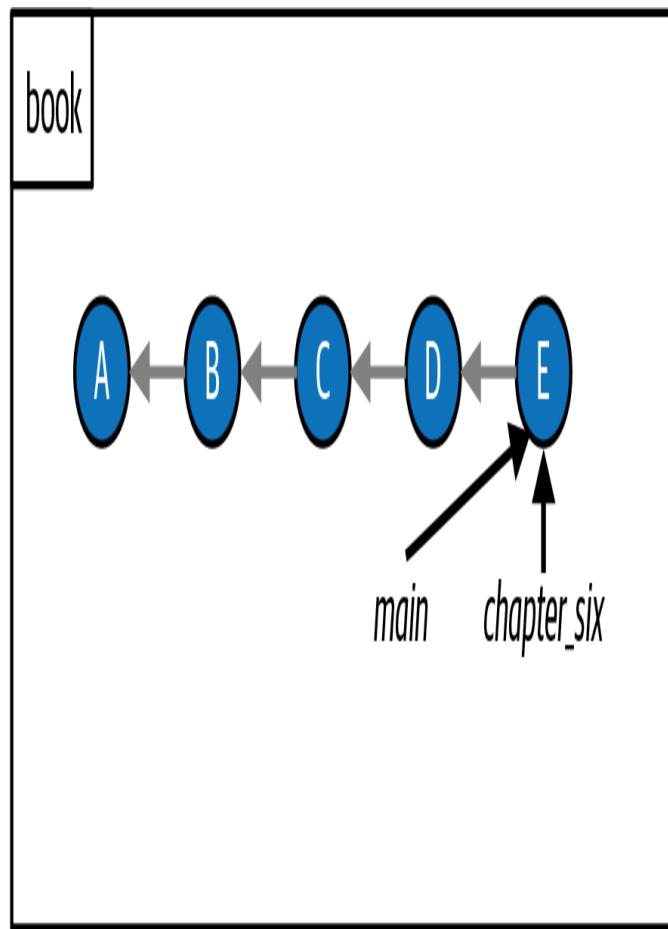


FIGURE 5-3

The commit history after I merge the `chapter_six` branch into the `main` branch in the `book` repository

In this merge example, `chapter_six` is the source branch and `main` is the target branch. [Figure 5-3](#) shows that the `main` branch pointer simply moved forward from commit B to commit E. This is why these kinds of merges are called “fast-forward” merges.

Next, let’s go over an example of a three-way merge in [Example Book Project 5-3](#).

From [Example Book Project 5-3](#) we can see that a three-way merge is a type of merge that occurs when the development histories of the branches involved in the merge have diverged. Development histories *have* diverged when it is *not* possible to reach the target branch by following the commit history of the source branch. In this case when you merge the source branch into the target branch, Git performs a three-way merge, creating a merge commit to tie the two development histories together; it then moves the pointer of the target branch to the merge commit.

Example Book Project 5-3

Suppose that the last two commits on the `main` branch in my `book` repository are commits F and G, as seen in [Figure 5-4](#).

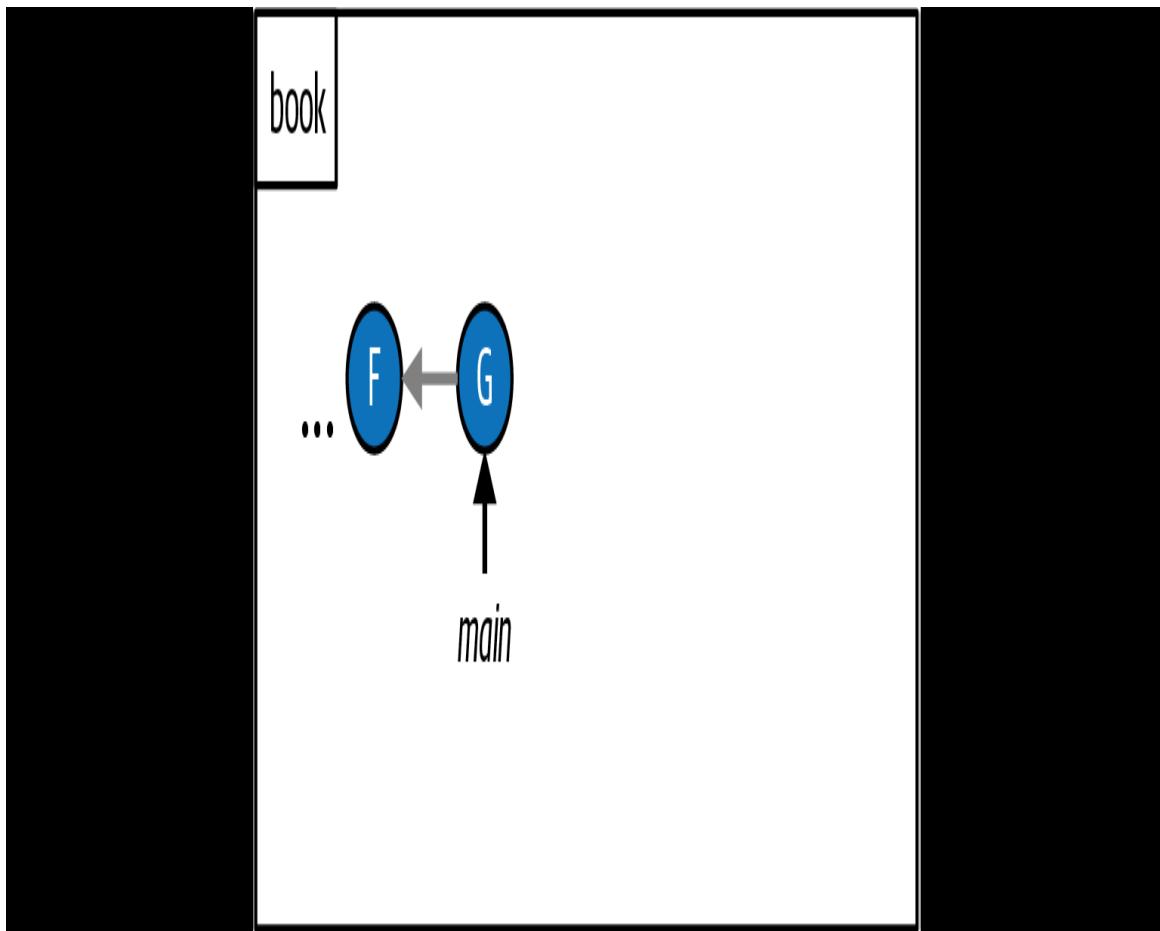


FIGURE 5-4

The commit history of the `book` repository with the last two commits on the `main` branch

Now suppose I decide to make a `chapter_eight` branch to work on chapter 8 of my book, and I make commits H, I, and J. At the same time,

however, I add some work to the `main` branch, and it now points to commit L. This is illustrated in [Figure 5-5](#).

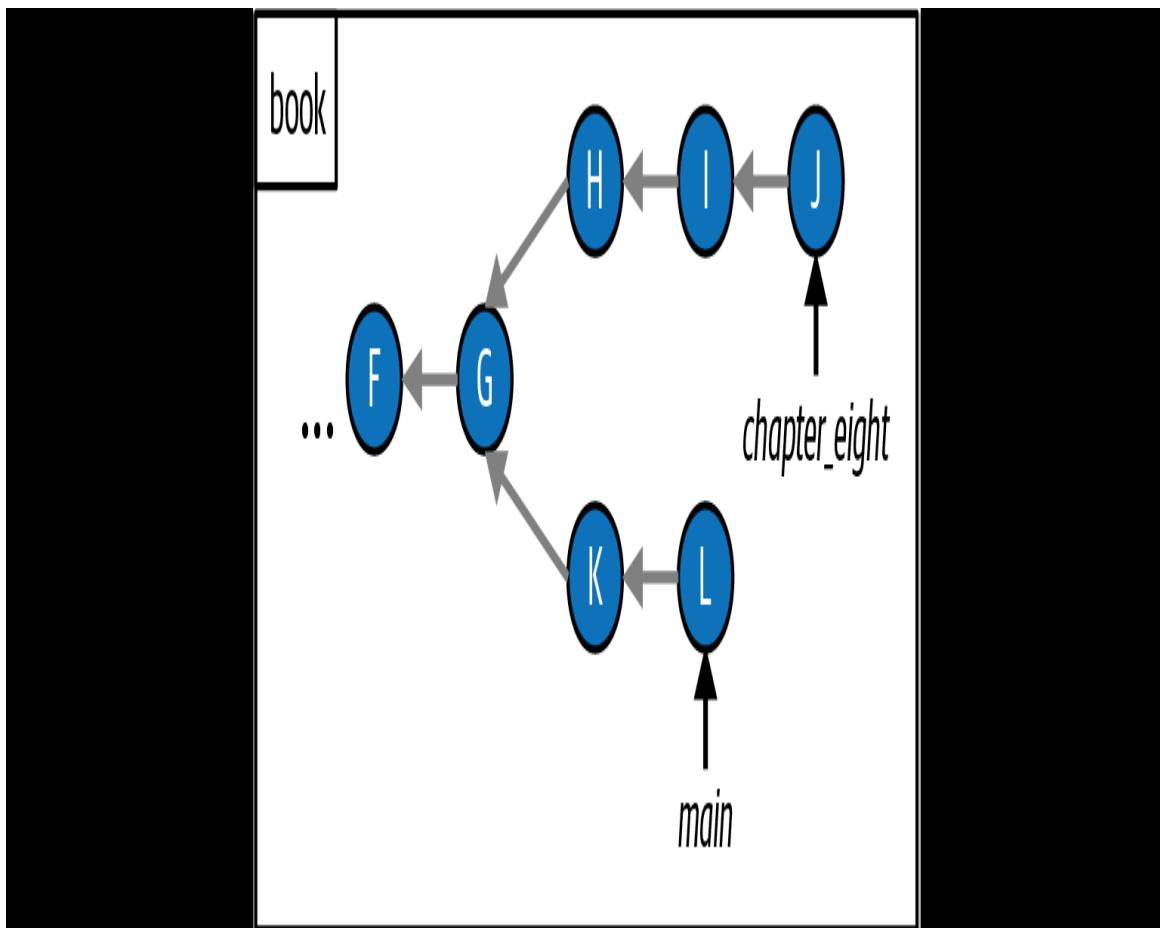


FIGURE 5-5

An example of a commit history where work has been added to the `main` branch and the `chapter_eight` branch

In [Figure 5-5](#), you can see that the development history of the `chapter_eight` branch is made up of commits F, G, H, I, and J. On the other hand, the commit history of the `main` branch is made up of commits F, G, K, and L. There is no way to follow the parent links (represented

by gray arrows) of the `chapter_eight` branch backward to reach the commit that the `main` branch points to, which is commit L. In Git, to describe this situation, we say that the development histories of the branches *have diverged*.

If I merge the `chapter_eight` branch into the `main` branch, it can't be a fast-forward merge because there is no way to just move the branch pointer forward to combine these two development histories. Instead, a *merge commit* (represented by commit M) will be created to tie the two development histories together, as shown in [Figure 5-6](#). A merge commit is a commit that has more than one parent. This is an example of a *three-way merge*.

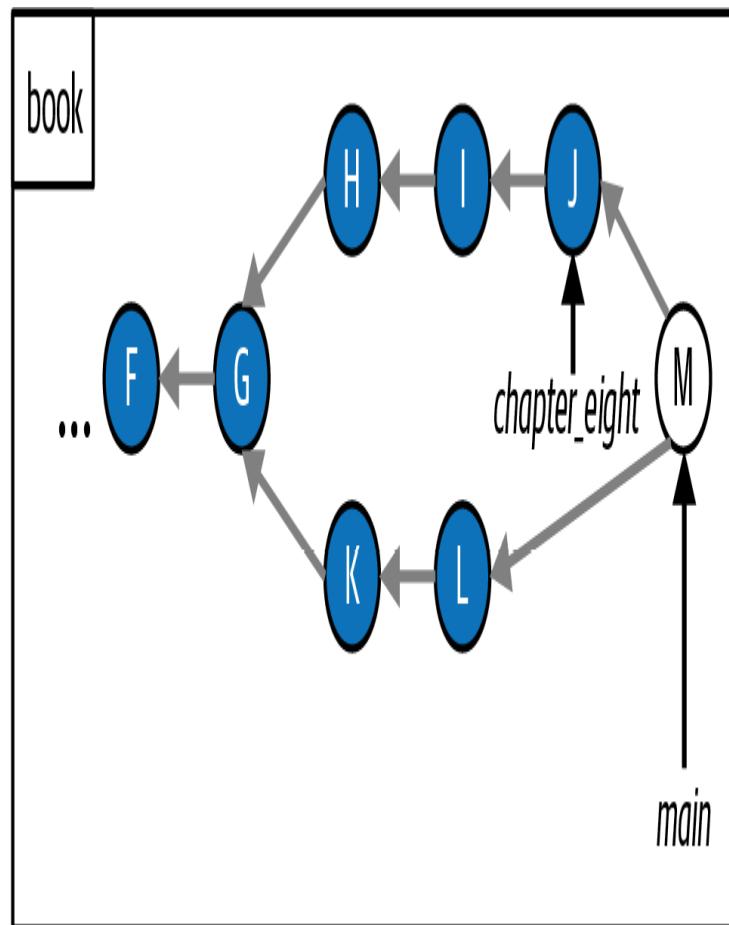


FIGURE 5-6

The commit history after I merge the `chapter_eight` branch into the `main` branch

In [Figure 5-6](#), commit M points back to both commit J and commit L. The reason that this kind of merge is called a three-way merge is because in order to carry out the merge, Git will take a look at the two commits that the branches involved in the merge are pointing to—in the case of the `book` repository, commit J and commit L—as well as the commit that is the common ancestor of these two commits, which in this case is commit G. Hence the name, “three-way” merge.

Three-way merges are a more complex type of merge where you may experience *merge conflicts*. These arise when you merge two branches where different changes have been made to the same parts of the same file(s), or if in one branch a file was deleted that was edited in the other branch.

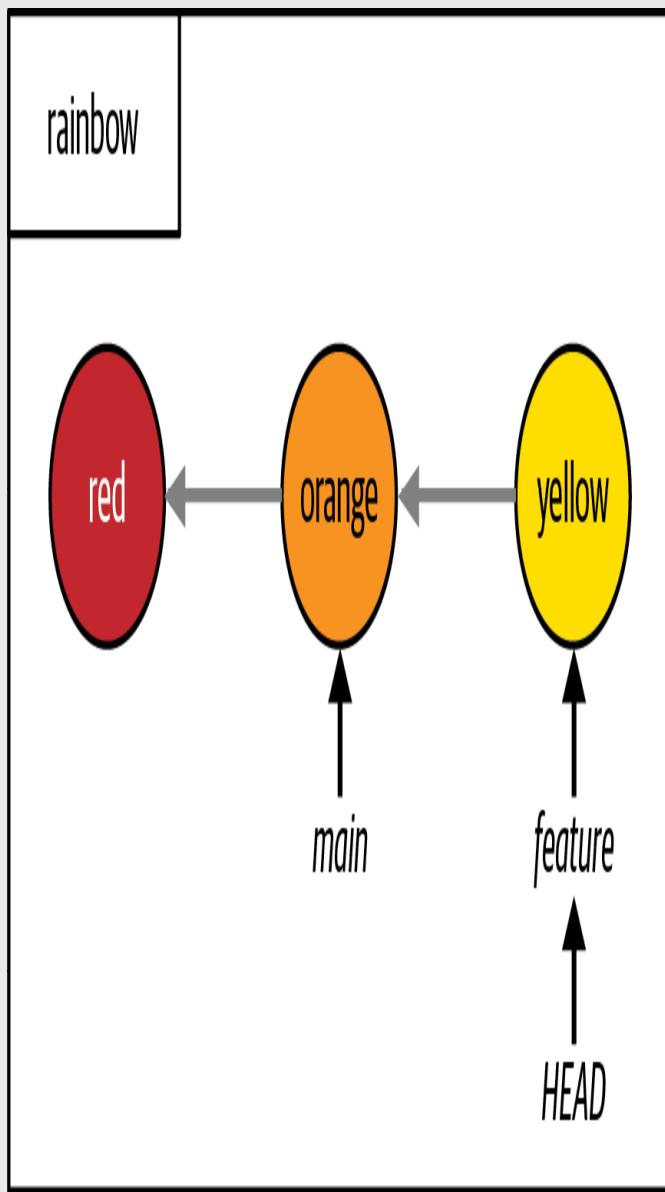
[Chapter 9](#) of this book is dedicated to explaining three-way merges in more depth, and in [Chapter 10](#) you will learn more about merge conflicts. In each chapter, you will go through a hands-on example. In this chapter, you will practice carrying out a fast-forward merge in the Rainbow project.

Doing a Fast-Forward Merge

To practice merging in the `rainbow` repository, you will merge the `feature` branch into the `main` branch. The `feature` branch is the source branch and the `main` branch is the target branch. In [Visualize it 5-3](#), you can see that you can

reach the `main` branch by following the commit history of the `feature` branch; therefore, this will be a fast-forward merge.

[VISUALIZE IT 5-3]



In the `rainbow` repository, the development histories of the `main` branch and the `feature` branch have not diverged

There are two steps involved in doing a merge:

1. Switch onto the branch that you want to merge into (the target branch).
2. Use the `git merge` command and pass in the name of the branch you're merging (the source branch).

[SAVE THE COMMAND]

`git merge <branch_name>`

Integrate changes from one branch into another branch

This section will walk you through performing your first merge. In the process, you will learn two more important lessons about Git: first, that Git protects you from losing any work you have done in files that you have not committed, and second, that changing branches may cause the contents of the working directory to change.

SWITCHING ONTO THE BRANCH YOU ARE MERGING INTO

The first step of doing a merge is to switch onto the target branch. You are going to merge `feature` into `main`, so you need to switch onto the `main` branch.

Recall the three things that happen when you use the `git switch` (or `git checkout`) command to switch onto a branch:

1. It changes the `HEAD` pointer to point to the branch you are switching onto.
2. It populates the staging area with all the files and directories that are part of the commit you are switching onto.
3. It copies the contents of the staging area into the working directory.

As these three steps indicate, if the branches point to different commits, changing branches also changes the contents of your working directory. You'll see this in practice momentarily, but first let's look at the other important feature I mentioned.

Git protects you from losing uncommitted changes

I just mentioned that switching branches changes the contents of your working directory. But what if you have modified files in your working directory (in other words, files that you have edited) that you have not yet committed? Will you lose all the work you've done in those files if you switch branches? Thankfully not! Git protects you from losing uncommitted changes.

If Git detects that switching branches will cause you to lose uncommitted changes in your working directory, then it will stop you from switching branches and present you with an error message. However, this happens only if the files that contain uncommitted changes have conflicting changes in the branch you are switching onto.

To illustrate why this is important, let's look at [Example Book Project 5-4](#).

Example Book Project 5-4

Suppose I want to work on two different approaches for chapter 5 in my `book` repository. This means I'll be working on the `chapter_five.txt` file. I make two branches off the `main` branch, once called `chapter_five_approach_a` and one called `chapter_five_approach_b`.

First I go onto the `chapter_five_approach_a` branch and work on approach A. I make a couple commits. In these commits I'm editing the `chapter_five.txt` file.

Next, I decide to switch onto the `chapter_five_approach_b` branch to work on approach B in my text editor. While on the `chapter_five_approach_b` branch, I edit the `chapter_five.txt` file extensively, but I forget to actually make any commits on the branch (in other words, to properly save my work).

At some point, I decide I want to switch back to the `chapter_five_approach_a` branch to check something I had done on that branch. If Git simply allowed me to switch branches, then the version of the `chapter_five.txt` file in my working directory that I was working on in the `chapter_five_approach_b` branch would be replaced by the version of the file in the `chapter_five_approach_a` branch, and I would lose *all* my changes that I had worked on in approach B because I forgot to commit them.

Luckily, Git won't let this happen. Instead, it will warn me that I have uncommitted changes and it will remind me to make a commit so I won't lose them.

For a hands-on example in the Rainbow project of editing a file and witnessing how Git protects you from losing uncommitted changes, go on to [Follow Along 5-1](#).

[FOLLOW ALONG 5-1]

1 rainbow \$ **git status**

On branch feature

nothing to commit, working tree clean

2 In the `rainbow` project directory in your text editor, in the `rainbowcolors.txt` file, add “Green is the fourth color of the rainbow.” on line 4. Then save the file.

3 rainbow \$ **git status**

On branch feature

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

modified: `rainbowcolors.txt`

no changes added to commit (use "git add" and/or "git commit -a")

4 rainbow \$ **git switch main**

error: Your local changes to the following files would be overwritten by checkout:

`rainbowcolors.txt`

Please commit your changes or stash them before you switch branches.

Aborting

What to notice:

- In step 1, the `git status` output shows you that there are no modified files in the working directory.
- In step 3, the `git status` output indicates that the `rainbowcolors.txt` file is a modified file in the working directory.
- In step 4, you witness how Git *does not* allow you to switch branches. It warns you that `Your local changes to the following files would be overwritten by checkout` and it tells you to `Please commit your changes or stash them before you switch branches`.

If Git had allowed you to switch branches, then the version of the `rainbowcolors.txt` file in the working directory that mentions the colors red, orange, yellow, and green would have been replaced by the version of the `rainbowcolors.txt` file in the orange commit that the `main` branch points to, which only mentions the colors red and orange. This means that you would have lost *all* the work you did in adding the color green to your list of colors.

Instead, Git protects you from losing your work in the `rainbowcolors.txt` file by warning you to commit your changes.

[NOTE]

Git will not prevent you from switching branches if you make changes to files without saving them in the text editor, because those files will be considered unmodified files. So, always remember to save files in your text editor when you're done working on them!

Now, you'll continue with carrying out the first merge in the Rainbow project. In this case, let's assume you aren't yet ready to add notes about the color green. Go ahead to [Follow Along 5-2](#) and remove the sentence about that color, so that there are no modified files in your working directory.

[FOLLOW ALONG 5-2]

- 1 In the `rainbow` project directory in your text editor, remove the line "Green is the fourth color of the rainbow." from the `rainbowcolors.txt` file and save the file.

Make sure to remove any extra lines or spaces that you might have added.

- 2 `rainbow $ git status`

on branch feature

nothing to commit, working tree clean

What to notice:

- The `git status` command indicates that your working directory no longer has any modified files.

Next, you're going to carry out the first step of a merge, which is to switch onto the branch you are merging into. While you do this, you will witness how switching branches changes the files in your working directory.

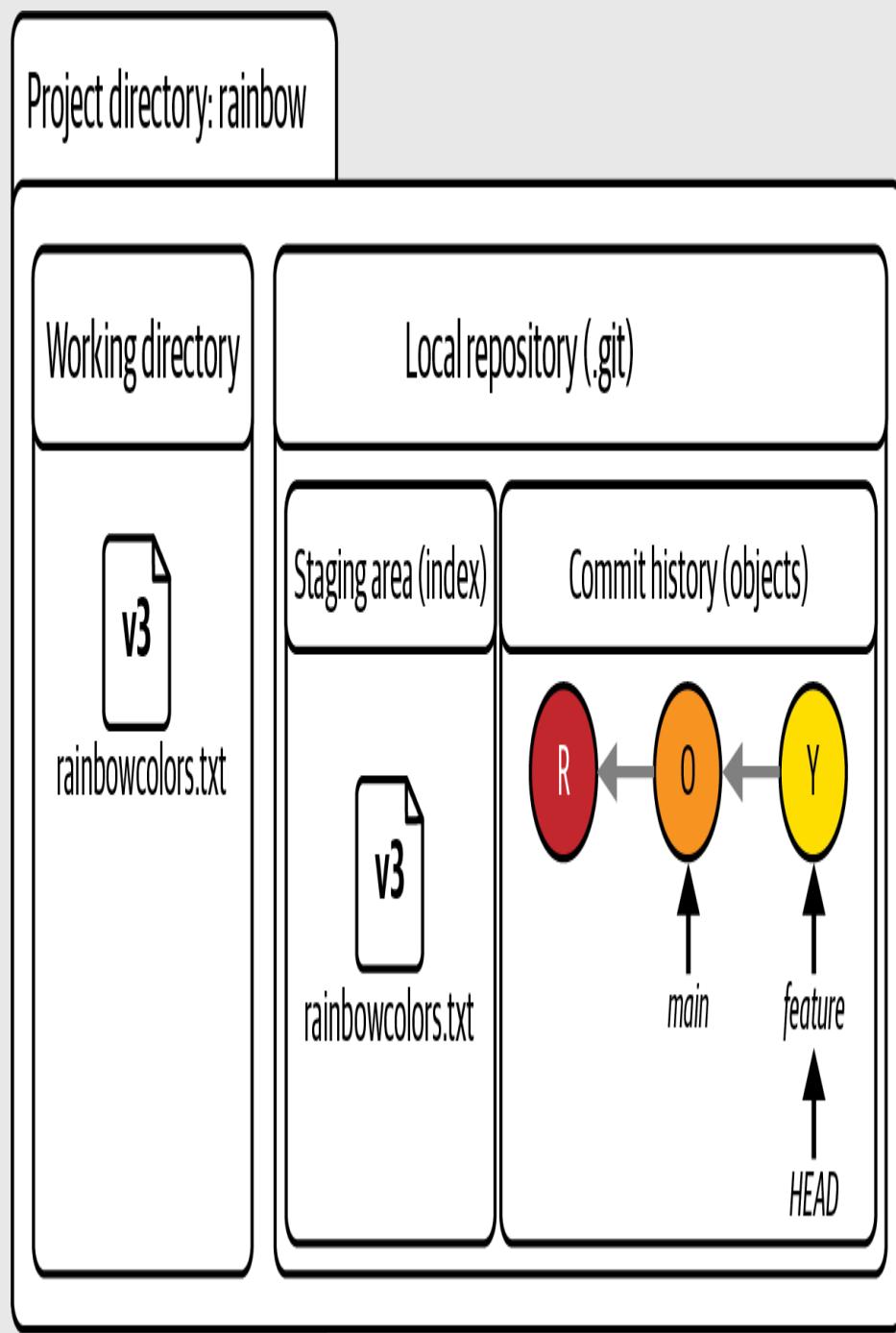
Switching branches changes files in the working directory

To see how switching branches changes the files in the working directory, take a look at [Visualize it 5-4](#), which uses the Git Diagram introduced in [Chapter 2](#) to show what the different areas of the `rainbow` project directory look like at the moment.

[NOTE]

In Visualize It diagrams where there isn't enough space to include the full name of a commit, I'll use an abbreviated form of the name instead: for example, R for red, O for orange, and so on. For a full list of the abbreviations used for each commit, see Table P-1 in the Preface.

[VISUALIZE IT 5-4]



The `rainbow` project directory before you switch from the `feature` branch onto the `main` branch

What to notice:

- The version of the `rainbowcolors.txt` file in the working directory and staging area is the one that mentions the colors red, orange, and yellow. It is represented as version 3 (v3) of the file.

Now, go to [Follow Along 5-3](#) to switch from the `feature` branch onto the `main` branch in the `rainbow` repository (which points to a different commit) and observe how the contents of your working directory change.

[FOLLOW ALONG 5-3]

1 Make sure you have the `rainbowcolors.txt` file open in your text editor window, and place it next to your command line window so that you have a view of both of them when you execute the upcoming commands. Look at the contents of the `rainbowcolors.txt` file.

2 `rainbow $ git switch main`
Switched to branch 'main'

3 Look at the contents of the `rainbowcolors.txt` file.

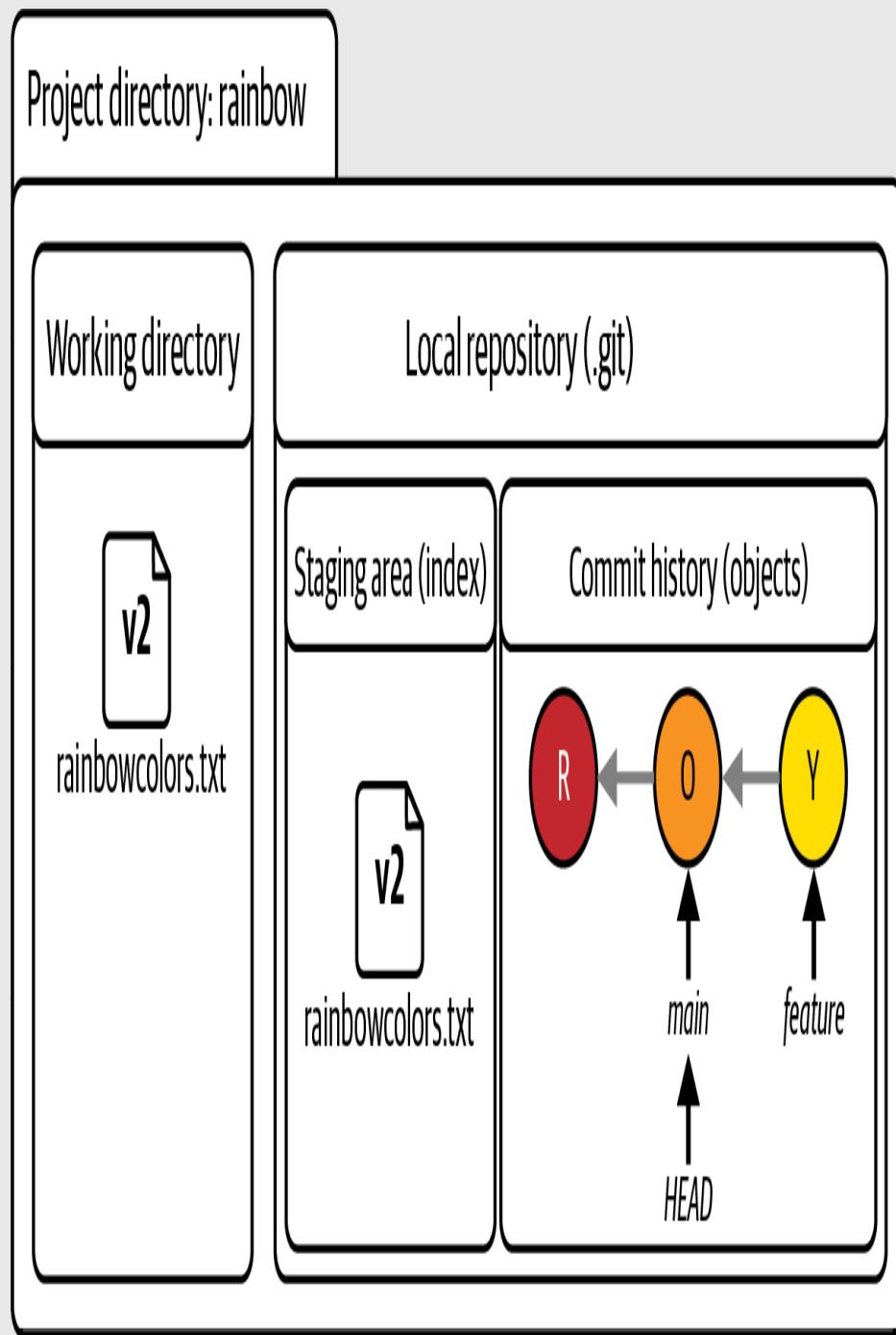
4 `rainbow $ git log`
`commit 7acb333f08e12020efb5c6b563b285040c9dba93 (HEAD -> main)`
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date: Sat Feb 19 09:42:07 2022 +0100
orange
`commit c26d0bc371c3634ab49543686b3c8f10e9da63c5`
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date: Sat Feb 19 09:23:18 2022 +0100
red

What to notice:

- The version of the `rainbowcolors.txt` file that was in your text editor before you switched onto the `main` branch mentioned the colors red, orange, and yellow. The version of the `rainbowcolors.txt` file that is in your text editor after you switch onto the `main` branch mentions only the colors red and orange. This is illustrated in Visualize It 5-5.

- In step 4, the `git log` output shows only the red and orange commits. It no longer shows the yellow commit.

[VISUALIZE IT 5-5]



The `rainbow` project directory after you switch from the `feature` branch onto the `main` branch

What to notice:

- You are on the `main` branch and it points to the orange commit, which includes the version of the `rainbowcolors.txt` file that mentions the colors red and orange. This is represented as version 2 (v2).
- v3 of the `rainbowcolors.txt` file has been replaced by v2 of the file in both the staging area and the working directory.

You have just explicitly observed that switching branches changes the contents of the working directory. Before you go on to the second step of doing the merge, let's briefly touch upon why the `git log` output in step 4 of [Follow Along 5-3](#) shows only the red and orange commits.

Viewing a list of all commits

In [“Viewing a List of Commits” on page 40](#), I mentioned that the `git log` command shows a list of commits in reverse chronological order. However, in reality, it shows only a list of commits that are reachable by following the parent links from the commit you are on when you execute the command. To see a list of commits for *all* the branches in your local repository, you must use the `git log` command with the `--all` option.

[SAVE THE COMMAND]

`git log --all`

Show a list of commits in reverse chronological order for all branches in a local repository

Go to [Follow Along 5-4](#) to see a list of the commits for all the branches in your local repository.

[FOLLOW ALONG 5-4]

```
1 rainbow $ git log --all
commit fc8139cbf8442cddb5e469285abaac6de919ace6 (feature)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 10:09:59 2022 +0100

    yellow

commit 7acb333f08e12020efb5c6b563b285040c9dba93 (HEAD -> main)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 09:42:07 2022 +0100

    orange

commit c26d0bc371c3634ab49543686b3c8f10e9da63c5
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 09:23:18 2022 +0100

    red
```

What to notice:

- The `git log --all` output shows the red, orange, and yellow commits.

Now that you are on the `main` branch, you can move on to the second step of doing a merge.

USING THE GIT MERGE COMMAND TO EXECUTE A MERGE

Go to [Follow Along 5-5](#) to execute the merge by passing the name of the branch you are merging—the source branch, which is `feature`—to the `git merge` command.

[FOLLOW ALONG 5-5]

1 Make sure your text editor window with a view of the `rainbowcolors.txt` file is open next to your command line window so that you can see both of them when you execute the upcoming commands. Look at the contents of the `rainbowcolors.txt` file.

2 `rainbow $ git merge feature`

```
Updating 7acb333..fc8139c
Fast-forward
 rainbowcolors.txt | 3 +-+
 1 file changed, 2 insertions(+), 1 deletion(-)
```

3 Look at the contents of the `rainbowcolors.txt` file.

4 `rainbow $ git log`

```
commit fc8139cbf8442cdbb5e469285abaac6de919ace6 (HEAD -> main, feature)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 10:09:59 2022 +0100

    yellow

commit 7acb333f08e12020efb5c6b563b285040c9dba93
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 09:42:07 2022 +0100

    orange

commit c26d0bc371c3634ab49543686b3c8f10e9da63c5
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 09:23:18 2022 +0100

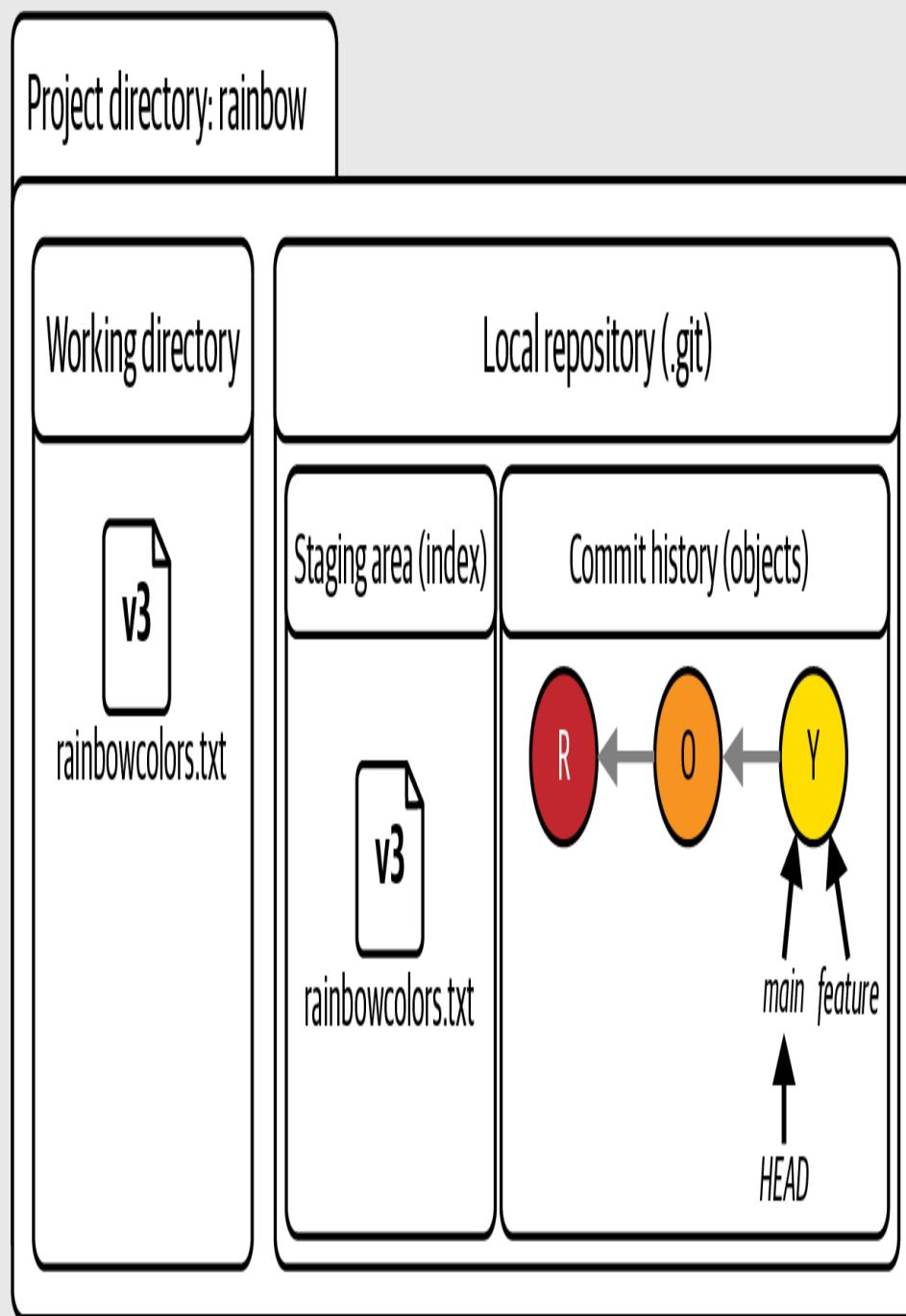
    red
```

What to notice:

- The `git merge` output mentions `Updating 7acb333..fc8139c` and `Fast-forward`. This tells you that Git updated the commit the `main` branch points to, and it tells you that this was a fast-forward merge. Note that the commit hashes in your output will be different from the ones in this book.
- The `git log` output shows that `main` points to the yellow commit. In your text editor, you can see that your working directory contains the version of the `rainbowcolors.txt` file that is part of the yellow commit.
- You merged the `feature` branch into the `main` branch, but it still exists; it did not get automatically deleted.

All of these observations are illustrated in [Visualize it 5-6](#).

[VISUALIZE IT 5-6]



The working directory and the local repository after merging `feature` into `main`

You merged `feature` into `main`, but in Git, merging a branch does not delete the branch. You must explicitly delete a branch if you no longer want to use it. For now, you will keep the `feature` branch, and we will go over how to delete branches in [Chapter 8](#). The next topic I want to cover here is how to check out commits.

Checking Out Commits

In [Chapter 4](#), I mentioned that the `git checkout` command may be used to switch branches as well as to carry out other actions. One of the other things you can do with the `git checkout` command is check out commits.

At the moment, you are on the `main` branch, which points to the yellow commit. But what if you want to look at an older version of your project? For example, what if you want to see the state of your project at the orange commit?

There is currently no branch pointing to the orange commit, so you can't switch onto it by switching onto a branch. Instead, you can choose to *check out* that commit by using the `git checkout` command and passing in the commit hash of the orange commit.

[SAVE THE COMMAND]

```
git checkout <commit_hash>
```

Check out a commit

When you do this, the `git checkout` command will carry out three actions that are similar to the ones described in [Chapter 4](#) and earlier in this chapter:

1. It changes the `HEAD` pointer to point to the *commit* you are switching onto.
2. It populates the staging area with all the files and directories that are part of the commit you are switching onto.
3. It copies the contents of the staging area into the working directory.

The main difference between these steps and the steps mentioned previously is that in step 1 the `HEAD` pointer will point directly to a commit instead of pointing to a branch. This means that you will be in something that Git (scarily) calls *detached HEAD state*. This allows you to look at any commit—or, in other words, any version of your project—in your entire repository.

As these steps indicate, checking out commits changes the contents of the working directory in the same way that switching branches does.

[NOTE]

It is not recommended to make any changes to a repository while in detached `HEAD` state (that is, while not on a branch). You will usually want to make commits on branches because branches are easier to remember and refer to than commit hashes, and because Git was designed to be used with branches. Therefore, if you check out a commit, it is common to make and switch onto a new branch that will point to that commit, so that you're no longer in detached `HEAD` state.

Go to [Follow Along 5-6](#) to check out the orange commit and observe what it's like to be in detached `HEAD` state.

[FOLLOW ALONG 5-6]

1 Retrieve the commit hash for your orange commit (you can copy this from your `git log` output). Make sure to use your commit hash in step 2 of this Follow Along.

2 `rainbow $ git checkout 7acb333f08e12020efb5c6b563b285040c9dba93`

Note: switching to '7acb333f08e12020efb5c6b563b285040c9dba93'.

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by switching back to a branch. If you want to create a new branch to retain commits you create, you may do so (now or later) by using -c with the switch command. Example:

`git switch -c <new-branch-name>`

Or undo this operation with:

`git switch -`

Turn off this advice by setting config variable `advice.detachedHead` to `false`

`HEAD is now at 7acb333 orange`

[FOLLOW ALONG 5-6]

3

```
rainbow $ git log --all
commit fc8139cbf8442cdbb5e469285abaac6de919ace6 (main, feature)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 10:09:59 2022 +0100

    yellow

commit 7acb333f08e12020efb5c6b563b285040c9dba93 (HEAD)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 09:42:07 2022 +0100

    orange

commit c26d0bc371c3634ab49543686b3c8f10e9da63c5
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 09:23:18 2022 +0100

    red
```

4

Look at the contents of the `rainbowcolors.txt` file.

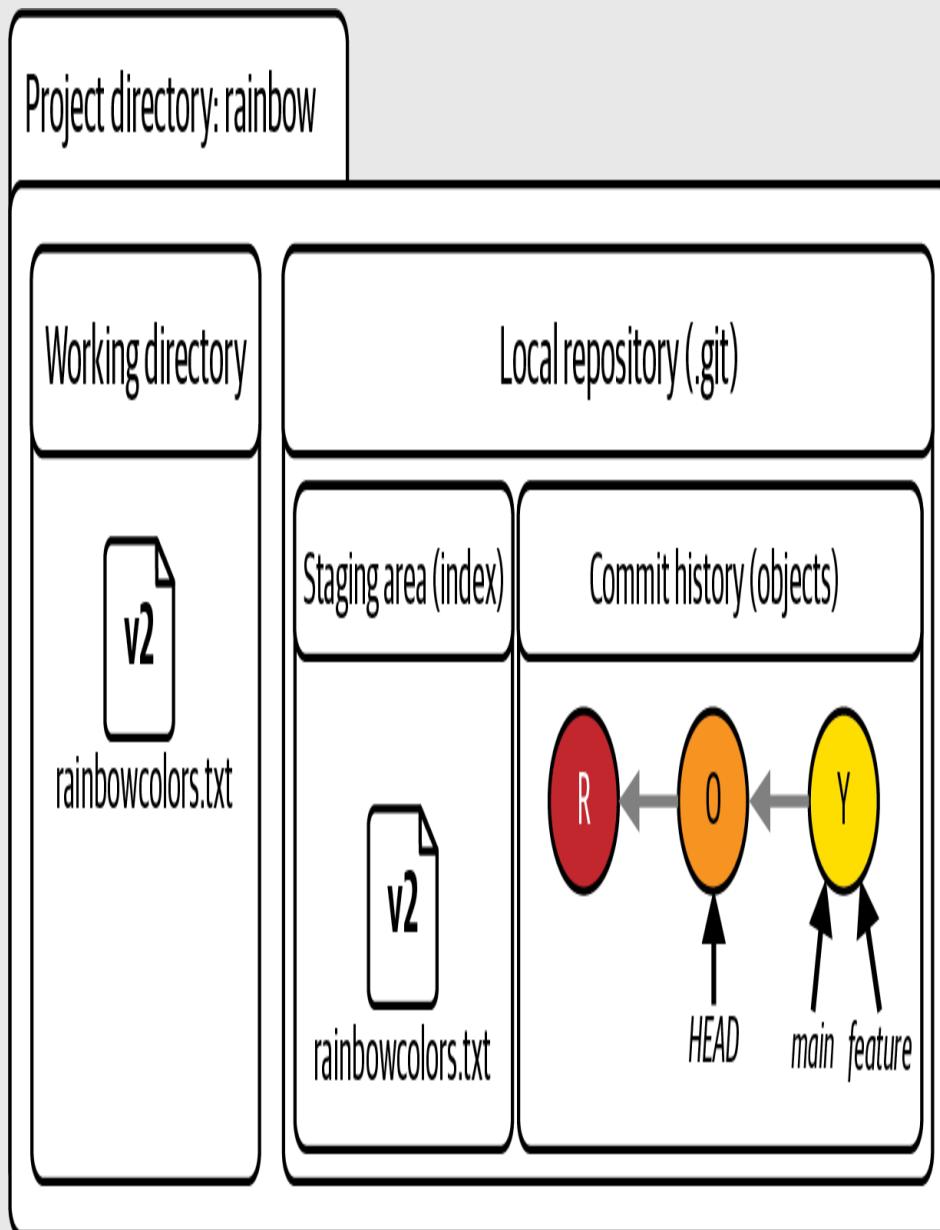
What to notice:

- In step 2, the `git checkout` output tells you that `You are in 'detached HEAD' state`. The output also refers to a modification of the `git switch` command, which we will touch upon in the following section.
- In step 3, the `git log` output shows you that `HEAD` now points to the orange commit.
- The version of the `rainbowcolors.txt` file in your working directory is the version that is part of the orange commit, which mentions only the

colors red and orange (represented by v2).

These observations are illustrated in [Visualize it 5-7](#).

[VISUALIZE IT 5-7]



The working directory and the local repository after you check out the orange commit and enter detached `HEAD` state

[NOTE]

It may seem scary that the contents of the working directory change every time you switch branches or check out a commit directly—but remember that you have not lost anything. All of your commits are safely stored in the commit history, and you can always choose to switch to another branch or to check out another commit to see what you were looking at before.

You just observed what it is like to check out a commit directly instead of checking out a branch. Next, go to [Follow Along 5-7](#) to switch back onto the `main` branch and exit detached `HEAD` state.

[FOLLOW ALONG 5-7]

1 rainbow \$ **git switch main**

Previous HEAD position was 7acb333 orange
Switched to branch 'main'

2 rainbow \$ **git log**

commit fc8139cbf8442cdbb5e469285abaac6de919ace6 (**HEAD -> main, feature**)

Author: annaskoulikari <gitlearningjourney@gmail.com>

Date: Sat Feb 19 10:09:59 2022 +0100

yellow

commit 7acb333f08e12020efb5c6b563b285040c9dba93

Author: annaskoulikari <gitlearningjourney@gmail.com>

Date: Sat Feb 19 09:42:07 2022 +0100

orange

commit c26d0bc371c3634ab49543686b3c8f10e9da63c5

Author: annaskoulikari <gitlearningjourney@gmail.com>

Date: Sat Feb 19 09:23:18 2022 +0100

red

3

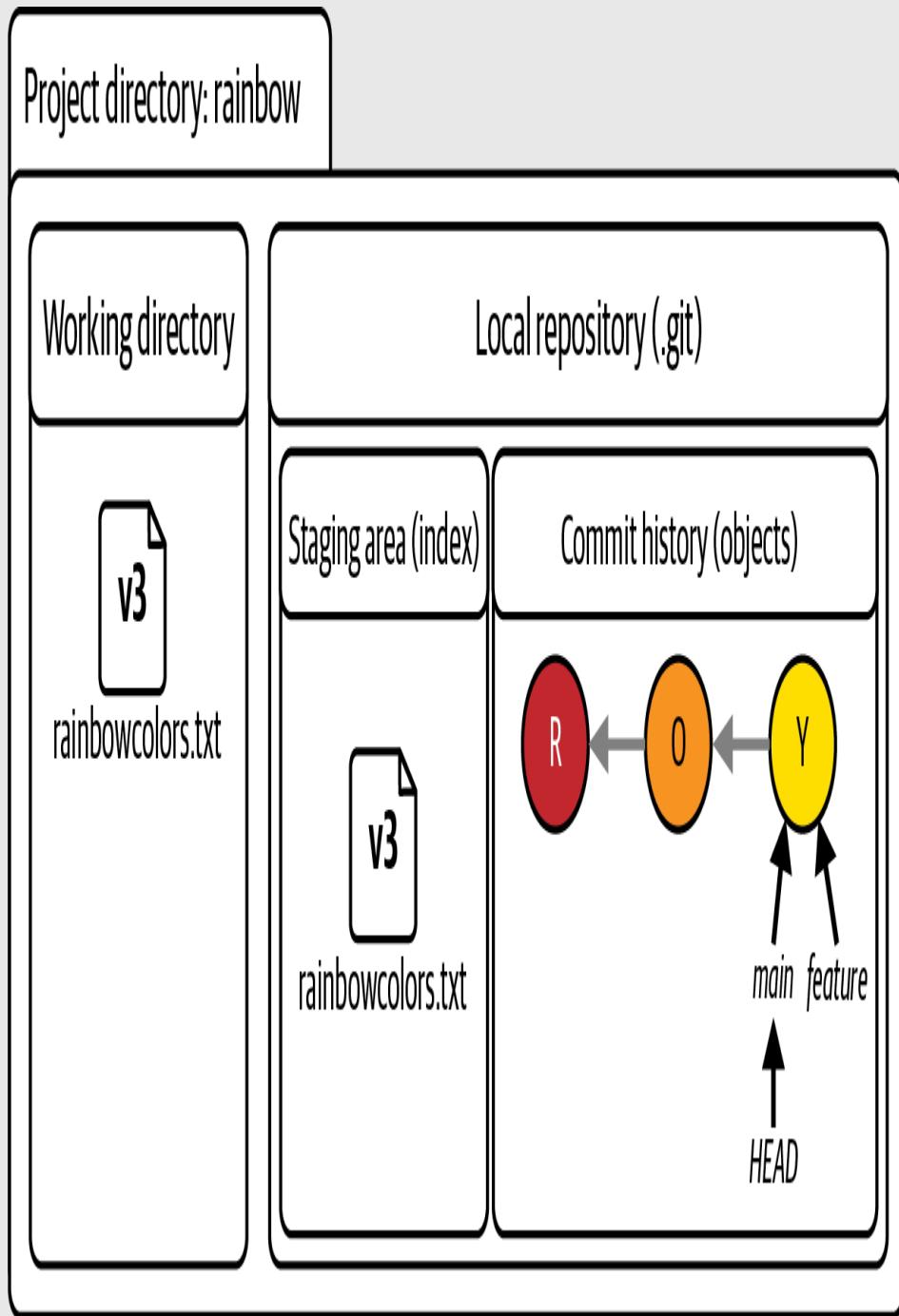
Look at the contents of the `rainbowcolors.txt` file.

What to notice:

- The version of the `rainbowcolors.txt` file in your working directory is the version that is part of the yellow commit, which mentions the colors red, orange, and yellow (represented by v3).
- You are back on the `main` branch.

This is illustrated in [Visualize it 5-8.](#)

[VISUALIZE IT 5-8]



The working directory and the local repository after you switch onto the `main` branch and exit detached `HEAD` state

Creating a Branch and Switching onto It in One Go

In [Chapter 4](#), you learned to make a new branch by using the `git branch` command and to use the `git switch` (or `git checkout`) command to change onto that branch.

In step 2 of [Follow Along 5-6](#), the `git checkout` output stated: *If you want to create a new branch to retain commits you create, you may do so (now or later) by using -c with the switch command. Example: git switch -c <new-branch-name>.*

This is because it is actually possible to use the `git switch` or `git checkout` command to create a branch and switch onto it in one go. If you use the `git switch` command you must use the `-c` option (which stands for “create”), and if you use the `git checkout` command you must use the `-b` option.

[SAVE THE COMMAND]

`git switch -c <new_branch_name>`

Create a new branch and switch onto it

`git checkout -b <new_branch_name>`

Create a new branch and switch onto it

You will practice using this command in [“Preparing to Make a Pull Request” on page 240](#), when you make a new branch.

Summary

In this chapter, we covered what merging is and why we do it. I introduced the two types of merges—fast-forward merges and three-way merges—and explained how the type of merge that will be carried out depends on the development histories of the branches involved in the merge.

Finally, you performed a fast-forward merge yourself in the [rainbow](#) repository, learning in the process that Git protects you from losing uncommitted changes, that changing branches may change the contents of the working directory, and how to check out commits.

This chapter marks the end of the first part of this book, in which you worked exclusively with local repositories. Next, in [Chapter 6](#), you are going to prepare your hosting service account and authentication details in order to start working with remote repositories.

Hosting Services and Authentication

In the last chapter, you learned about merging and how this feature of Git allows you to integrate changes from one branch into another.

Up until this point we have discussed working only in local repositories, and you have worked only on the `rainbow` repository, which is a local repository. This chapter marks the start of the second part of this book, in which you will work with hosting services and remote repositories.

In this chapter, you will choose a hosting service and prepare the authentication details you will use to connect to remote repositories on that hosting service by using either the HTTPS (Hypertext Transfer Protocol Secure) or SSH (Secure Shell) protocol. The information that you will need to carry out these tasks is contained in this chapter as well as in the resources in the Learning Git repository (<https://github.com/gitlearningjourney/learning-git>).

[NOTE]

If you work at a company that uses Git and a hosting service, you may already have a hosting service account using your company email address. However, I recommend using a personal account with a hosting service for the exercises in this book instead of a company account. This is because your company may have configured additional settings on the company hosting service account that could complicate exercises later in the book.

If you already work with a hosting service that you want to use to complete the exercises in this book and you already have authentication details set up to connect to remote repositories over a secure protocol, then you may skip this chapter and move on to [Chapter 7](#) to learn more about how to create and push to remote repositories.

If you haven't yet chosen which hosting service you want to use, have not set up authentication details for a specific protocol, or want to learn more about the available protocols, then keep reading.

Hosting Services and Remote Repositories

In [Chapter 2](#) I mentioned two types of repositories: local repositories and remote repositories. Local repositories are found on a computer, while remote repositories are hosted on a hosting service in the cloud.

I also mentioned that hosting services are companies that provide hosting for projects using Git. In this book, I will provide information about the three main Git hosting services: GitHub, GitLab, and Bitbucket.

To transfer data between a local repository and a remote repository on a hosting service, you must connect and authenticate using either SSH or

HTTPS. From [Chapter 7](#) onward, you will be uploading and downloading data to and from a remote repository using commands such as `git push`, `git clone`, `git fetch`, and `git pull`. To use these commands and connect to the remote repository you will have to have prepared authentication details for the protocol you want to use beforehand.

In this chapter, first you will choose a hosting service, and then you will set up authentication details to connect to remote repositories over HTTPS or SSH.

Setting Up a Hosting Service Account

As mentioned previously, if you already work with a hosting service and know you want to use that one for the exercises in this book, you can simply log in to your account on the hosting service website.

If you've never worked with a hosting service, you will have to choose which one you want to use and create an account. The three main hosting services are GitHub, GitLab, and Bitbucket. If you're not sure which one to use, then I recommend using GitHub since it is the most popular one and the one I use for the example output for the Rainbow project in this book.

Go to [Follow Along 6-1](#) to make your choice of hosting service and log in.

[FOLLOW ALONG 6-1]

1

Choose the hosting service on which you want to create a remote repository, and go to the website. Either log in or create an account. In these examples, we will use GitHub.

Now that you have chosen and logged in to your hosting service, you must choose which protocol you want to use to connect to remote repositories and prepare your authentication details for that protocol.

Setting Up Authentication Credentials

When you create a remote repository, there will be two ways to make changes to it:

1. By logging in to the hosting service via its website and making changes there directly.
2. By making changes in your local repository and uploading those changes to the remote repository on the hosting service.

In both cases, you will need to *authenticate*, or verify your identity.

Authentication is important, to control who logs in to your hosting service account and who uploads changes to a remote repository.

In the first case, you will use your username or email address and password to authenticate and log in to your hosting service account.

But what about in the second case? How will your hosting service know that it should allow you to upload things from a local repository to a remote repository?

The answer is that you will have to authenticate through the protocol you choose to use. In this chapter, we will cover the HTTPS and SSH protocols. You need to set up only one of these, and you can choose which one you prefer.

[NOTE]

In [Chapter 7](#) you will go through the process to create a remote repository for your Rainbow project, which will produce two URLs: an HTTPS URL and an SSH URL. You will have to use the URL for the protocol you decide to use.

We'll cover the HTTPS protocol first. If you only want to learn about and use the SSH protocol, then you may skip the following section and go straight to [“Using SSH” on page 93](#). Keep in mind that you don't need to understand these protocols in depth in order to use them.

If you're not sure which protocol to choose, I recommend setting up the HTTPS protocol to carry out the exercises in this book because the process is slightly simpler and the examples in this book will use the HTTPS protocol. Whichever protocol you choose to set up, you can always choose to set up the other one as well in your future work with Git, so this isn't a critical decision right now.

USING HTTPS

The HTTPS protocol uses a username and some sort of password (or authentication credential) to allow you to securely connect to remote repositories. In the past, all the hosting services allowed you to use the password you use to log in to your account on the hosting service (which we will refer to as the *account password*) for HTTPS authentication as well. However, GitHub and Bitbucket no longer allow this; they require you to create another authentication credential.

In GitHub, the authentication credential is called a *personal access token*. In Bitbucket, it's called an *app password*. With GitLab, you can still simply use your account password to authenticate. See [Table 6-1](#) for an overview

of the three most common hosting services and the access credentials necessary to authenticate over HTTPS.

TABLE 6-1. Hosting services and necessary access credentials for authenticating over HTTPS

HOSTING SERVICE	USERNAME	PASSWORD
GitHub	Email address or username	Personal access token
GitLab	Email address or username	Account password
Bitbucket	Email address or username	App password

Go to [Follow Along 6-2](#) to prepare your authentication credential.

[FOLLOW ALONG 6-2]

- 1 If you are using GitHub or Bitbucket and you have chosen to use the HTTPS protocol, then go to the Learning Git repository (<https://github.com/gitlearningjourney/learning-git>) for more information on how to set up your authentication credential. You may skip the following section. If you have decided to use the SSH protocol, continue on to “Using SSH.”

USING SSH

The SSH protocol uses a public and private SSH key pair to allow you to securely connect to remote repositories. The three main steps to setting up SSH access are:

1. Create an SSH key pair on your computer.
2. Add the private SSH key to the SSH agent.

3. Add the public SSH key to the hosting service account.

Go to [Follow Along 6-3](#) to carry out the three steps to set up SSH access.

[FOLLOW ALONG 6-3]

- 1** If you have chosen to use the SSH protocol, then go to the Learning Git repository (<https://github.com/gitlearningjourney/learning-git>) for more information on how to set up authentication details to connect over SSH.

[NOTE]

Sharing your private SSH key is a security risk. Think of it as a password, and do not share it with anyone.

Now that you have prepared your authentication details to connect to remote repositories over either HTTPS or SSH, you may continue to the next chapter to create the first remote repository for the Rainbow project.

Summary

In this chapter, you chose which hosting service you wanted to use to complete the rest of the exercises in this book, and you prepared the

authentication details you will need to connect to remote repositories over either HTTPS or SSH. In [Chapter 7](#), you will learn about creating remote repositories and start exploring how you can work with others on Git projects.

Creating and Pushing to a Remote Repository

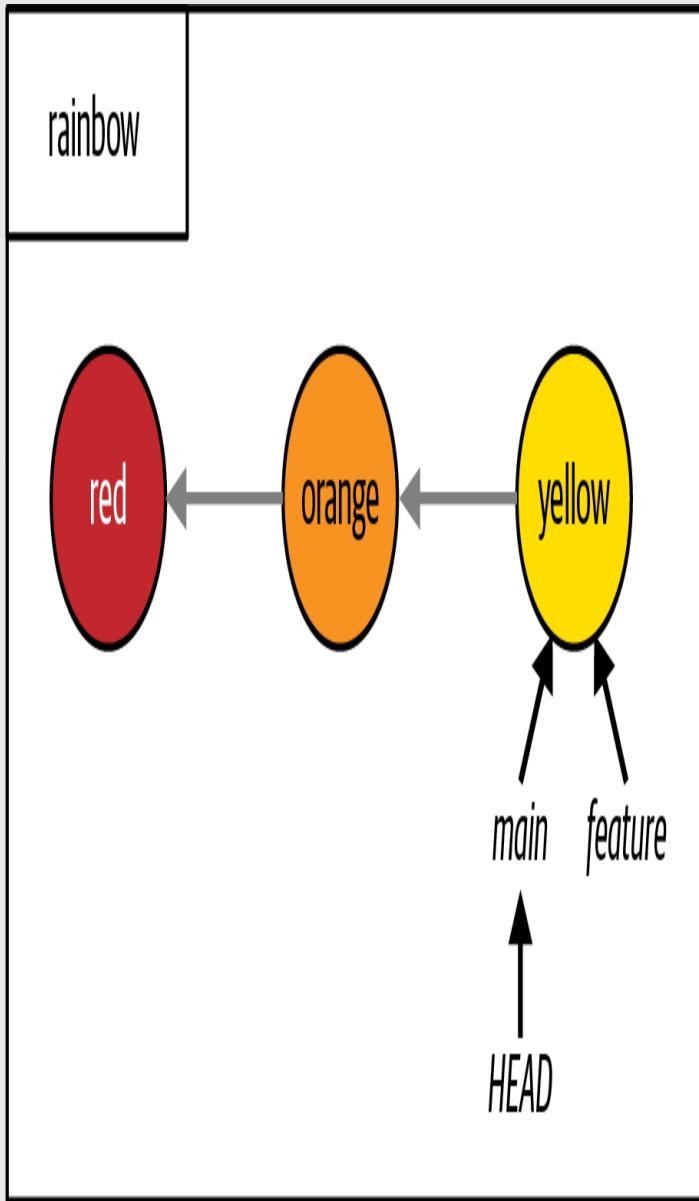
In the last chapter, we went over choosing which hosting service you want to use, creating an account, and setting up authentication details to securely connect to remote repositories over HTTPS or SSH.

In this chapter, we will look at the different ways you can use either local or remote repositories to start working on a Git project and why remote repositories are useful. You will carry out the steps to create a remote repository for the Rainbow project, and you'll upload some data to it. Additional resources to assist you as you work through this chapter are available in the Learning Git repository (<https://github.com/gitlearningjourney/learning-git>).

State of the Local Repository

At the start of this chapter you should have three commits and two branches in your `rainbow` repository, and you should be on the `main` branch. The current state of the `rainbow` repository is illustrated in [Visualize it 7-1](#).

[VISUALIZE IT 7-1]



The `rainbow` repository at the start of [Chapter 7](#)

The Two Ways to Start Work on a Git Project

In [Chapter 2](#) you learned that there are two types of repositories: local repositories, which are stored on a computer, and remote repositories, which

are hosted on a hosting service. You can start working on a project using Git from either a local or a remote repository. Up until now in the Rainbow project, you have taken the first approach because you started working on your project from a local repository. In [Chapter 8](#), you will walk through a hands-on example of starting to work on a Git project from a remote repository. I'll provide a brief overview of each approach in this section.

STARTING FROM A LOCAL REPOSITORY

To start to work on a Git project from a local repository, you must first create a local repository on a computer using the `git init` command and make at least one commit. Next, you must create a remote repository on a hosting service. Finally, you may upload data from the local repository to the remote repository.

In Git, we use the term *push* or *pushing* to refer to the process of uploading data from a local repository to a remote repository, and the command we use to do this is `git push`.

[SAVE THE COMMAND]

`git push`

Upload data to a remote repository

This is the approach you are going to take with the Rainbow project in this chapter, and it is illustrated in [Figure 7-1](#).

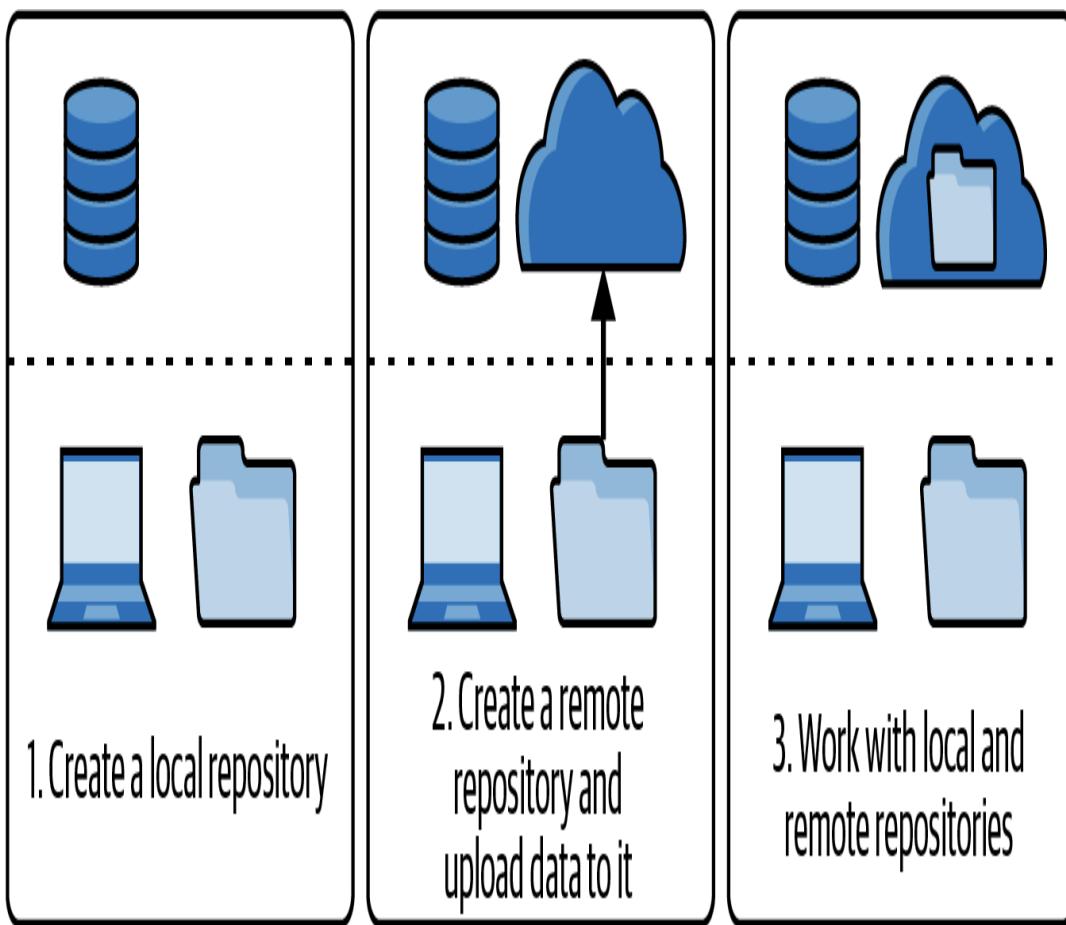


FIGURE 7-1

Starting to work on a Git project from a local repository

An example of a situation where you would start to work on a Git project from a local repository is if you have a project on your computer that you've been working on for a while that is not a repository—in other words, you are not using Git to version control it—and then you decide that you want to start version controlling it with Git. In this case, you would use the `git init` command to initialize the local repository on your computer, make an initial commit, and then carry out the rest of the steps to create a remote repository and upload data to it.

The other way to start working on a Git project is to start from a remote repository.

STARTING FROM A REMOTE REPOSITORY

To start to work on a Git project from a remote repository, you can either find a remote repository that you want to work on or create a new remote repository on a hosting service. Then, you *clone* (in other words, copy) the remote repository to your computer, which will create a local repository. This is the approach we will cover in [Chapter 8](#), and it is illustrated in [Figure 7-2](#).

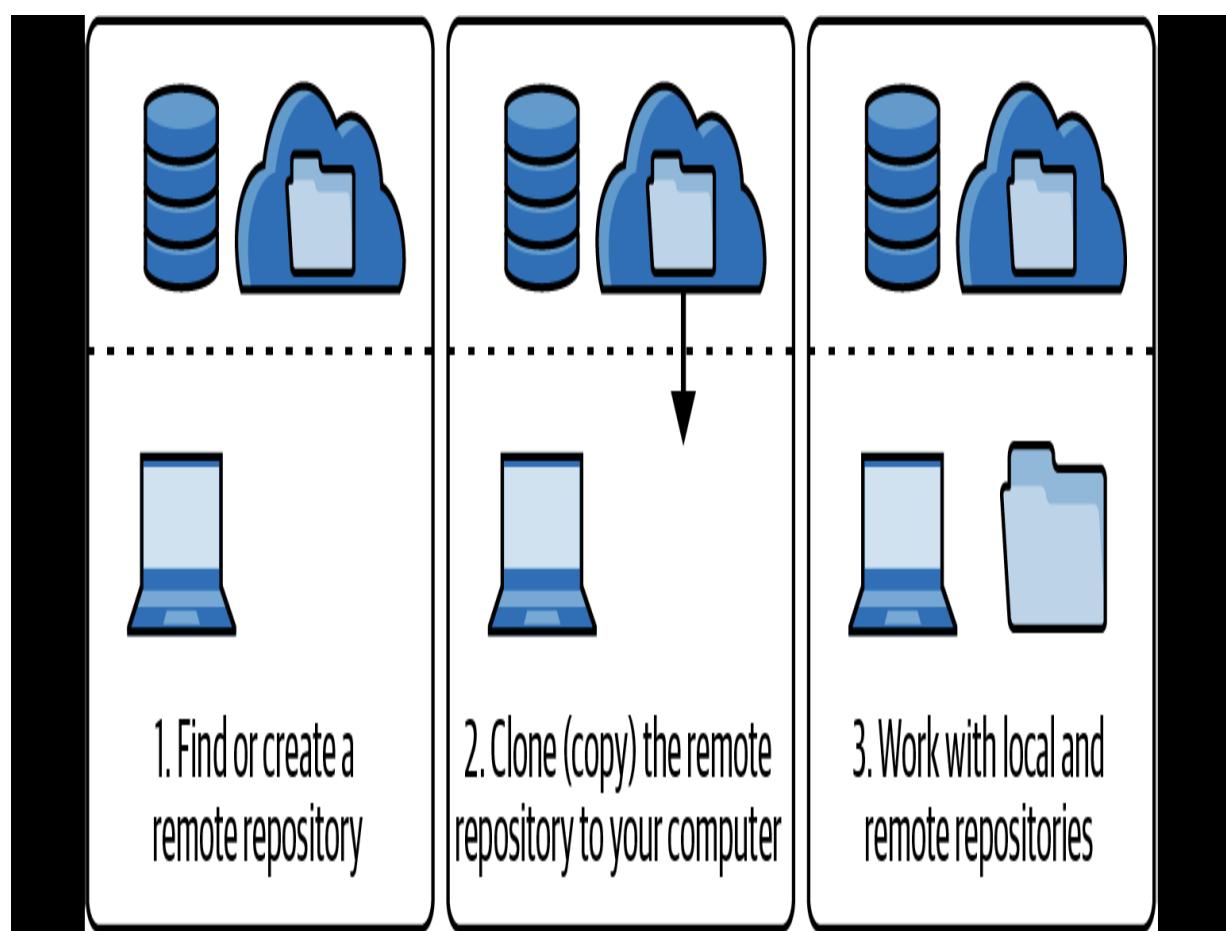


FIGURE 7-2

Starting to work on a Git project from a remote repository

An example of when you would start working on a project from a remote repository is if your friend is working on a project that has a remote repository, and they ask you to contribute to it. In this case, you would ask your friend to tell you where to find the remote repository, and then you would clone (or copy) it to your computer and start working on it.

Up to this point, with the Rainbow project you've been working exclusively with a local repository. Later in this chapter, you will create a remote repository and upload data to it. But first, I want to make a quick point about the interaction between local repositories and remote repositories.

The Interaction Between Local and Remote Repositories

Local repositories and remote repositories act separately. When it comes to working with them, it's important to understand that no interaction between them happens automatically. In other words, no updates from the local repository to the remote repository will happen automatically, and conversely, no updates from the remote repository to the local repository will happen automatically.

There is no live connection between the two. Any changes in either will be the result of you explicitly executing commands. In this and the following chapters, you will learn what these commands are.

Before you go on to create a remote repository, let's discuss why you would want to create one in the first place.

Why Do We Use Remote Repositories?

There are three main reasons why remote repositories are useful and important when working on a Git project. They allow you to:

- Easily back up your project somewhere other than your computer.
- Access a Git project from multiple computers.
- Collaborate with others on Git projects.

Let's take a look at [Example Book Project 7-1](#) to see why you might want to create a remote repository for a Git project.

Example Book Project 7-1

Suppose I had only a local repository on my computer for my Book project, and no remote repository. If someone stole my laptop or the hard drive died, I would lose all the work I had done on my book.

Remote repositories are a good backup mechanism.

A remote repository also will allow me to access my project from other computers. Imagine I have two laptops—one at home and one in my office that I work on during the day—and my Book project is stored on the laptop at home. If I have a remote repository for it as well, I can also access the Book project from my work laptop. Then I can either make changes to it directly on the hosting service website, or I can clone the remote repository onto my work laptop to create a local repository. An additional benefit is that I can show my progress on the book to my coworkers.

A remote repository also allows me to collaborate easily with others on my Book project. I can give access to the remote repository to anyone that wants to view my progress on it or contribute to it. Other people will then have the ability to either clone the remote repository onto their computer to make a local repository, or to simply view the contents of the remote repository directly on the hosting service website. For example, I need a remote repository to collaborate on my Book project with my coauthor and my editor.

[Example Book Project 7-1](#) illustrates why working with remote repositories is so useful. Now that you understand this, let's go over how to create one. Suppose you've told a friend about your experience working on the

Rainbow project, and they want to see what you've been doing. To make this possible, you'll need to create a remote repository. Let's do that now.

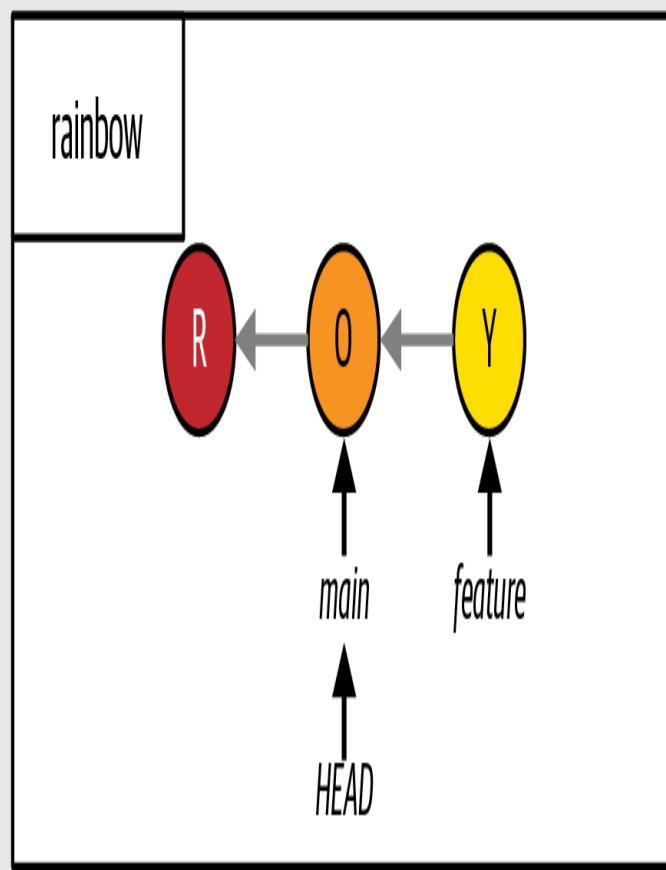
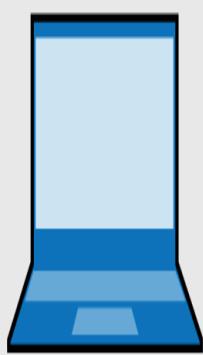
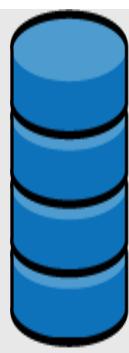
Creating a Remote Repository with Data

The Rainbow project currently consists of one local repository: your `rainbow` repository, shown in [Visualize it 7-2](#).

[NOTE]

From this point forward in the book, we are going to expand the Repository Diagram to include a space at the top to represent what is happening in the remote repository, as seen in Visualize It 7-2.

[VISUALIZE IT 7-2]



The Rainbow project currently consists of one local repository, called `rainbow`

Your friend wants to see what you've been working on. To enable this, you need to create a remote repository and upload some data to it. There are three steps to this process:

1. Create the remote repository on the hosting service.
2. Add a connection to the remote repository in the local repository.
3. Upload (or push) data from the local repository to the remote repository.

In the following sections we'll walk through each of these, and along the way I'll introduce some new kinds of branches.

CREATING THE REMOTE REPOSITORY

When you create a remote repository on a hosting service, you will give it a *remote repository project name* and the hosting service will provide a *remote repository URL*. The remote repository URL will automatically include the remote repository project name. As you learned in [Chapter 6](#), the hosting service will generate an SSH URL and an HTTPS URL for your repository. You should use the one for the protocol you have chosen to use.

The process of creating the remote repository is done entirely on the hosting service's website. You will need to consult your hosting service's documentation for details on the specific steps to follow. For additional resources, go to the Learning Git repository (<https://github.com/gitlearningjourney/learning-git>).

Here, I'll provide you with some general guidance. As mentioned previously, when you create your remote repository you will have to

provide the project name. For the Rainbow project, this will be `rainbow-remote`.

[NOTE]

It is common for a remote repository and a local repository to share the same name. However, to make it easier for you to distinguish between the local and remote repositories in the Rainbow project, we'll give them distinct names.

You also will have to choose whether the remote repository will be public or private. This is a setting you can adjust on a hosting service for each repository. A *public repository* is visible to anyone on the internet. A *private repository* is visible only to the individuals given access to it. In both cases, you can contribute to the repository only if you are provided with access. Since the repository you are working on in this book is for learning purposes only, I recommend you make it private. This way you can control who is able to view it (for example, you can grant access to the friend who wants to see what you've been working on).

Some hosting services may ask if you want to include any files when you create the repository, such as a `README` or `.gitignore` file. For the purposes of this book, you should not include any files; if the option to include them is selected, make sure to deselect it. Since you will be uploading data from your local repository to the remote repository, you want your remote repository to be empty.

If the hosting service provides an option to choose a license, you can ignore this (i.e., don't select a license), as this is just a learning exercise.

Finally, some hosting services may ask you to enter a value for the default branch name. For the exercises in this book, you may either leave this value blank or enter `main` as the value.

Now, go to [Follow Along 7-1](#) to create your remote repository.

[FOLLOW ALONG 7-1]

1

Log in to your account on your hosting service of choice.

2

Complete the steps to create a remote repository. For more information on this, go to the Learning Git repository

(<https://github.com/gitlearningjourney/learning-git>) or consult your hosting service's documentation directly.

When creating the repository for this exercise:

- For the repository name, use `rainbow-remote`.
- You may choose to make the repository public or private. I recommend making it private.
- Do not include any files. For example, do not include a `README` file or a `.gitignore` file.
- If you are asked to provide a default branch name, you may leave the field blank or set it to `main`.

3

Once you've finished the steps to create the remote repository, locate the remote repository URL. There will be two versions of the URL, one for HTTPS and one for SSH. In an upcoming exercise, you will have to use the URL for the protocol you set up in [Chapter 6](#). In the examples in this book, the two remote repository URLs are:

- HTTPS: <https://github.com/gitlearningjourney/rainbow-remote.git>
- SSH: <git@github.com:gitlearningjourney/rainbow-remote.git>

What to notice:

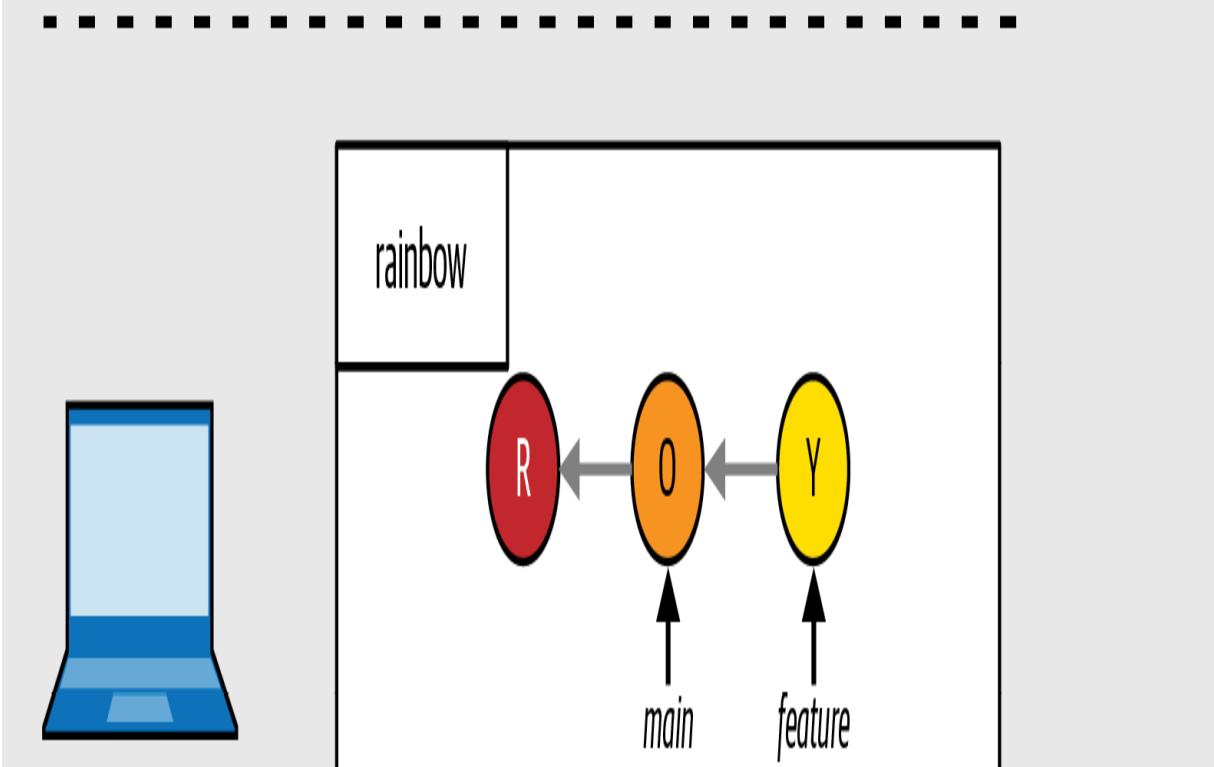
- You created a remote repository called `rainbow-remote`; however, there is no data in the remote repository.

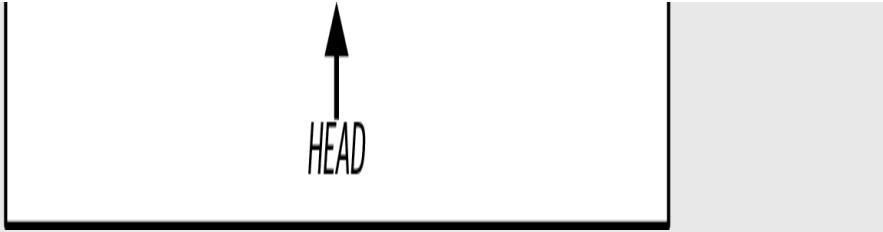
The state of the Rainbow project after completing the Follow Along is illustrated in [Visualize it 7-3](#).

[NOTE]

From this point forward, in the Repository Diagram, we use a rectangle with normal 90-degree corners to represent a local repository and a rectangle with rounded corners to represent a remote repository.

[VISUALIZE IT 7-3]





The Rainbow project after you create the empty remote repository, [rainbow-remote](#)

You've now created a remote repository on your hosting service, but as this Visualize It shows, it's currently empty. Creating a remote repository on a hosting service does not upload any data to it. To upload data, you need to first add a connection to the remote repository in your local repository. Let's take a look at how to do this.

ADDING A CONNECTION TO THE REMOTE REPOSITORY

A local repository can communicate with a remote repository when the local repository has a connection to the remote repository stored within it. This connection will have a name, which we refer to as the *remote repository shortname* or just *shortname*. A local repository can have connections to multiple remote repositories, although this isn't very common.

If a local repository was initialized locally, to set up a connection to a remote repository you must explicitly associate the remote repository URL to the remote repository shortname. To do this you use the [`git remote add`](#) command, passing in the shortname followed by the remote repository URL.

[SAVE THE COMMAND]

`git remote add <shortname> <URL>`

Add a connection to a remote repository named `<shortname>` at `<URL>`

[NOTE]

The examples in this book will use the HTTPS URL; however, you may choose to use either your SSH URL or your HTTPS URL, depending on which protocol you have chosen to use. Be sure to use the URL provided by your hosting service, which will be different from the URL used in this book's examples.

Once a connection to a remote repository is stored in a local repository, you are able to connect to the remote repository by referring to the shortname rather than the URL in the command line.

[NOTE]

When you clone a remote repository to create a local repository, Git automatically adds a connection to the remote repository with the default shortname `origin`. You will see this in practice in [“Cloning a Remote Repository” on page 114](#). It is common even when a local repository is initialized locally to use `origin` for the shortname, as this is the default shortname for a repository when it is cloned.

This is a similar concept to how, as mentioned in [“A Bit of Git History: master and main” on page 48](#), when we initialize a repository using the `git init` command with no additional options Git automatically creates a branch with the default name `master`. Again, there is nothing special about the terms `origin` or `master`; they are simply the current default naming conventions used by Git.

The `rainbow` repository was initialized locally, so you are going to have to add a connection to the remote repository in the local repository explicitly. Before you go ahead and do that, let's first go over some additional useful commands.

[NOTE]

In the official Git documentation, a connection to a remote repository stored in a local repository is simply referred to as a *remote*.

To see the list of connections to remote repositories stored in a local repository by shortname, you may use the `git remote` command. If you pass in the `-v` option (which stands for “verbose”) to the `git remote` command, then it lists the connections to remote repositories stored in a local repository by shortname along with their URLs.

[SAVE THE COMMAND]

`git remote`

List the remote repository connections stored in the local repository by shortname

`git remote -v`

List the remote repository connections in the local repository with shortnames and URLs

Go to [Follow Along 7-2](#) now, and associate your URL to the shortname `origin` in the `rainbow` repository. In step 4, be sure to enter the entire command

on a single line.

[FOLLOW ALONG 7-2]

1 Using a filesystem window, go to `rainbow > .git > config`. Open the `config` file. You will see that at the moment there are no connections to remote repositories listed in the file. In other words, you will not see any mentions of your remote repository URL or shortname in the file.

2 Go to your hosting service and copy the remote repository URL for the protocol you have chosen (either SSH or HTTPS). You will need to use this URL in step 4 of this Follow Along.

3 `rainbow $ git remote`

4 `rainbow $ git remote add origin https://github.com/gitlearningjourney/rainbow-remote.git`

5 `rainbow $ git remote`

6 `rainbow $ git remote -v`

```
origin  https://github.com/gitlearningjourney/rainbow-remote.git (fetch)
origin  https://github.com/gitlearningjourney/rainbow-remote.git (push)
```

7 Close the `config` file and open it again in a new window. You will see that there is one connection to a remote repository listed in the file. It should look similar to:

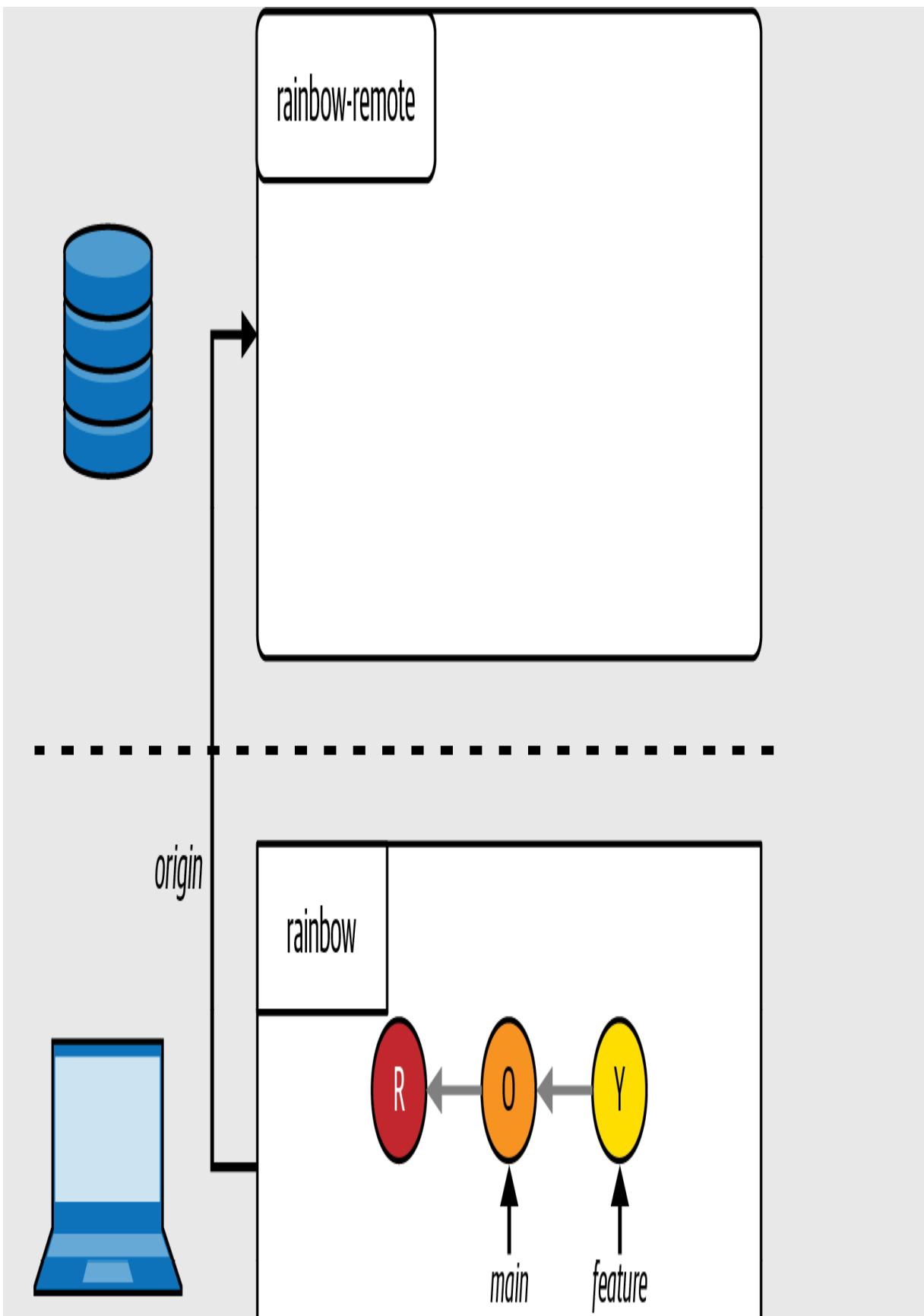
```
[remote "origin"]
url = https://github.com/gitlearningjourney/rainbow-remote.git
fetch = +refs/heads/*:refs/remotes/origin/*
```

What to notice:

- In step 3, there are no shortnames listed under the `git remote` output.
- In step 5, there is one shortname listed under the `git remote` output, which is `origin`.

The addition of this connection is illustrated in [Visualize it 7-4](#).

[VISUALIZE IT 7-4]





The Rainbow project after you add a connection to the `rainbow-remote` repository in your `rainbow` repository

What to notice:

- The `rainbow` repository has a shortname associated with the remote repository URL, called `origin`.
- The `rainbow-remote` repository still does not have any data in it.

In [Visualize it 7-4](#), you can see that the arrow that represents the shortname stored in the `rainbow` repository that relates to the `rainbow-remote` repository is going in only one direction. This is because the connection between a local and a remote repository goes from the local repository to the remote repository but not the other way around. In a local repository, you can find a list of all the remote repositories it has stored a connection to; however, in a remote repository you cannot find a list of local repositories that store a connection to that remote repository.

Notice as well that just because you added a connection to the remote repository in your local repository does not mean that any data from the local repository was uploaded to the remote repository. To upload data to a remote repository, you must push a branch to the remote repository. This process will upload all the commits that are part of that branch.

To help you understand what happens when you carry out this final step, I need to introduce a few other kinds of branches.

INTRODUCING REMOTE BRANCHES AND REMOTE-TRACKING BRANCHES

In [Chapter 4](#) you learned about branches, which as you saw are movable pointers to commits. Up until now, you have worked only with local branches. When you push a local branch to a remote repository, you will create a *remote branch*. A remote branch is a branch in a remote repository.

Remote branches *do not* automatically update when you make more commits on local branches. You have to explicitly push commits from a local branch to a remote branch. Every remote branch (that a local repository knows about) also has a *remote-tracking branch*. This is a reference in a local repository to the commit a remote branch pointed at the last time any network communication happened with the remote repository. You can think of it as being like a bookmark.

You can set up a tracking relationship between a local branch and a remote branch by defining which remote branch a local branch should track. This is referred to as the *upstream branch*. There are some cases where Git will set the upstream branch automatically, but in other cases you have to set it explicitly.

When you push work from a local branch to a remote branch, Git needs to know which remote branch you want to push to. If the local branch has an upstream branch defined for it, you can use `git push` with no arguments, and Git will automatically push the work to that branch. However, if no upstream branch is defined for the local branch you’re working on, you’ll need to specify which remote branch to push to when you enter the `git push` command. (If you don’t, you’ll get an error message.)

In this chapter, we will show how to specify the remote branch you want to push to when using the `git push` command. In [Chapter 9](#), you will learn how

to define an upstream branch.

You're now ready to complete the third step of creating a remote repository: pushing a local branch to the remote repository.

PUSHING TO A REMOTE REPOSITORY

To push a local branch to your remote repository, you will use the `git push` command and pass in the shortname for the remote repository and the name of the branch that you want to push. For the `rainbow` repository, the shortname you will use is `origin` and the branch you will push is `main`.

[SAVE THE COMMAND]

`git push <shortname> <branch_name>`

Upload content from `<branch_name>` to the `<shortname>` remote repository

After you execute the `git push` command, two things will happen:

1. A remote branch will be created in your remote repository.
2. A remote-tracking branch will be created in your local repository.

[NOTE]

To push data to a remote repository, you must be connected to the internet and have either SSH access or HTTPS access to your hosting service of choice.

In [Chapter 4](#), you used the `git branch` command to list all the local branches. Now you will use the `git branch` command with the `--all` option to list all the local and remote-tracking branches in your local repository.

[SAVE THE COMMAND]

git branch --all

List local branches and remote-tracking branches

Go to [Follow Along 7-3](#) to push your branch to the remote repository.

[FOLLOW ALONG 7-3]

1 Using a filesystem window, go to `rainbow > .git > refs` and look at the directories that are inside the `refs` directory. There should be two directories: `heads` and `tags`.

2

```
rainbow $ git push origin main
Enumerating objects: 9, done.

Counting objects: 100% (9/9), done.

Delta compression using up to 4 threads

Compressing objects: 100% (5/5), done.

Writing objects: 100% (9/9), 747 bytes | 373.00 KiB/s, done.

Total 9 (delta 1), reused 0 (delta 0), pack-reused 0

remote: Resolving deltas: 100% (1/1), done.

To github.com:gitlearningjourney/rainbow-remote.git
 * [new branch]      main -> main
```

3 In the filesystem window, look at the directories that are inside the `refs` directory again. There should now be three directories: `heads`, `tags`, and `remotes`.

4 Go to your `rainbow-remote` repository on your hosting service and refresh the page. Go to the page that lists your commits. You should see your three commits there now.

5

```
rainbow $ git branch --all
feature
* main
remotes/origin/main
```

[FOLLOW ALONG 7-3]

6

```
rainbow $ git log
commit fc8139cbf8442cdbb5e469285abaac6de919ace6 (HEAD -> main,
origin/main, feature)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 10:09:59 2022 +0100
yellow
commit 7acb333f08e12020efb5c6b563b285040c9dba93
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 09:42:07 2022 +0100
orange
commit c26d0bc371c3634ab49543686b3c8f10e9da63c5
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 09:23:18 2022 +0100
red
```

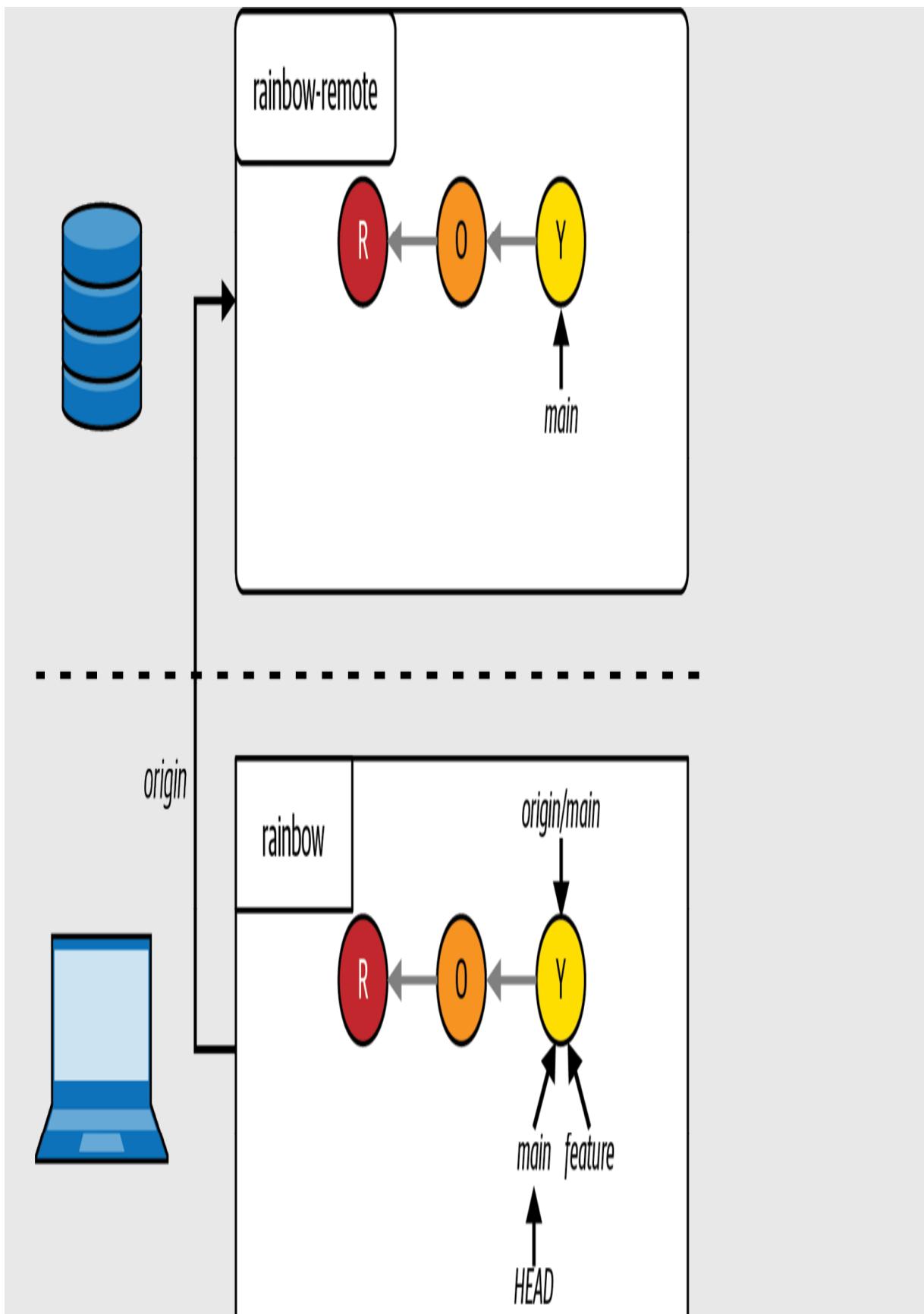
What to notice:

- In step 2, the output of the `git push` command indicates that you have pushed your branch to the remote repository. Note that it doesn't matter if the numbers in your output are slightly different.
- In step 3, you can see that the `refs` directory has a new directory inside it called `remotes`. Inside the `remotes` directory is a directory called `origin`, and inside this directory is a file called `main`. This represents the new `origin/main` remote-tracking branch.

- In step 4, you can see that in the `rainbow-remote` repository, there is a remote `main` branch. If you go to your hosting service, you will find the three commits of the local `main` branch.
- There is no `feature` branch in the remote repository.

All of these observations are illustrated in [Visualize it 7-5](#).

[VISUALIZE IT 7-5]



The Rainbow project after you push your local `main` branch to the remote repository

[NOTE]

For commands like `git commit`, `git push`, and `git merge`, it doesn't matter if your output has slightly different numbers than the output in this book. The same goes for other commands you'll use in later chapters in this book such as `git clone`, `git fetch`, and `git pull`.

Notice that when you push a specific branch to a remote repository, only the data from that branch is uploaded to the remote repository. You pushed the `main` branch to the remote repository, but the `feature` branch was not pushed to the remote repository.

At the moment, in your `rainbow` repository, the `main` and `feature` branches have the same commits. In other words, their development histories, which can be traced by following the commits and the parents links backward, consist of the same commits. Therefore, it is not so obvious in [Visualize it 7-5](#) that the `feature` branch is not in the remote repository. However, in real-world projects, your branches will often have different development histories and therefore consist of different commits. This means that if you don't push a particular branch to the remote repository, you won't have some of the commits in your remote repository.

If you also want to push your `feature` branch to the remote repository, you have to do so explicitly. Go to [Follow Along 7-4](#) and do this now.

[FOLLOW ALONG 7-4]

1 rainbow \$ **git switch feature**

Switched to branch 'feature'

2 rainbow \$ **git push origin feature**

Total 0 (delta 0), reused 0 (delta 0), pack-reused 0

remote:

remote: Create a pull request for 'feature' on GitHub by visiting:

remote: <https://github.com/gitlearningjourney/rainbow->

remote/pull/new/feature

remote:

To [github.com:gitlearningjourney/rainbow-remote.git](https://github.com/gitlearningjourney/rainbow-remote.git)

* [new branch] feature -> feature

3 rainbow \$ **git branch --all**

* **feature**

main

remotes/origin/feature

remotes/origin/main

4

Go to your **rainbow-remote** repository on your hosting service and refresh the page. Look at the list of branches.

[FOLLOW ALONG 7-4]

5

```
rainbow $ git log
commit fc8139cbf8442cdbb5e469285abaac6de919ace6 (HEAD -> feature,
origin/main, origin/feature, main)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 10:09:59 2022 +0100

yellow

commit 7acb333f08e12020efb5c6b563b285040c9dba93
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 09:42:07 2022 +0100

orange
```

[NOTE]

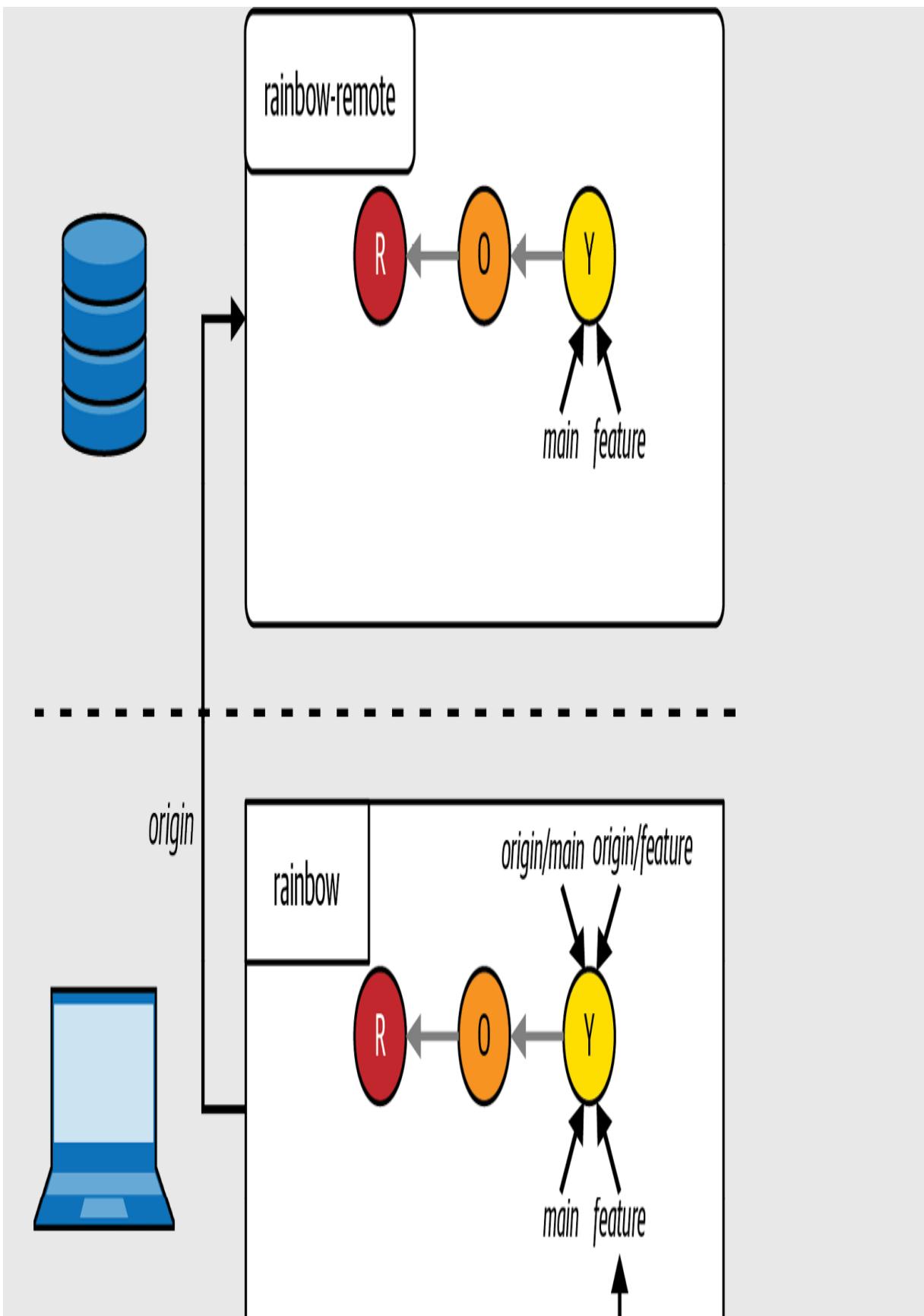
From this point forward in the book, the `git log` output in Follow Alongs will display only the last few commits that were made in a given repository.

What to notice:

- In step 3, you can see that in the `rainbow` repository there are two remote-tracking branches, `origin/main` and `origin/feature`.
- In step 4, you can see that in the `rainbow-remote` repository there are two remote branches, `main` and `feature`.

This is illustrated in [Visualize it 7-6](#).

[VISUALIZE IT 7-6]



The diagram illustrates a local Git repository structure. It features a central white rectangular box with a black border. Inside this box, the word "HEAD" is written vertically in black capital letters. To the left of this box is a smaller gray rectangular box containing the word "feature" in red capital letters. A horizontal line connects the bottom of the "feature" box to the top of the "HEAD" box, indicating that the local "feature" branch points to the remote "HEAD" branch.

The Rainbow project after you push the local `feature` branch to the remote repository

You have now created a remote repository with data in it. In [Chapter 6](#), I mentioned that there are two ways to make changes to a remote repository:

1. By logging in to the hosting service via its website and making changes there directly.
2. By making changes in your local repository and uploading those changes to the remote repository on the hosting service.

In this chapter we covered an example of the second situation, but let's talk briefly about the first one.

Working on a Remote Repository Directly on a Hosting Service

Up until now, you have learned how to use your local repository to make commits ([Chapter 3](#)), work with branches ([Chapter 4](#)), and carry out a basic merge ([Chapter 5](#)).

However, it is also possible to use the UI of a hosting service's website to carry out a lot of actions (and many more) that are similar to these. For example, on a hosting service you can make commits directly on a remote repository, you can create remote branches, and, as you will see in [Chapter 12](#), you can merge branches through a feature called a pull request.

Because this book focuses on teaching how to use Git in the command line, we won't cover how to carry out all the corresponding actions in a hosting

service. However, all of these features will be comprehensively documented in your hosting service’s documentation. These features may be helpful for collaborators you work with in the future that don’t know how to use Git.

Now that you have learned the basics of working with local and remote repositories and you have created a remote repository for the Rainbow project and pushed some branches to it, the next step in your learning journey is to start collaborating on the project.

Summary

This chapter introduced what remote repositories are, why we use them, and the three steps to create a remote repository with data in it when you’re starting from a local repository. You set up a remote repository on your hosting service, added a connection to the remote repository in your local repository, and pushed data from the local repository to the remote repository. Along the way, you learned about the concepts of remote branches, remote-tracking branches, and upstream branches.

Next, in [Chapter 8](#), your friend will start helping you work on your project by cloning the remote repository.

Cloning and Fetching

In the last chapter, you learned about remote repositories and you created one for the Rainbow project.

In this chapter, you are going to start simulating what it would be like to work with a friend on the Rainbow project. To do that, you are going to learn about cloning remote repositories and how this differs from initializing repositories locally. You will also learn more about defining upstream branches, deleting branches, and fetching data from a remote repository.

[NOTE]

Some text editors that have Git integrations allow you to define certain settings for Git. This chapter assumes that you have not enabled any special settings that will cause Git to deviate from its default behavior.

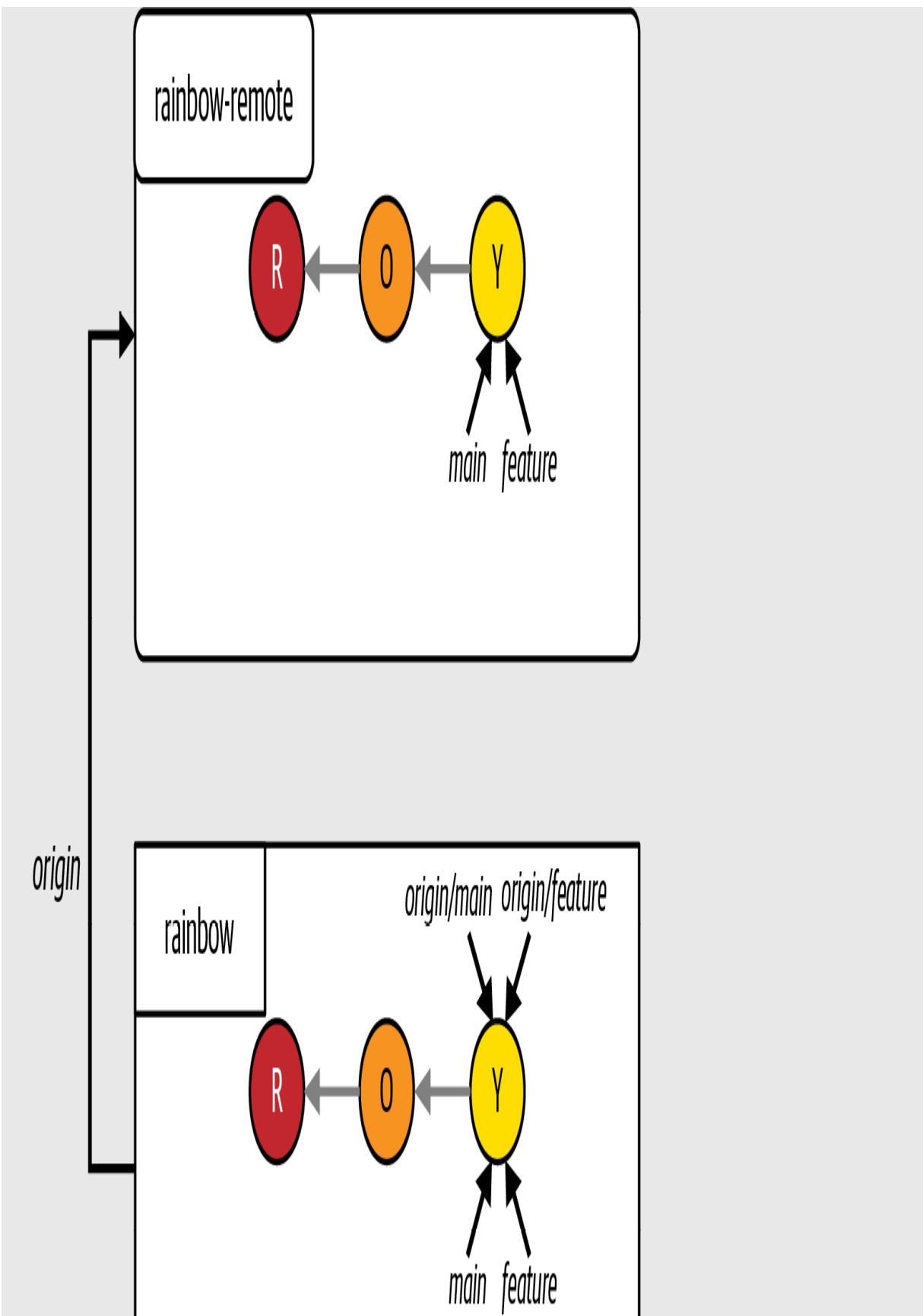
For example, Visual Studio Code has a feature called Autofetch (`git.autofetch`) that will periodically fetch changes from a remote repository. (You will learn about fetching later in this chapter.) This feature is disabled by default, and for you to be able to do the exercises in this chapter properly it should remain disabled. If you have enabled it in the past, then please disable it before continuing.

If you have never configured any special Git settings for your text editor, you don't need to worry about this.

State of the Local and Remote Repositories

At the start of this chapter, you should have a local repository called `rainbow` and a remote repository called `rainbow-remote`. These two repositories should be in sync; that is, they should contain the same commits and branches. The expected current state of the two repositories is illustrated in [Visualize it 8-1](#).

[VISUALIZE IT 8-1]





The Rainbow project at the start of [Chapter 8](#), with a local repository called `rainbow` and a remote repository called `rainbow-remote`

Cloning a Remote Repository

In the previous chapter you created a remote repository on a hosting service so you could show your friend the work you've been doing on the Rainbow project. Now, let's pretend that your friend has decided they want to help you work on the project. If they're going to contribute to the project, they will need to clone (that is, copy) your remote repository. In the following section I will describe how you will simulate this scenario.

At the start of the book, you learned that Git is a useful tool for collaboration. Cloning remote repositories is an essential part of being able to collaborate with other people on a Git project, since it allows them to work with their own copy of the repository on their computer. An unlimited number of people can clone a remote repository (as long as they are provided with access to it).

Let's take a look at [Example Book Project 8-1](#) for an illustration.

Example Book Project 8-1

Suppose I initially planned to work on my Book project on my own, then later changed my mind and decided that I wanted to work with a coauthor.

My coauthor will need to clone the remote repository to have their own copy of the Book project on their computer so they can start working with me on the book. To enable this, they will need to have an account with whichever hosting service is hosting the remote repository. Then, I will have to grant them editing access to the repository so they can comment and contribute, regardless of whether it is public or private.

If at some point I want my editor to review my work, we'll need to follow the same steps: they'll set up an account with the hosting service, then I'll grant them access to the repository and tell them where to find it, and they will clone the repository onto their computer.

[Example Book Project 8-1](#) shows why cloning repositories is such an important part of collaborating with others on any Git project. Next, we'll look at how you are going to simulate the experience of working with someone else on the Rainbow project.

THE COLLABORATION SIMULATION

Normally, if two people are working on the same project, each will have a local repository on their own computer and each will contribute to one remote repository. Given that you may not have two computers or another person available to help you work through the collaboration exercises in this and the following chapters, you are going to clone the remote

repository onto your computer and create a second local repository, which you will call `friend-rainbow` to distinguish it from the `rainbow` repository.

You will then pretend that this second local repository is on your friend's computer. From now on, when we refer to your friend completing some action, you will perform the action yourself in the `friend-rainbow` repository.

After your friend clones the remote repository, you will be working with two local repositories. I recommend working with two separate text editor windows, one for each project directory. I also recommend working with two separate command line windows (or integrated terminals), one for each repository, rather than navigating into and out of each project directory in one command line window.

In Git, we use the term *clone* or *cloning* to refer to the process of copying a remote repository onto a computer to create a local repository and the command we use to do this is `git clone`. To clone a remote repository, you use the `git clone` command and pass in the remote repository URL and, optionally, a project directory name.

[SAVE THE COMMAND]

`git clone <URL> <directory_name>`

Clone a remote repository

If you don't pass a project directory name, then the local repository will be assigned the remote repository project name. For example, if you don't pass a project directory name in the upcoming Follow Along when you clone the `rainbow-remote` repository, then your friend's local repository will also be

called `rainbow-remote`. Because that would make this learning exercise very confusing, instead you are going to pass the project directory name `friend-rainbow` to the `git clone` command as an argument to create a local repository of that name.

The `git clone` command does the following:

1. Create a project directory inside the current directory.
2. Create (initialize) the local repository.
3. Download all the data from the remote repository.
4. Add a connection to the remote repository that was cloned; by default it will have the shortname `origin` in the new local repository.

To explicitly see the repository being created, you can clone it into your `desktop` directory. This will allow you to see the new project directory appear on the desktop of your computer.

Go to [Follow Along 8-1](#) to clone the remote repository. In step 3, be sure to enter the entire command on a single line.

[FOLLOW ALONG 8-1]

1

Open a new command line window and navigate to the `desktop` directory.

2

Go to your hosting service and copy your remote repository URL for the protocol you have chosen to use, either SSH or HTTPS. You will use this URL in step 3 of this Follow Along.

3

```
desktop $ git clone https://github.com/gitlearningjourney/rai  
nbow-remote.git friend-rainbow  
Cloning into 'friend-rainbow'...  
  
remote: Enumerating objects: 9, done.  
  
remote: Counting objects: 100% (9/9), done.  
  
remote: Compressing objects: 100% (4/4), done.  
  
Receiving objects: 100% (9/9), done.  
  
Resolving deltas: 100% (1/1), done.  
  
remote: Total 9 (delta 1), reused 9 (delta 1), pack-reused 0
```

4

Locate the newly created `friend-rainbow` project directory in your filesystem or on your desktop.

What to notice:

- You cloned the remote repository onto your computer and created a second local repository called `friend-rainbow`.
- You are not in the `friend-rainbow` directory in the command line, because cloning a Git repository does not mean you automatically navigate into it.

Go to [Follow Along 8-2](#) to navigate into the new `friend-rainbow` project directory to explore its contents and setup.

[FOLLOW ALONG 8-2]

1 desktop \$ **cd friend-rainbow**

2 Open the `friend-rainbow` project directory in a new text editor window.

3 friend-rainbow \$ **git remote -v**
origin https://github.com/gitlearningjourney/rainbow-remote.git (fetch)
origin https://github.com/gitlearningjourney/rainbow-remote.git (push)

4 friend-rainbow \$ **git branch --all**
* main
 remotes/origin/HEAD -> origin/main
 remotes/origin/feature
 remotes/origin/main

5 friend-rainbow \$ **git log**
commit fc8139cbf8442cdbb5e469285abaac6de919ace6 (**HEAD -> main**,
origin/main, **origin/feature**, **origin/HEAD**)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date: Sat Feb 19 10:09:59 2022 +0100
 yellow
commit 7acb333f08e12020efb5c6b563b285040c9dba93
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date: Sat Feb 19 09:42:07 2022 +0100
 orange

What to notice:

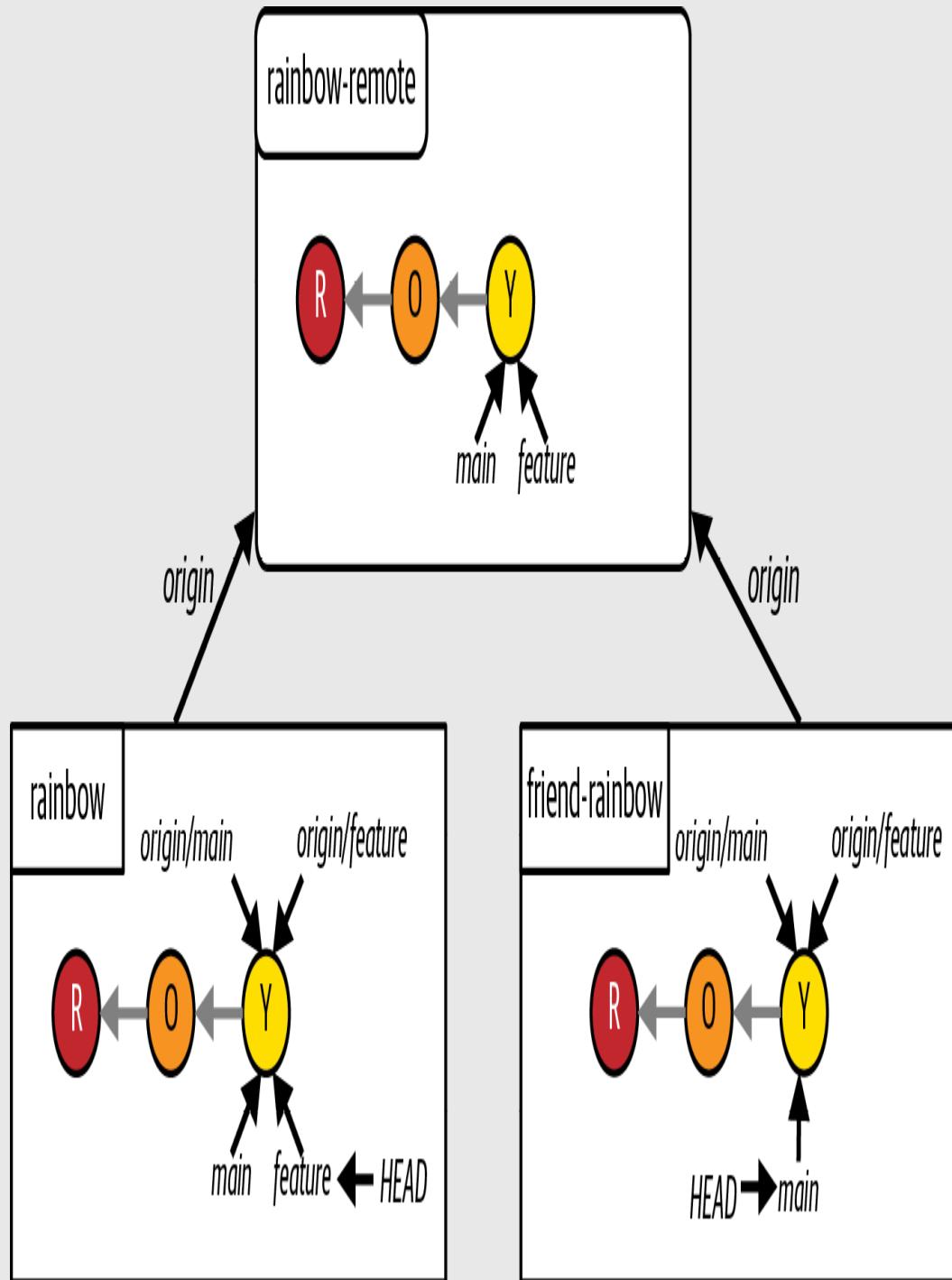
- In step 1, you used the `cd` command to navigate into the `friend-rainbow` repository in the command line. This is necessary because cloning a repository does not mean you navigate into it automatically.
- In step 3, in the `git remote` output, the `origin` remote repository shortname is already listed.
- In step 4, the `git branch` output shows a pointer called `origin/HEAD` that points to the `origin/main` remote-tracking branch, and that you have an `origin/feature` remote-tracking branch. You can also see that there is no local `feature` branch.

All of the preceding observations are illustrated in [Visualize it 8-2](#).

[NOTE]

From now on, I'll expand the Repository Diagrams to depict the state of the two local repositories, `rainbow` and `friend-rainbow`, as well as the remote repository, `rainbow-remote`.

[VISUALIZE IT 8-2]



The Rainbow project after your friend clones the remote repository and creates a local repository called `friend-rainbow`

In the next sections, we will dive deeper into what happens during the cloning process and answer the following questions:

- What is the `origin/HEAD` pointer?
- Why is there no local `feature` branch in the new `friend-rainbow` local repository?
- Why was the `origin` shortname automatically created for the new local repository?

WHAT IS ORIGIN/HEAD?

Earlier, you noticed that there is a pointer called `origin/HEAD` in the `friend-rainbow` repository. What is this?

When you clone a repository, Git needs to know which branch it should be on when it's done cloning. The `origin/HEAD` pointer determines which branch this is. In the Rainbow project, `origin/HEAD` points to the `main` branch, which is why your friend that cloned the repository is on the `main` branch in the `friend-rainbow` repository.

[NOTE]

For simplicity, I will not include the `origin/HEAD` pointer in the Visualize It diagrams.

By contrast, notice that in the `rainbow` repository you are currently on the `feature` branch—but in the `friend-rainbow` repository, the local `feature` branch doesn't even exist. You'll learn why that is in the next section.

CLONING REPOSITORIES AND DIFFERENT TYPES OF BRANCHES

In the `git log` output produced by step 5 of [Follow Along 8-2](#), you can see that in the `friend-rainbow` repository there is no reference to the local `feature` branch. However, there is a reference to the `origin/feature` remote-tracking branch. This is because when you clone a repository the `git clone` command will create remote-tracking branches for all the branches currently present in the remote repository that is being cloned, but the only local branch that is created is the branch that `origin/HEAD` points to.

For your friend to work on the `feature` branch, they must switch onto it. Then Git will create a local `feature` branch based on where the remote-tracking branch was pointing.

Go to [Follow Along 8-3](#) to simulate your friend switching onto the `feature` branch.

[FOLLOW ALONG 8-3]

1 friend-rainbow \$ **git branch --all**

```
* main  
  remotes/origin/HEAD -> origin/main  
  remotes/origin/feature  
  remotes/origin/main
```

2 friend-rainbow \$ **git switch feature**

```
branch 'feature' set up to track 'origin/feature'.  
Switched to a new branch 'feature'
```

3 friend-rainbow \$ **git branch --all**

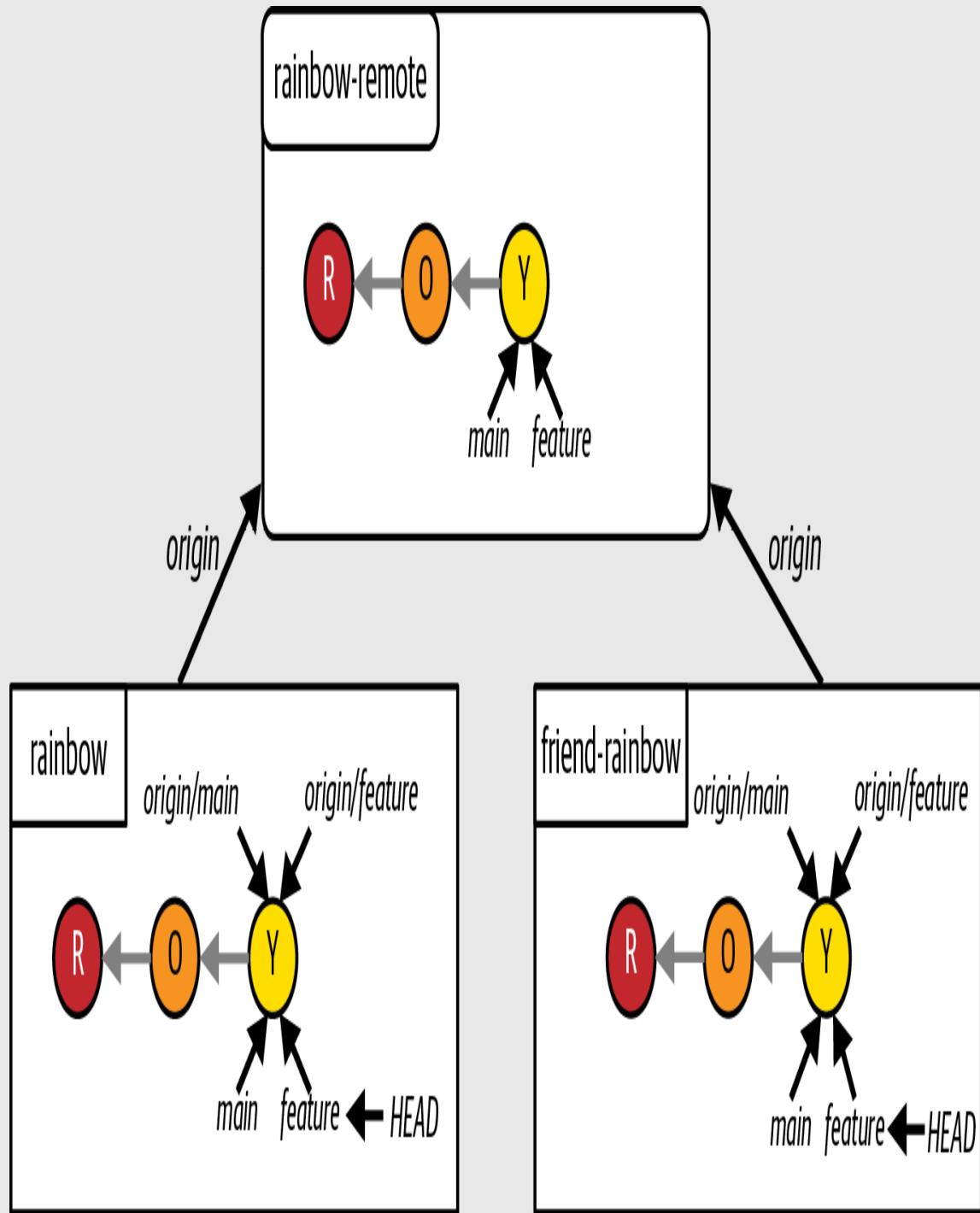
```
* feature  
  main  
  remotes/origin/HEAD -> origin/main  
  remotes/origin/feature  
  remotes/origin/main
```

What to notice:

- In step 3, in the `git branch --all` output, you can see that there is a new local `feature` branch and your friend is on it.

This is illustrated in [Visualize it 8-3](#).

[VISUALIZE IT 8-3]



The Rainbow project after your friend switches onto the `feature` branch

YOU'VE JUST LEARNED HOW TO SWITCH ONTO AND CREATE NEW LOCAL BRANCHES BASED OFF REMOTE-TRACKING BRANCHES THAT YOU DOWNLOADED FROM A REMOTE REPOSITORY. NEXT, LET'S DISCUSS WHY THE FRIEND-RAINBOW REPOSITORY ALREADY HAS THE ORIGIN SHORTNAME ASSIGNED TO THE CONNECTION TO THE REMOTE REPOSITORY.

THE ORIGIN SHORTNAME

Recall that in [Chapter 7](#), you learned that for a local repository to communicate with a remote repository, the local repository must have a connection with a shortname to the remote repository stored within it. To work with the remote repository, you had to explicitly associate its URL with a shortname using the `git remote add <shortname> <URL>` command. This is because you created the `rainbow` repository locally using the `git init` command, and it had not yet had any interaction with the remote repository.

However, in step 3 of [Follow Along 8-2](#), in the `git remote` output, you saw that you already have the `origin` shortname listed. This means the friend-rainbow repository already has the remote repository URL associated with the shortname `origin`. This is because your friend's local repository did not originate locally; it was directly cloned from a remote repository. At the time of cloning the remote repository URL was associated with a shortname in the local repository, and `origin` is the default shortname Git associates with a remote repository when you clone it.

Now that you know a bit more about the cloning process, let's talk some more about branches. In [Chapter 4](#), you learned how to create and switch between branches. Next, you'll learn how to delete them.

Deleting Branches

The main reason to delete branches is to keep a Git project organized and uncluttered. Before deleting a branch, you should always make sure that either you have merged it into another branch or you’re sure you don’t want to use any of the work that is found only on that branch.

[NOTE]

When you delete a branch with commits that are not part of any other branch, you don’t delete the commits that are part of that branch. The commits still exist in your commit history. However, they are no longer easy to reach, because there is no simple branch reference to them and they are not part of the development history of any existing branch.

To demonstrate how branch deletion works, let’s assume that your friend decides they don’t need the `feature` branch, so they want to remove it. To fully delete a branch, you need to delete the remote branch, the remote-tracking branch, and the local branch. To delete a remote branch and a remote-tracking branch, you use the `git push <shortname> -d <branch_name>` command (where `-d` stands for “delete”). With this command, you essentially upload a deletion to the remote repository. You must pass it the name of a remote branch.

[SAVE THE COMMAND]

`git push <shortname> -d <branch_name>`

Delete a remote branch and the associated remote-tracking branch

[NOTE]

You can also delete a remote branch directly on the website of a hosting service, but keep in mind that this will not delete the remote-tracking branch. We'll walk through this process in [“Deleting Remote Branches” on page 249](#).

To delete a local branch, you use the `git branch` command with the `-d` option, passing in the name of the branch you want to delete.

[SAVE THE COMMAND]

`git branch -d <branch_name>`

Delete a local branch

Go to [Follow Along 8-4](#) to carry out the branch deletion on behalf of your friend. Note that you cannot be on a branch when you delete it, so you (acting as your friend) will have to switch from the `feature` branch to the `main` branch in the `friend-rainbow` repository to delete the local `feature` branch.

[FOLLOW ALONG 8-4]

1 friend-rainbow \$ **git branch --all**

```
* feature  
  main  
  
remotes/origin/HEAD -> origin/main  
  
remotes/origin/feature  
  
remotes/origin/main
```

2 friend-rainbow \$ **git push origin -d feature**

```
To github.com:gitlearningjourney/rainbow-remote.git  
- [deleted] feature
```

3 Go to the `rainbow-remote` repository on your hosting service, refresh the page, and look at the list of branches. The `feature` branch should no longer be there.

4 friend-rainbow \$ **git branch --all**

```
* feature  
  main  
  
remotes/origin/HEAD -> origin/main  
  
remotes/origin/main
```

5 friend-rainbow \$ **git switch main**

```
Switched to branch 'main'  
  
Your branch is up to date with 'origin/main'.
```

6 friend-rainbow \$ **git branch -d feature**

```
Deleted branch feature (was fc8139c).
```

7 friend-rainbow \$ **git branch --all**

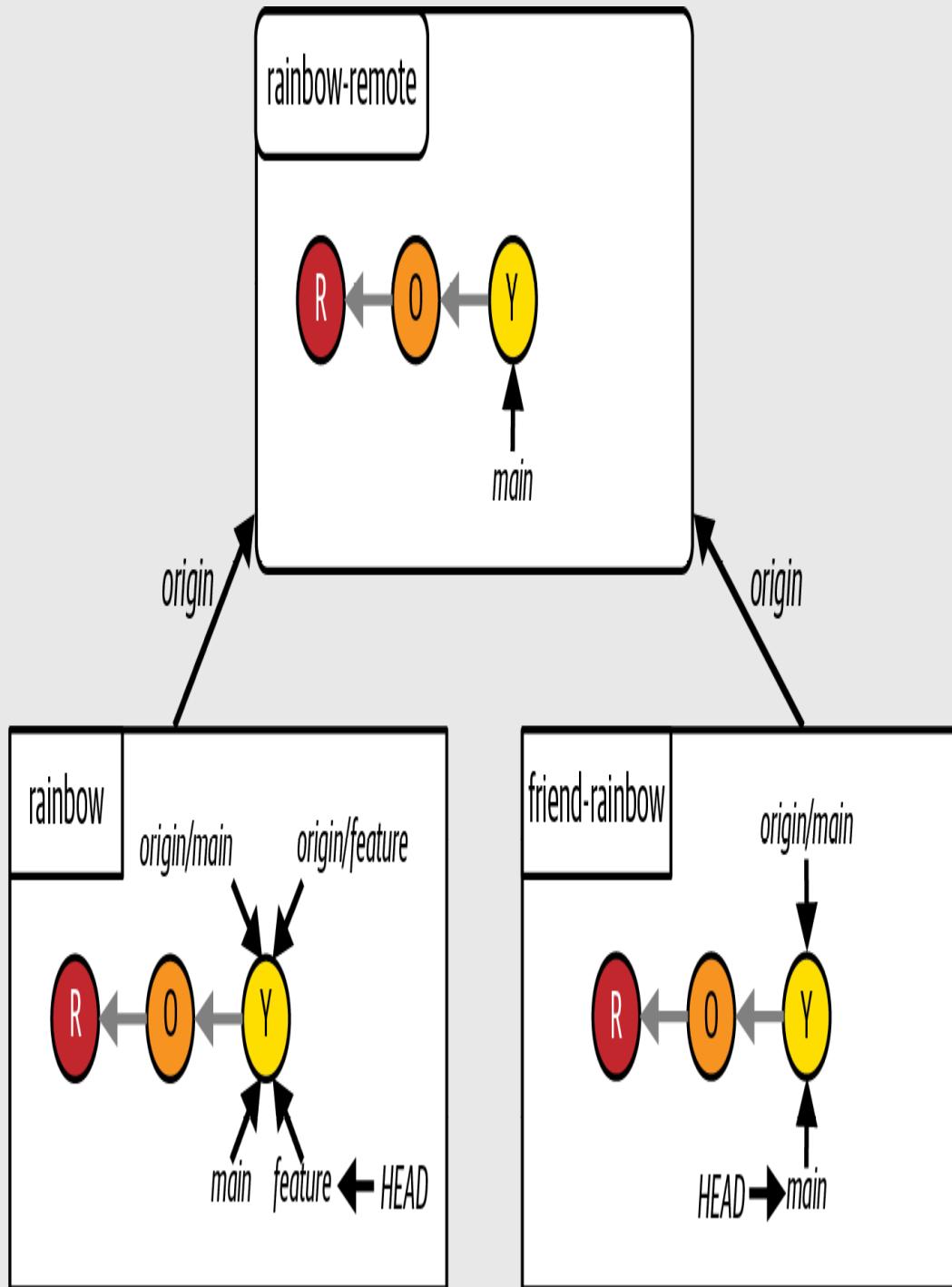
```
* main  
  
remotes/origin/HEAD -> origin/main  
  
remotes/origin/main
```

What to notice:

- In step 2, you deleted the remote `feature` branch and the `origin/feature` remote-tracking branch.
- In step 3, you can see there is no remote `feature` branch in the `rainbow-remote` repository.
- In step 6, you deleted the local `feature` branch.
- In step 7, the `git branch --all` output indicates that there is no local `feature` branch in the `friend-rainbow` repository.

[Visualize it 8-4](#) illustrates the state of the local and remote repositories after making these changes.

[VISUALIZE IT 8-4]



The Rainbow project after your friend deletes the remote **feature** branch, the **origin/feature** remote-tracking branch, and the local **feature** branch

As you can see, in the `rainbow` repository there still is a `feature` branch and an `origin/feature` remote-tracking branch. The fact that your friend deleted the `feature` and `origin/feature` branches in their `friend-rainbow` repository does not affect your `rainbow` repository. You'll delete these branches in the `rainbow` repository later in this chapter, but for now, let's learn about how the collaboration works when your friend starts contributing to the Rainbow project.

Git Collaboration and Branches

Now that you're going to start collaborating with others on the Rainbow project, you may want to discuss what some of your Git conventions should be. For example, in [Chapter 3](#) I mentioned that some teams may have rules about what to include in a commit message, and in [Chapter 4](#) you learned that some teams may also have conventions about how they name their branches.

Teams also may have rules about how to manage branches. For example, there may be conventions about:

- Which branches are allowed to be merged into one another.
- When branches need to be created.
- What the review process for work on a branch should be.

An example of a rule that you may encounter in a Git project is that individuals should work only on their own topic branches and avoid working on other people's topic branches. This helps avoid merge conflicts (which we will cover in [Chapter 10](#)). We won't apply this rule to the Rainbow project in this book, to keep things simple and allow you to focus on the new concepts you are learning in each chapter rather than branch

management. However, keep in mind that you may set or encounter different branching rules in future Git projects that you work on. It's important to communicate with your collaborators about branching conventions and to have a good understanding of what branches are and how working in local and remote repositories with other people affects branches.

Now, let's walk through some examples to see how the collaboration on the Rainbow project works.

MAKING A COMMIT IN THE LOCAL REPOSITORY

In this section, your friend is going to add a note about the color green to the `rainbowcolors.txt` file and make a commit on the `main` branch of their local repository. Remember that you're acting as your friend now, which means you'll be working in their project directory, `friend-rainbow`. As discussed earlier, you should have this project directory open in a separate text editor window.

Go to [Follow Along 8-5](#) to simulate your friend making a commit.

[NOTE]

In the `git log` output of step 4 in Follow Along 8-5, you are the author of both the yellow commit and the green commit. This is because you are simulating that you are your friend contributing to the Rainbow project. However, in reality, if someone else had made the green commit, then they would be the author of the commit, and their name and email address would be displayed in the output. Keep this in mind for the rest of the chapters in this book.

[FOLLOW ALONG 8-5]

1 Open the `rainbowcolors.txt` file in the `friend-rainbow` project directory in your text editor, add “Green is the fourth color of the rainbow.” on line 4, and save the file.

2 friend-rainbow \$ `git add rainbowcolors.txt`

3 friend-rainbow \$ `git commit -m "green"`

```
[main 6987cd2] green  
 1 file changed, 2 insertions(+), 1 deletion(-)
```

4 friend-rainbow \$ `git log`

```
commit 6987cd2996e245ec24ee9c5ea99874f0a01a31cd (HEAD -> main)
```

```
Author: annaskoulikari <gitlearningjourney@gmail.com>
```

```
Date: Sat Feb 19 11:49:03 2022 +0100
```

```
green
```

```
commit fc8139cbf8442cdbb5e469285abaac6de919ace6 (origin/main,  
origin/HEAD)
```

```
Author: annaskoulikari <gitlearningjourney@gmail.com>
```

```
Date: Sat Feb 19 10:09:59 2022 +0100
```

```
yellow
```

5 Go to the `rainbow-remote` repository on your hosting service and refresh the page. The green commit won’t be there yet.

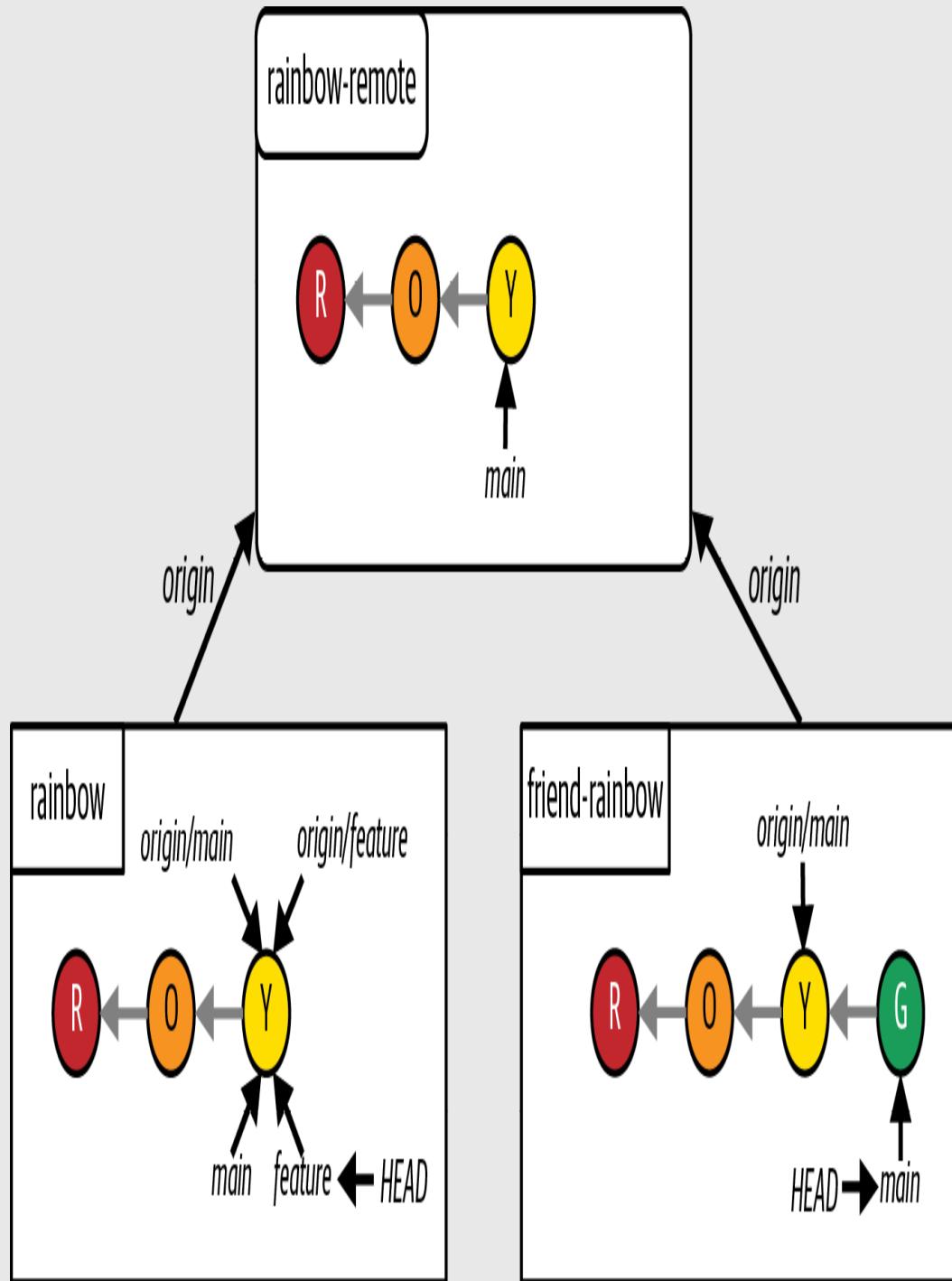
What to notice:

- In step 4, the `git log` output indicates that in the `friend-rainbow` repository:

- The local `main` branch has updated to point to the green commit.
- The `origin/main` remote-tracking branch still points to the yellow commit.

[Visualize it 8-5](#) illustrates the changes in the `friend-rainbow` repository.

[VISUALIZE IT 8-5]



The Rainbow project after your friend makes the green commit on the `main` branch in the `friend-rainbow` repository

As you can see, the green commit does not yet appear in the `rainbow-remote` repository. This is because remote repositories do not update automatically; work done in local repositories needs to be explicitly pushed to them.

Notice also that `origin/main` remote-tracking branch still points to the yellow commit. This is because, as mentioned in [Chapter 7](#), remote-tracking branches represent the state of remote branches. Since your friend has not yet pushed their changes to the remote repository, the remote `main` branch has not updated, and therefore the `origin/main` remote-tracking branch has not updated either.

At this point there is no way for you to have the green commit in the `rainbow` repository, because updating one local repository with the changes made in another local repository is a two-step process. First, the local repository with the changes has to explicitly push those changes to a remote repository. Then, the local repository without the changes has to explicitly fetch and integrate the changes from the remote repository.

Next, let's see how your friend will push the work they've done to the remote repository.

PUSHING TO THE REMOTE REPOSITORY

In [Chapter 7](#), you learned that when you push work from a local branch to a remote branch, Git needs some way to know which remote branch you want to push to. If the local branch has an upstream branch defined for it, you can use `git push` with no arguments and Git will automatically push the work to that branch. However, if no upstream branch is defined for the local branch you're working on, you'll need to specify which remote branch to push to when you enter the `git push` command.

The last time you used the `git push` command in the `rainbow` repository, you passed in the shortname and the branch name. This was because you had not defined an upstream branch for your local branch. Recall that an upstream branch is the remote branch that a particular local branch tracks. When you clone a repository, upstream branches are *automatically* set up for the branches that exist in the cloned repository.

The command to use to see whether upstream branches are defined is `git branch` with the `-vv` option (which stands for “very verbose”). This command also tells you whether a local branch is ahead of or behind an upstream branch, if it is defined.

[SAVE THE COMMAND]

`git branch -vv`

List the local branches and their upstream branches, if they have any

There are two main benefits to defining upstream branches:

1. If you have fetched (downloaded) the remote repository commits, you can check if your repository is ahead of or behind the remote repository by using either the `git branch -vv` command or the `git status` command. (We will cover the process of fetching in the next section in this chapter.)
2. You can simplify the commands you use with the remote repository because you don’t need to specify the branch name and remote repository shortname. For example, you can use the `git push` command on its own without passing any arguments.

We will observe both of these benefits in the upcoming Follow Alongs.

Since your friend cloned their repository, the upstream branch for their local `main` branch is already set to be the remote `main` branch. Go to [Follow Along 8-6](#) to see how your friend can check the status of the remote repository.

[FOLLOW ALONG 8-6]

```
1 friend-rainbow $ git branch -vv
* main 6987cd2 [origin/main]: ahead 1] green

2 friend-rainbow $ git status
on branch main

Your branch is ahead of 'origin/main' by 1 commit.

(use "git push" to publish your local commits)

nothing to commit, working tree clean
```

What to notice:

- In step 1, the `git branch -vv` output shows that the upstream branch set up for the local `main` branch is the remote `main` branch in the remote repository with shortname `origin` and that the local `main` branch is ahead by one commit.
- In step 2, the `git status` output also indicates that the local `main` branch is ahead of the `origin/main` upstream branch by one commit.

Next, go to [Follow Along 8-7](#) to see how your friend can simply use the `git push` command without any arguments to update the remote repository.

[FOLLOW ALONG 8-7]

1 friend-rainbow \$ **git push**
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 295 bytes | 295.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:gitlearningjourney/rainbow-remote.git
 fc8139c..6987cd2 main -> main

2 friend-rainbow \$ **git status**
on branch main
Your branch is up to date with 'origin/main'.
nothing to commit, working tree clean

3 friend-rainbow \$ **git log**
commit 6987cd2996e245ec24ee9c5ea99874f0a01a31cd (**HEAD -> main**,
origin/main, origin/HEAD)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date: Sat Feb 19 11:49:03 2022 +0100
green
commit fc8139cbf8442cdbb5e469285abaac6de919ace6
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date: Sat Feb 19 10:09:59 2022 +0100
yellow

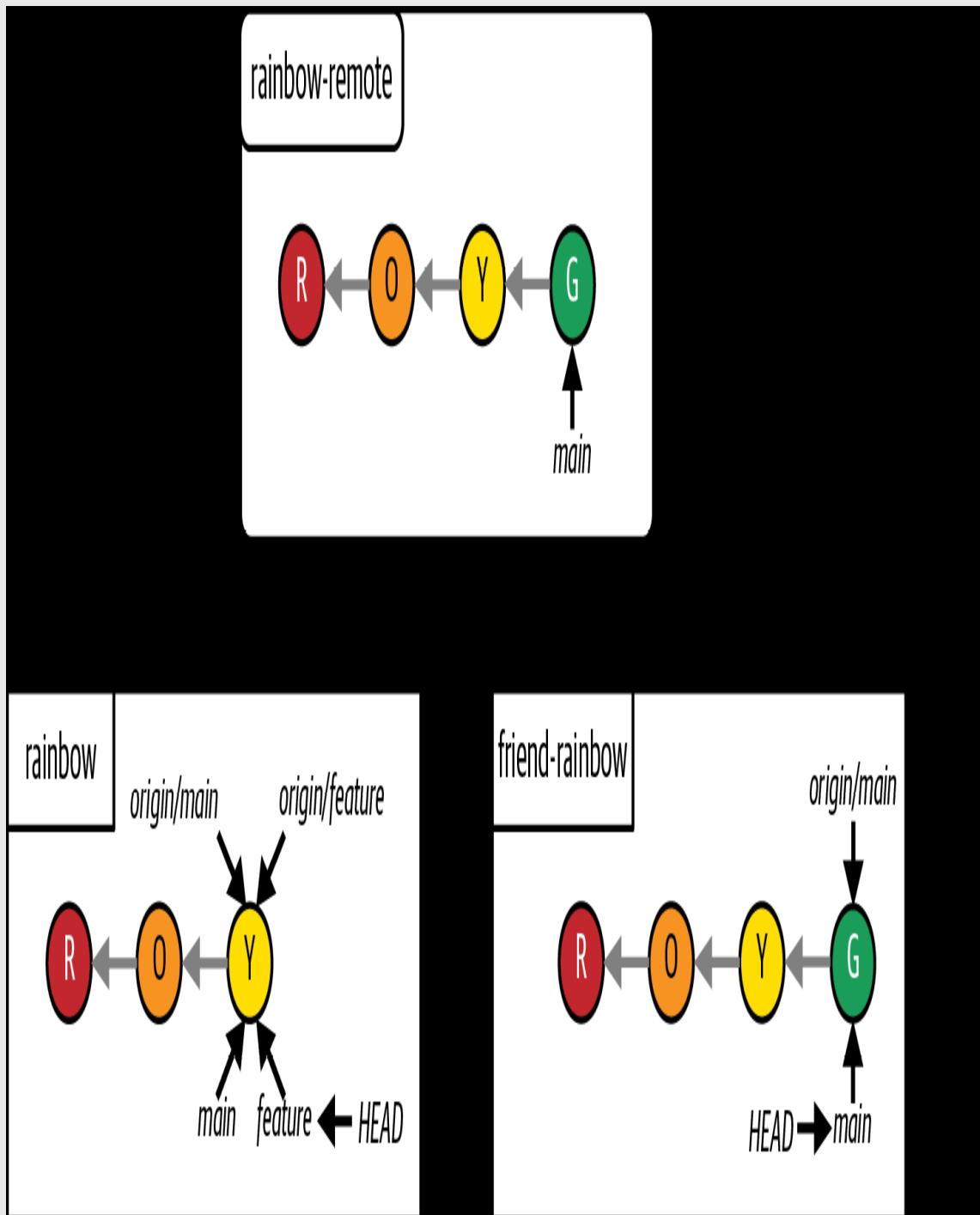
4 Go to the **rainbow-remote** repository on your hosting service and refresh the page. The green commit should now be there.

What to notice:

- In step 3, the `git log` output indicates that in the `friend-rainbow` repository, the `origin/main` remote-tracking branch has updated to point to the green commit.
- In step 4, you can see that in the `rainbow-remote` repository, the remote `main` branch has been updated to point to the green commit.

[Visualize it 8-6](#) illustrates these observations.

[VISUALIZE IT 8-6]



The Rainbow project after your friend pushes the green commit to the remote repository

Notice that the `rainbow` repository does not yet have the green commit, and that the local `main` branch and `origin/main` remote-tracking branch in this repository still point to the yellow commit.

Next, you want to make sure that the local `main` branch in your `rainbow` repository is in sync with the `main` branch in the `rainbow-remote` repository and the `friend-rainbow` repository. For that, you'll need to learn about fetching.

Incorporating Changes from the Remote Repository

The reason the local `main` branch and `origin/main` remote-tracking branch in the `rainbow` repository still point to the yellow commit is because local repositories do not automatically update with new data from remote repositories. Just as you need to take explicit actions to update a remote repository with changes in a local repository, you need to take explicit actions to update local branches and remote-tracking branches with changes from a remote repository.

Incorporating changes from a remote branch into a local branch is a two-step process: first you fetch the changes from the remote repository, then you integrate those changes into the local branch in the local repository. Let's start by exploring the first step.

FETCHING CHANGES FROM THE REMOTE REPOSITORY

In Git, we use the term *fetch* or *fetching* to refer to the process of downloading data from a remote repository to a local repository, and the command we use to do this is `git fetch`. The `git fetch` command downloads all the necessary commits to update all the remote-tracking branches in the

local repository to reflect the state of the remote branches in the remote repository that is specified. When no remote repository shortname is provided as an argument to the `git fetch` command, by default the remote repository with shortname `origin` will be used, unless there's an upstream branch defined for the current branch.

[SAVE THE COMMAND]

`git fetch <shortname>`

Download data from the `<shortname>` remote repository

`git fetch`

Download data from the remote repository with shortname `origin`

The `git fetch` command affects only remote-tracking branches. It does not affect local branches. In other words, it only fetches (downloads) data; it doesn't actually integrate the data into any local branches. Thus, nothing in your working directory will change when you fetch data from a remote repository.

You'll integrate the changes in the next section; for now, go to [Follow Along 8-8](#) to fetch the data from the remote repository.

[FOLLOW ALONG 8-8]

1

Go to the `rainbow` project directory in your command line window.

2

```
rainbow $ git log --all
commit fc8139cbf8442cdbb5e469285abaac6de919ace6 (HEAD -> feature,
origin/main, origin/feature, main)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 10:09:59 2022 +0100

yellow

commit 7acb333f08e12020efb5c6b563b285040c9dba93
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 09:42:07 2022 +0100

orange
```

3

```
rainbow $ git fetch
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (3/3), 275 bytes | 91.00 KiB/s, done.
From github.com:gitlearningjourney/rainbow-remote
fc8139c..6987cd2  main        -> origin/main
```

[FOLLOW ALONG 8-8]

4

```
rainbow $ git log --all
commit 6987cd2996e245ec24ee9c5ea99874f0a01a31cd (origin/main)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 11:49:03 2022 +0100

    green

commit fc8139cbf8442cdbb5e469285abaac6de919ace6 (HEAD -> feature,
origin/feature, main)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 10:09:59 2022 +0100

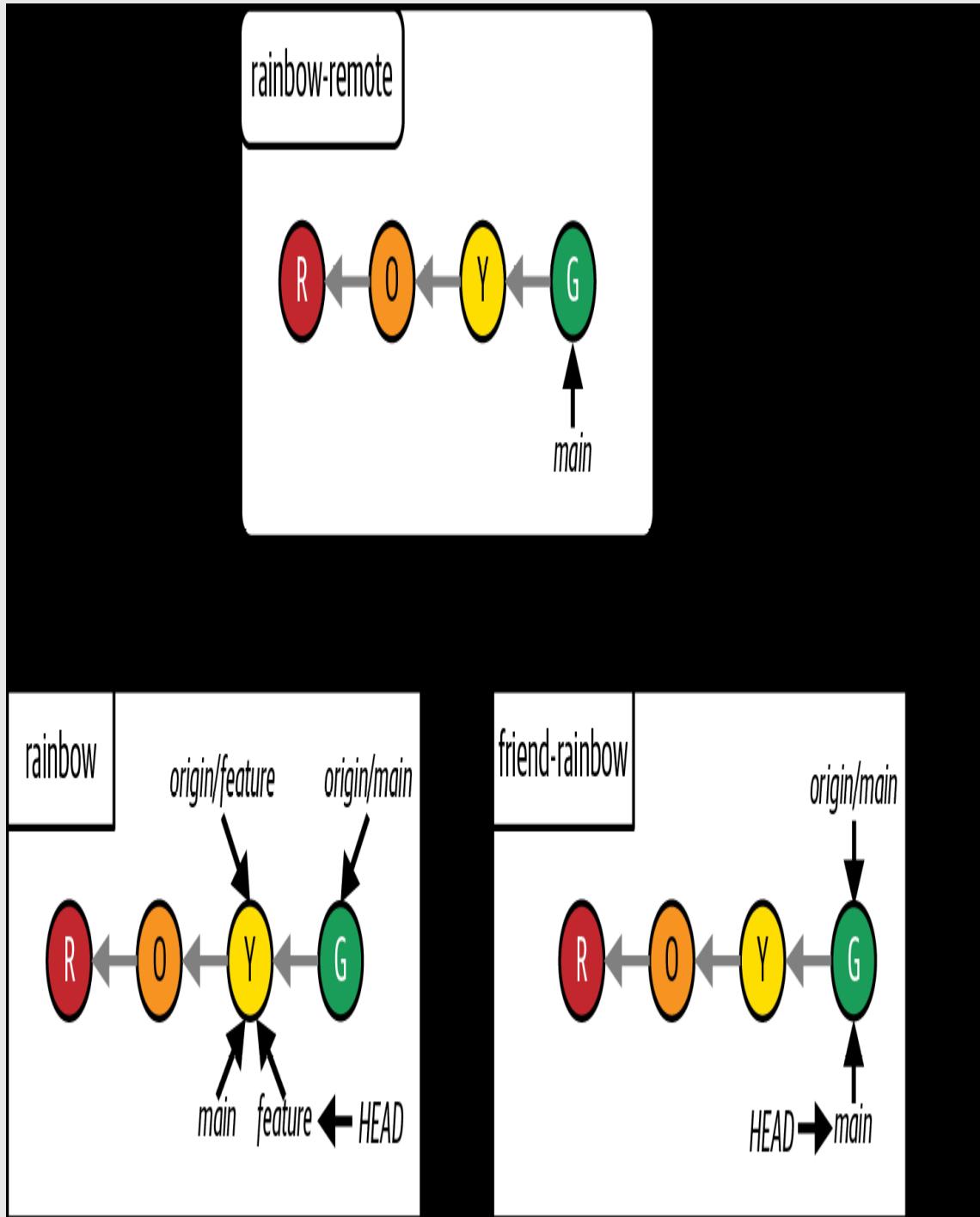
    yellow
```

What to notice:

- In step 2, the `git log` output shows that:
 - The local `main` branch and the `origin/main` remote-tracking branch are pointing to the yellow commit.
- In step 4, the `git log` output shows that:
 - The `origin/main` remote-tracking branch is pointing to the green commit.
 - The local `main` branch is still pointing to the yellow commit.

[Visualize it 8-7](#) illustrates these observations.

[VISUALIZE IT 8-7]



The Rainbow project after you fetch the data from the `rainbow-remote` repository to the `rainbow` repository

You just saw how the `git fetch` command updated the remote-tracking branches in the local repository for any remote branches that exist in the remote repository. Now, you are ready for the second step in the process: incorporating the changes from the remote repository into a local branch.

INTEGRATING CHANGES INTO A LOCAL BRANCH

Once you have fetched the changes from a remote repository and updated the remote-tracking branches in a local repository, you’re ready to update a local branch. Recall that in [Chapter 5](#), I mentioned that Git provides two ways to integrate changes: *merging* and *rebasing*. We will cover rebasing in [Chapter 11](#). For now, you will continue using merging.

In [“Types of Merges” on page 64](#), you learned about two kinds of merges: fast-forward merges and three-way merges. In the fast-forward merge you carried out in [Chapter 5](#), you merged a branch called `feature` into `main`. Both `feature` and `main` were local branches in the `rainbow` repository. In this section, you are going to merge the `origin/main` remote-tracking branch into the local `main` branch in the `rainbow` repository. The types of branches involved in the merge are different, but the process of executing the merge is the same. This will once again be a fast-forward merge.

As you learned in [Chapter 5](#), when you execute a merge you must be on the branch you’re merging into, which in this case will be the `main` branch in the `rainbow` repository. Therefore, in [Follow Along 8-9](#), you will first switch onto the `main` branch before executing the merge.

To integrate the commits that were on the `main` branch in the remote repository into your local `main` branch, you will use the `git merge` command. This time you will pass in the name of a remote-tracking branch, `origin/main`.

Go to [Follow Along 8-9](#) to execute the merge.

[FOLLOW ALONG 8-9]

1 rainbow \$ **git switch main**

Switched to branch 'main'

2 rainbow \$ **git merge origin/main**

Updating fc8139c..6987cd2

Fast-forward

rainbowcolors.txt | 3 +-

1 file changed, 2 insertions(+), 1 deletion(-)

3 rainbow \$ **git log**

commit 6987cd2996e245ec24ee9c5ea99874f0a01a31cd (**HEAD -> main**,
origin/main)

Author: annaskoulikari <gitlearningjourney@gmail.com>

Date: Sat Feb 19 11:49:03 2022 +0100

green

commit fc8139cbf8442cdbb5e469285abaac6de919ace6 (**origin/feature**,
feature)

Author: annaskoulikari <gitlearningjourney@gmail.com>

Date: Sat Feb 19 10:09:59 2022 +0100

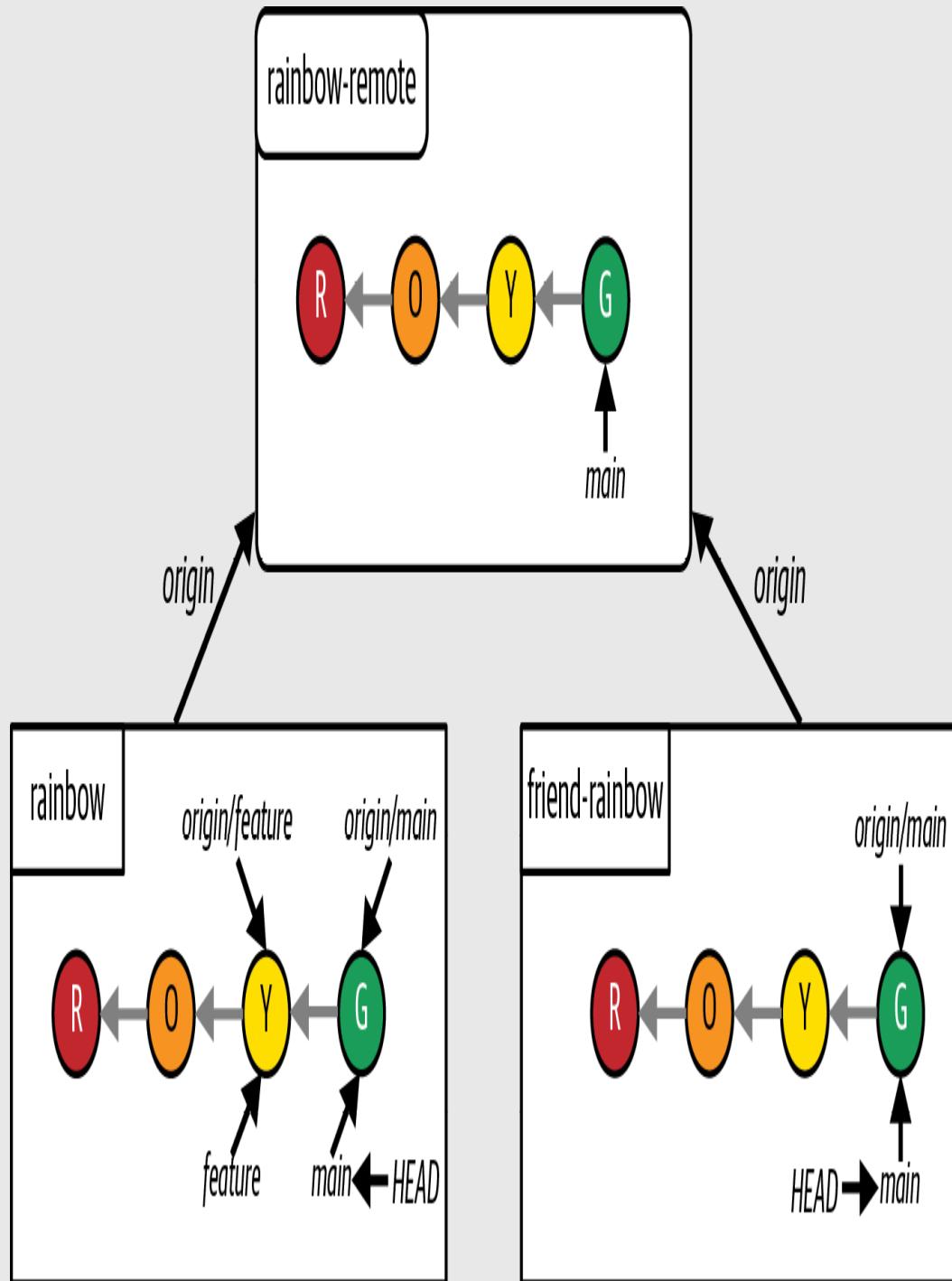
yellow

What to notice:

- In step 3, the `git log` output indicates that the local `main` branch points to the green commit.

This is illustrated in [Visualize it 8-8.](#)

[VISUALIZE IT 8-8]



The Rainbow project after you merge the `origin/main` remote-tracking branch into the local `main` branch in the `rainbow` repository

You have just observed how the local branch updates once you complete the merge. In [Visualize it 8-8](#), you can also see that in the `rainbow` repository you still have the `feature` branch. Next, let's carry out some more branch deletions to declutter the `rainbow` repository.

Deleting Branches (Continued)

In the `rainbow` repository, you still have the local `feature` branch and the `origin/feature` remote-tracking branch. For simplicity, moving forward you will work on the `main` branch, so you can now delete those branches.

In [“Deleting Branches” on page 122](#), you used the `git push <shortname> -d <branch_name>` command to delete the remote `feature` branch and the `origin/feature` remote-tracking branch in the `friend-rainbow` repository in one go. Now there is no remote `feature` branch any more, so to delete the `origin/feature` remote-tracking branch in the `rainbow` repository you can use the `git fetch` command with the `-p` option (which stands for “prune”). This command with this option will delete any remote-tracking branches that correspond to remote branches that have been deleted in the remote repository.

[SAVE THE COMMAND]

`git fetch -p`

Remove remote-tracking branches that correspond to deleted remote branches and download data from the remote repository

To delete the local `feature` branch, you will use the same command you used earlier in this chapter. Go to [Follow Along 8-10](#) to carry out the branch deletions.

[FOLLOW ALONG 8-10]

```
1 rainbow $ git branch --all
      feature
      * main
      remotes/origin/feature
      remotes/origin/main

2 rainbow $ git fetch -p
From github.com:gitlearningjourney/rainbow-remote
  - [deleted]          (none)    -> origin/feature

3 rainbow $ git branch --all
      feature
      * main
      remotes/origin/main

4 rainbow $ git branch -d feature
Deleted branch feature (was fc8139c).

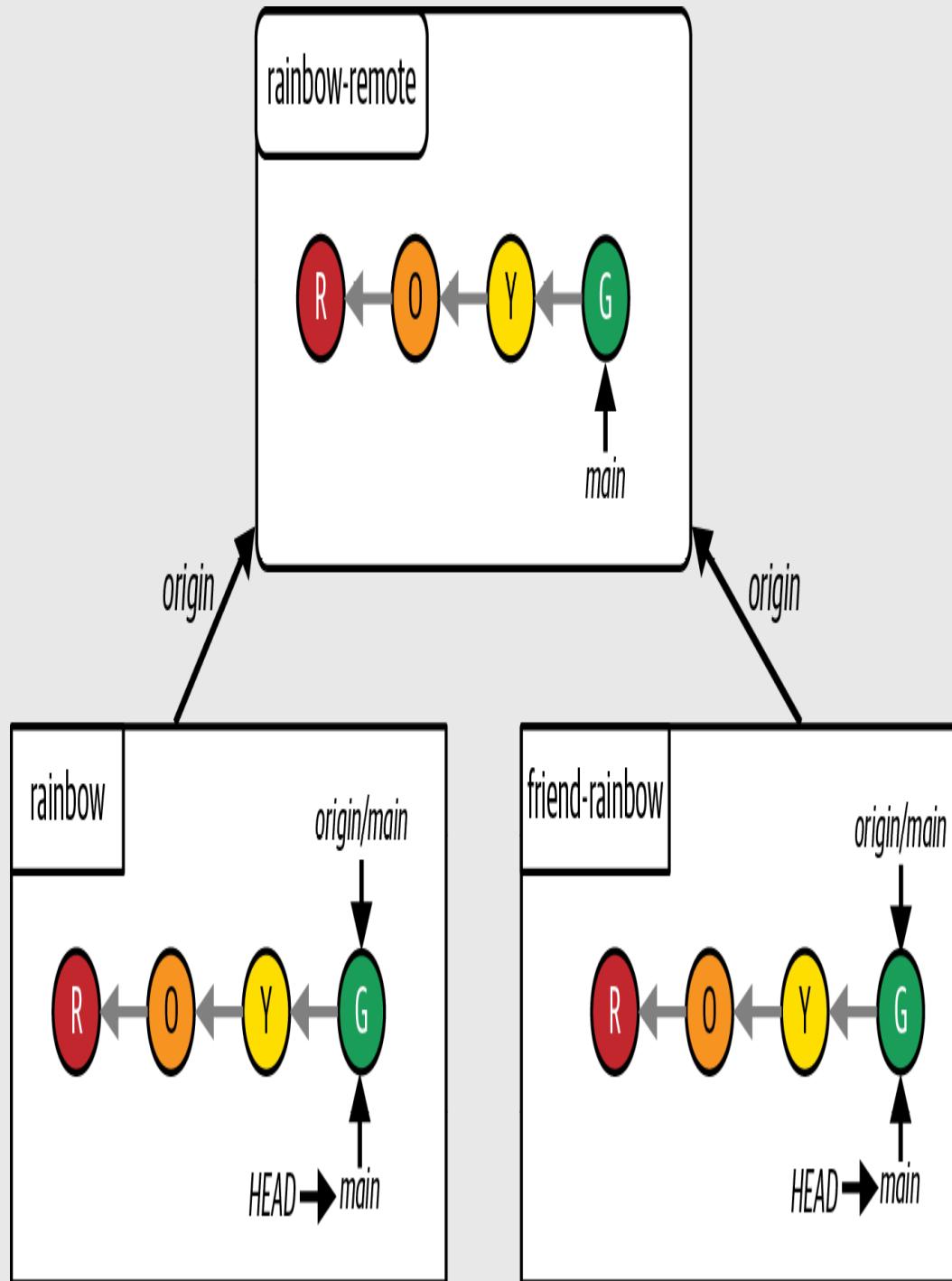
5 rainbow $ git branch --all
      * main
      remotes/origin/main
```

What to notice:

- In step 2, you deleted the `origin/feature` remote-tracking branch.
- In step 4, you deleted the local `feature` branch.
- In step 5, the `git branch --all` output indicates that you no longer have a local `feature` branch or an `origin/feature` remote-tracking branch in the `rainbow` repository.

This is illustrated in [Visualize it 8-9](#).

[VISUALIZE IT 8-9]



The Rainbow project after you delete the local `feature` branch and the `origin/feature` remote-tracking branch in the `rainbow` repository

Summary

In this chapter, you started simulating what it would be like to work with other people on a Git project. Your “friend” cloned the `rainbow-remote` repository and created a second local repository, which you called `friend-rainbow`. Through this process, you observed that `origin` is the default shortname that Git associates with a remote repository when it is cloned.

After your friend made the green commit in the `friend-rainbow` repository and pushed it to the remote repository, you learned about the process of updating a local branch with changes from a remote branch in a remote repository, which involves fetching the changes from the remote repository and then integrating them. In the `rainbow` repository, you integrated the changes by carrying out a fast-forward merge. Next, in [Chapter 9](#), you will learn more about the other kind of merge mentioned in [Chapter 5](#): the three-way merge.

Three-Way Merges

In [Chapter 8](#), you learned about cloning and what it's like to collaborate with another person on a Git project. You walked through an example of a friend contributing work to the Rainbow project, pushing the work to the remote repository, and you fetching and merging the work in order to keep your repositories in sync.

All of the merges you have done up until this point in the Rainbow project have been fast-forward merges. In this chapter, you will learn how to carry out a three-way merge. In the process, you will also go over an example of defining upstream branches and you will learn about what happens when you edit files multiple times in your working directory between commits. Finally, you will learn about pulling from a remote repository, and we will discuss how pulling is different from fetching.

State of the Local and Remote Repositories

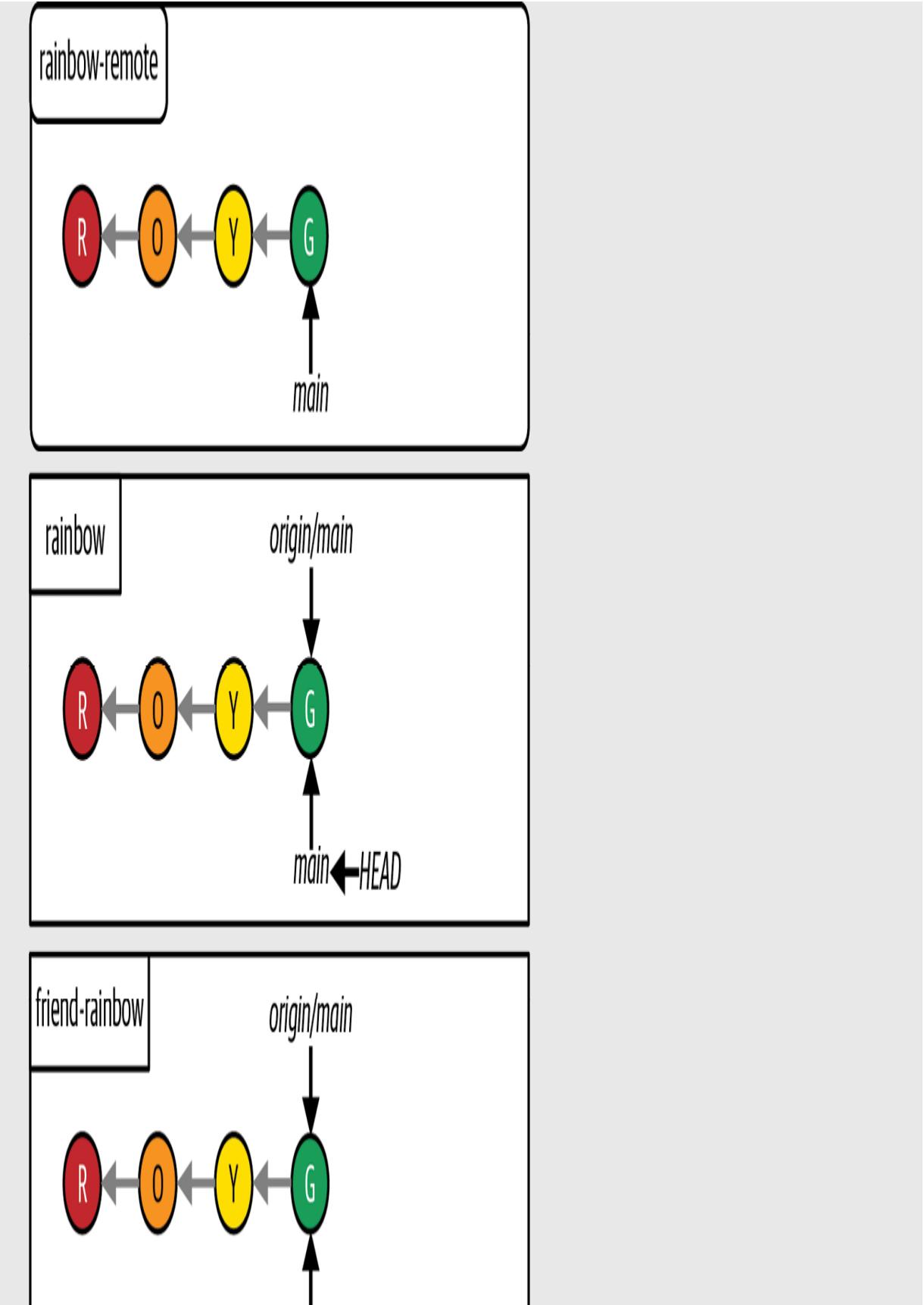
At the start of this chapter, you should have two local repositories called `rainbow` and `friend-rainbow` and one remote repository called `rainbow-remote`. All three of these repositories should be in sync; in other words, they should contain the same commits and branches. I recommend that you continue

using two separate text editor windows and command line windows for the `rainbow` repository and the `friend-rainbow` repository as you work through the examples in this chapter, as discussed in “[The Collaboration Simulation](#)” on [page 115](#).

[Visualize it 9-1](#) shows the state of the local and remote repositories in the Rainbow project with all the commits that were made from [Chapter 1](#) through [Chapter 8](#).

To focus on the commits you are going to make in this chapter, from here on I will simplify the Visualize It diagrams and show only the last two commits that are part of the `main` branch in all the repositories, which are the yellow commit and the green commit. This representation is shown in [Visualize it 9-2](#).

[VISUALIZE IT 9-1]

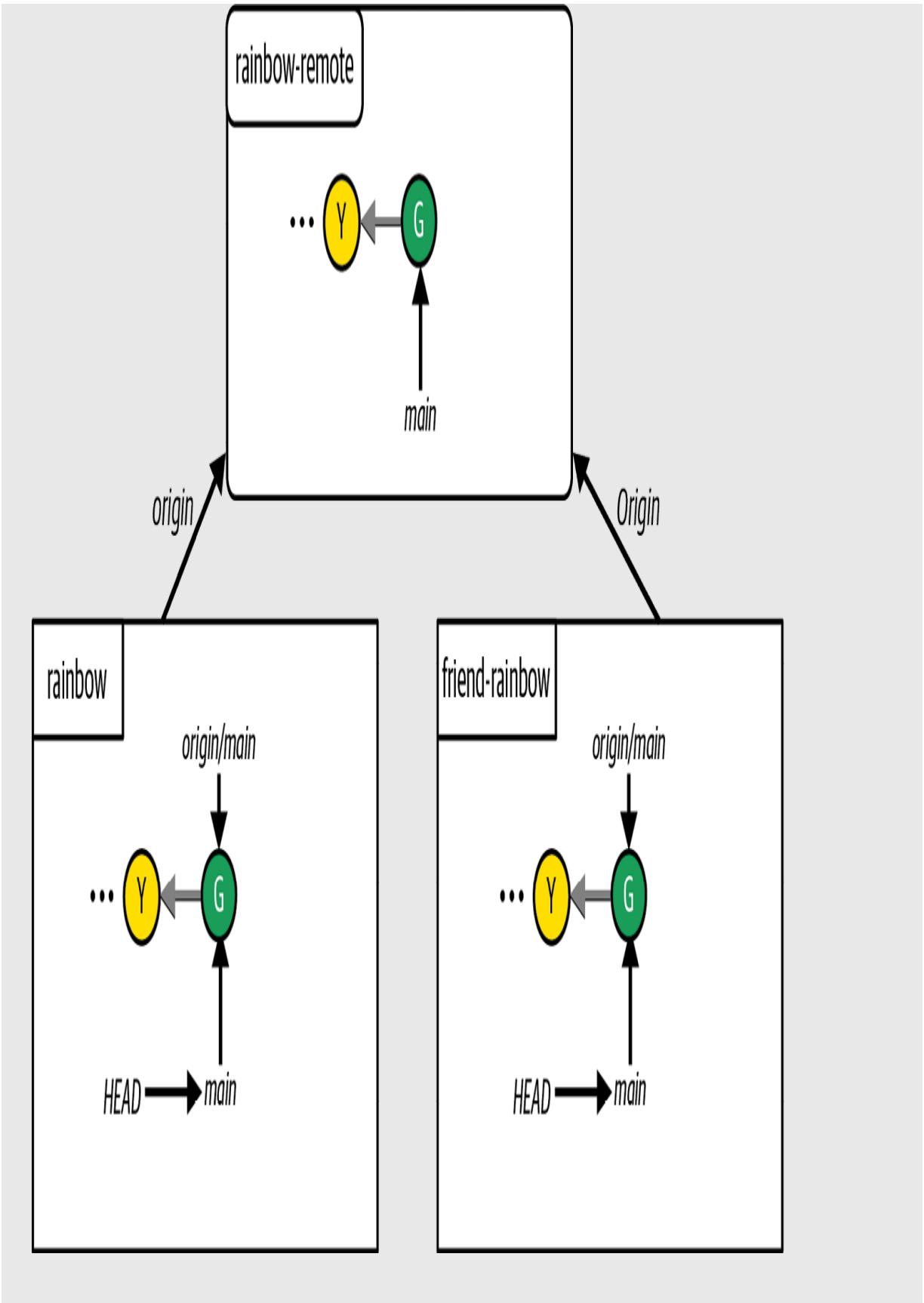




main ↪ HEAD

The Rainbow project at the start of [Chapter 9](#) with all the commits since [Chapter 1](#)

[VISUALIZE IT 9-2]



A simplified representation of the Rainbow project at the start of [Chapter 9](#), showing just the last two commits on the `main` branch in all the repositories

Why Are Three-Way Merges Important?

In [Chapter 5](#), I explained that merging involves integrating the changes made in one branch, called the source branch, into another branch, called the target branch. I also introduced two types of merges: fast-forward merges and three-way merges. Up until now, you've only carried out fast-forward merges, but it is important to learn about three-way merges as well because they are a regular part of a Git user's day-to-day activities.

Three-way merges are a bit more complicated than fast-forward merges because they create merge commits and they may lead to merge conflicts.

Merge conflicts arise when you merge two branches where different changes have been made to the same parts in the same file(s), or if in one branch a file was deleted that was edited in the other branch.

In [Chapter 10](#), we will discuss merge conflicts in more depth and go over an example of a three-way merge with a merge conflict. In the three-way merge example in this chapter, your friend will edit a different file than the one you will edit, and therefore you won't have any merge conflicts.

As you learned in [Chapter 5](#), three-way merges occur when the development histories of the branches involved in the merge have diverged—in other words, when it is *not* possible to reach the target branch by following the commit history (parents links) of the source branch. Let's explore an example of how this situation might arise in [Example Book Project 9-1](#).

Example Book Project 9-1

Suppose that while working on the Book project, my coauthor and I both decide to make a branch off the `main` branch at the same time, to work on different chapters. I make the `chapter_five` branch and my coauthor makes the `chapter_six` branch, as seen in [Figure 9-1](#).

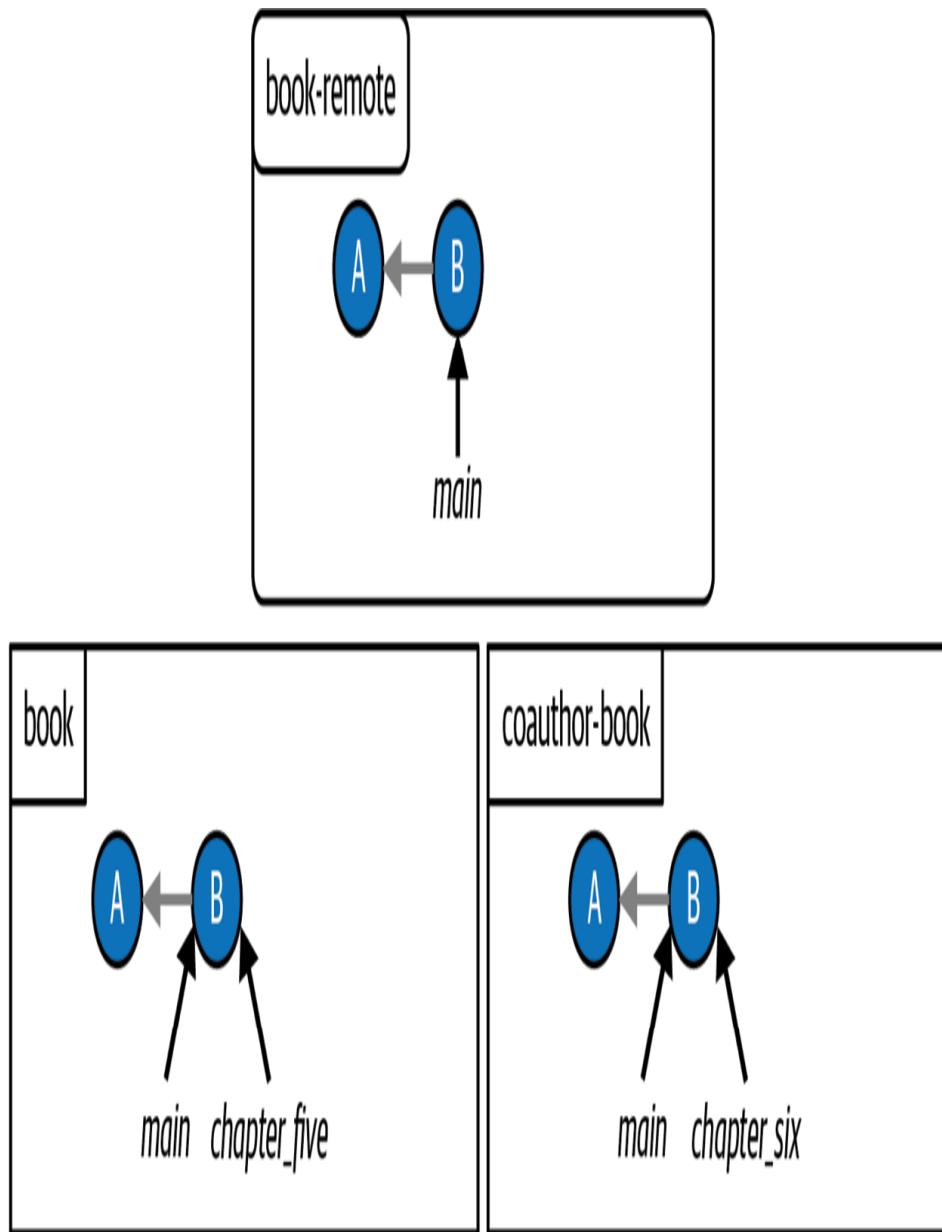


FIGURE 9-1

The Book project after my coauthor and I make branches to work on different chapters

We each work independently on our chapters. My coauthor finishes their work on the `chapter_six` branch first and proceeds to merge their work into the `main` branch and push the updated `main` branch to the remote repository. In [Figure 9-2](#), I represent the work my coauthor did as commit C and show the state of all the repositories.

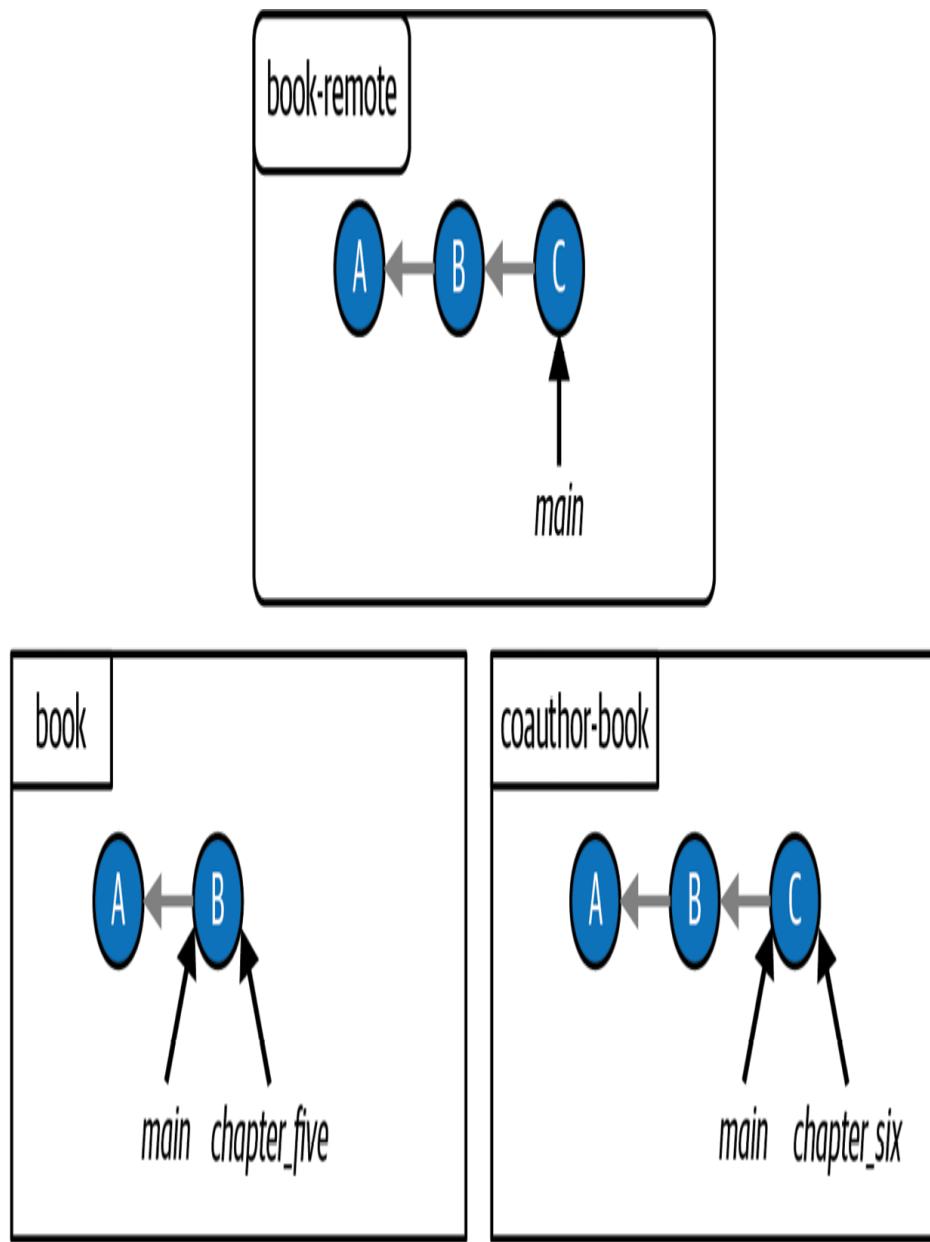


FIGURE 9-2

The Book project after my coauthor pushes their changes to the remote repository

When I finish working on the `chapter_five` branch, which I will represent as commit D, I also want to merge my work into the `main` branch. But my coauthor lets me know that they've already added work to the remote

`main` branch, so I first need to update my local `main` branch with the work that my coauthor added to the remote `main` branch, as seen in [Figure 9-3](#).

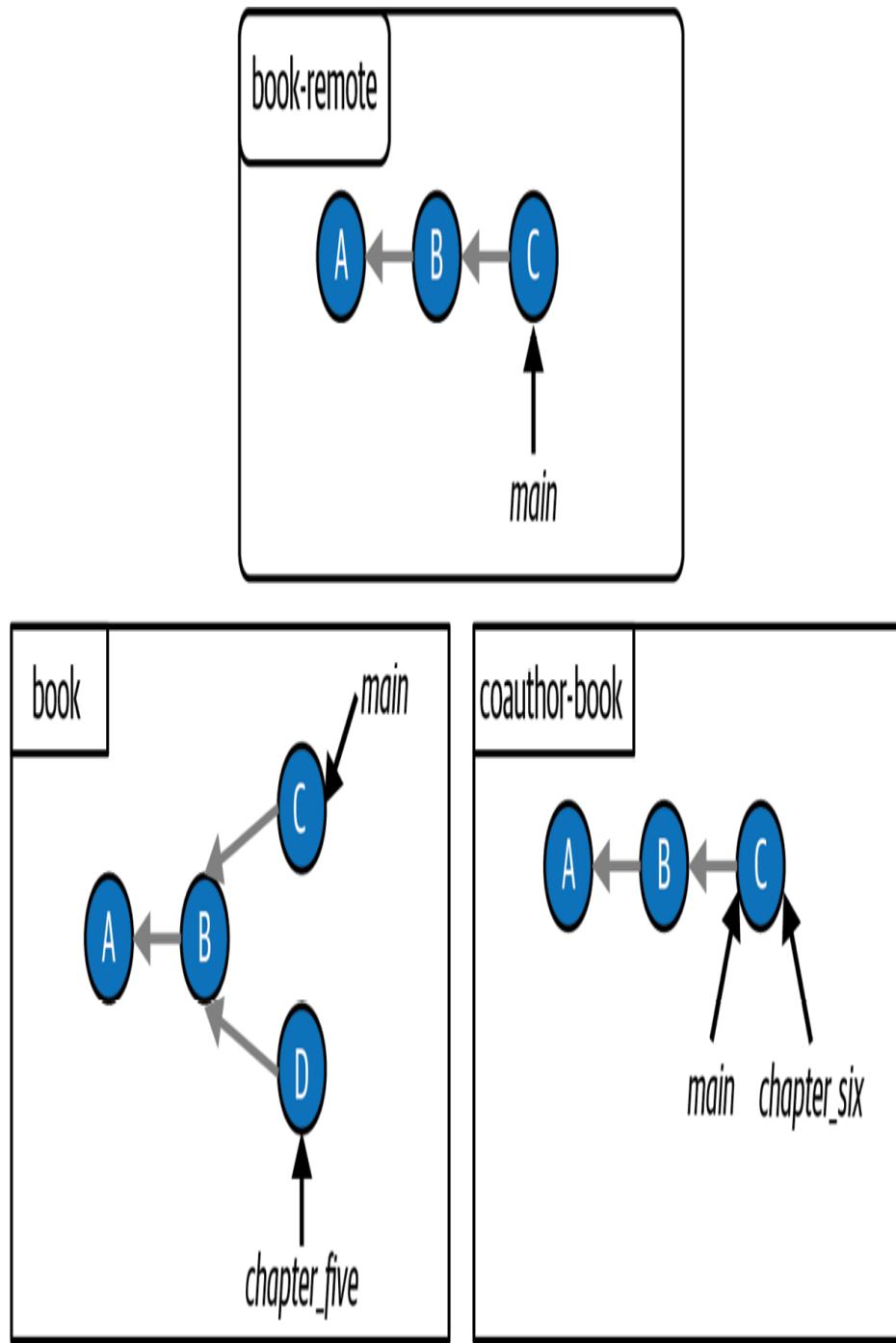


FIGURE 9-3

The Book project after I contribute work to my local `chapter_five` branch and fetch and

integrate the changes from the remote `main` branch

In [Figure 9-3](#), you can see that the development history of the local `main` branch in the `book` repository is made up of commits A, B, and C, while the development history of the `chapter_five` branch is made up of commits A, B, and D. Since it is not possible to follow the development history of the `chapter_five` branch to reach the `main` branch, this means the development histories of these branches have diverged. Next, I have a couple of options.

One option is to merge my local `chapter_five` branch into the local `main` branch, which will be a three-way merge, and then push the updated `main` branch to the remote repository.

The other option is to carry out a merge in the remote repository through a hosting service feature called a *pull request*. You'll learn about pull requests in [Chapter 12](#); for the purposes of this chapter, let's assume I decide to go ahead with the first option and carry out a three-way merge in my local repository.

[Figure 9-4](#) illustrates the state of the repositories when I integrate the `chapter_five` branch into `main` through a three-way merge. The merge commit produced by the three-way merge is represented as commit M.

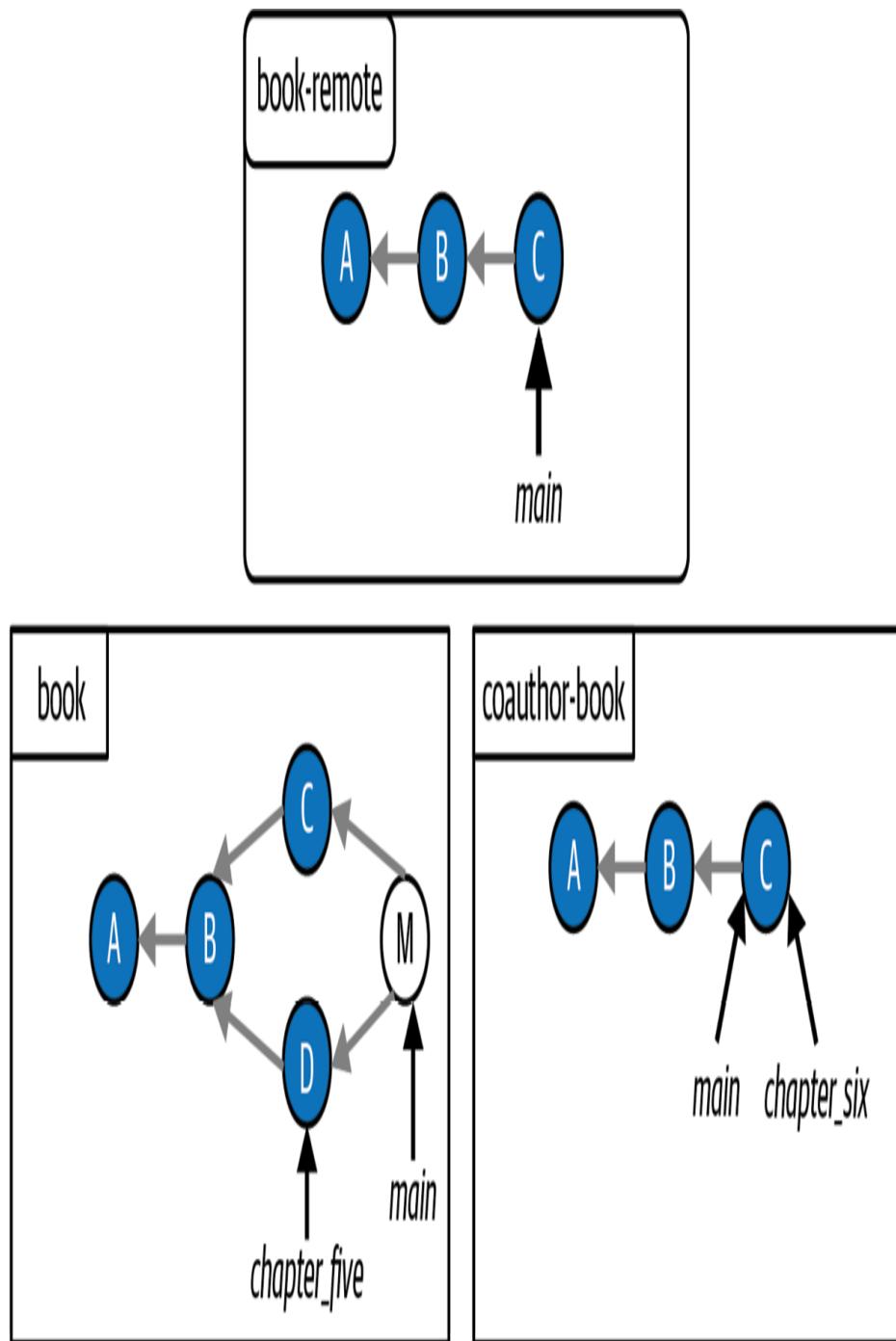


FIGURE 9-4

The Book project after I merge the `chapter_five` branch into the `main` branch through a three-way merge

As you can see in [Example Book Project 9-1](#), three-way merges produce merge commits, which are commits that can have more than one parent commit.

[NOTE]

Some people don't like three-way merges because they find that merge commits make the commit history more complicated. To avoid three-way merges you may use the process of rebasing, which you will learn about in [Chapter 11](#).

Next, you'll set up a situation in the Rainbow project where you will have to do a three-way merge. Along the way, you will learn about defining upstream branches and some characteristics of modified files in the working directory.

Setting Up a Three-Way Merge Scenario

First, go to [Follow Along 9-1](#) to start listing colors that are *not* part of the rainbow in a new file called `othercolors.txt` in the `rainbow` project directory.

[FOLLOW ALONG 9-1]

1 In the `rainbow` project directory in your text editor, create a new file called `othercolors.txt`. Type “Brown is not a color in the rainbow.” on line 1 in the file, and save it.

2 `rainbow $ git add othercolors.txt`

3 `rainbow $ git commit -m "brown"`

```
[main 7f0a87a] brown  
 1 file changed, 1 insertion(+)  
 create mode 100644 othercolors.txt
```

4 `rainbow $ git log`

```
commit 7f0a87a318e50638eec50a484bf8dfa76b76d08e (HEAD -> main)
```

```
Author: annaskoulikari <gitlearningjourney@gmail.com>
```

```
Date: Sat Feb 19 12:46:29 2022 +0100
```

```
brown
```

```
commit 6987cd2996e245ec24ee9c5ea99874f0a01a31cd (origin/main)
```

```
Author: annaskoulikari <gitlearningjourney@gmail.com>
```

```
Date: Sat Feb 19 11:49:03 2022 +0100
```

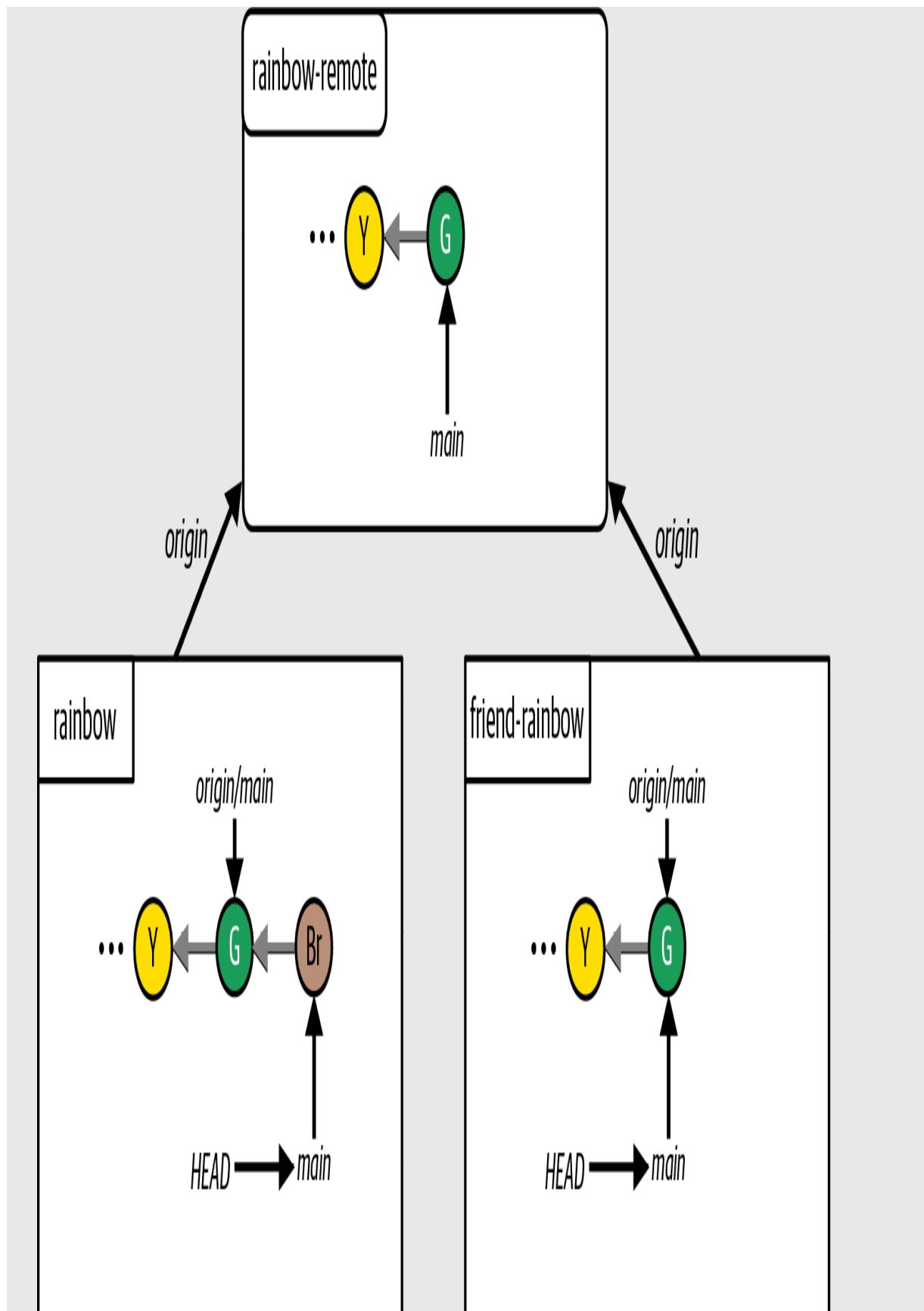
```
green
```

What to notice:

- You have made the brown commit in the `rainbow` repository.

This is illustrated in [Visualize it 9-3](#).

[VISUALIZE IT 9-3]



The Rainbow project after you make the brown commit in the `rainbow` repository

Next, before you push your commit to the remote repository, let's learn about defining upstream branches.

Defining Upstream Branches

In Chapters [7](#) and [8](#), I mentioned that when you push work from a local branch to a remote branch, Git needs a way to know which remote branch you want to push the work to. If no upstream branch is defined for the local branch you're working on, you'll need to specify which remote branch to push to when you enter the `git push` command. If a local branch has an upstream branch defined for it, you can use `git push` with no arguments and Git will automatically push the work to that branch.

You also learned that upstream branches are automatically set up when you clone a repository, but *not* when a repository is initialized locally. The `rainbow` repository was initialized locally, and you have not defined any upstream branches yet.

To avoid specifying the remote repository shortname and the branch every time you use the `git push` command on the `main` branch in your `rainbow` repository, you can define an upstream branch for the `main` branch and thereafter simply use the `git push` command with no arguments.

[NOTE]

Once an upstream branch has been defined for a local branch, you can also use other commands, such as `git pull`, without arguments. You will learn about the `git pull` command at the end of this chapter.

To set up the upstream branch you will use the `git branch` command with the `-u` option, which is short for `--set-upstream-to`. You'll pass in the name of the remote branch as an argument, specifying the remote repository shortname, a slash, and then the remote branch name (for example, `origin/main`.) Then, you will witness how you can use the `git push` command without any additional arguments.

[SAVE THE COMMAND]

`git branch -u <shortname>/<branch_name>`

Define an upstream branch for the current local branch

To check whether an upstream branch has been defined, you will use the `git branch -vv` command introduced in [Chapter 7](#).

Go to [Follow Along 9-2](#) to define the upstream branch for the local `main` branch.

[FOLLOW ALONG 9-2]

1 rainbow \$ **git branch -vv**

```
* main 7f0a87a brown
```

2 rainbow \$ **git branch -u origin/main**

```
branch 'main' set up to track 'origin/main'.
```

3 rainbow \$ **git branch -vv**

```
* main 7f0a87a [origin/main: ahead 1] brown
```

What to notice:

- In step 1, the `git branch -vv` output shows that there is no upstream branch set for the local `main` branch.
- In step 2, the command output explicitly states that an upstream branch has been set (`branch 'main' set up to track 'origin/main'`).
- In step 3, the `git branch -vv` output shows that the `main` branch from the remote repository with shortname `origin` has been set as the upstream branch for the local `main` branch.

Now that you have defined an upstream branch for your local `main` branch, go to [Follow Along 9-3](#) to use the `git push` command without any arguments.

[FOLLOW ALONG 9-3]

1

```
rainbow $ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 310 bytes | 310.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:gitlearningjourney/rainbow-remote.git
   6987cd2..7f0a87a main -> main
```

2

```
rainbow $ git log
commit 7f0a87a318e50638eec50a484bf8dfa76b76d08e (HEAD -> main,
origin/main)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 12:46:29 2022 +0100

brown

commit 6987cd2996e245ec24ee9c5ea99874f0a01a31cd
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 11:49:03 2022 +0100

green
```

3

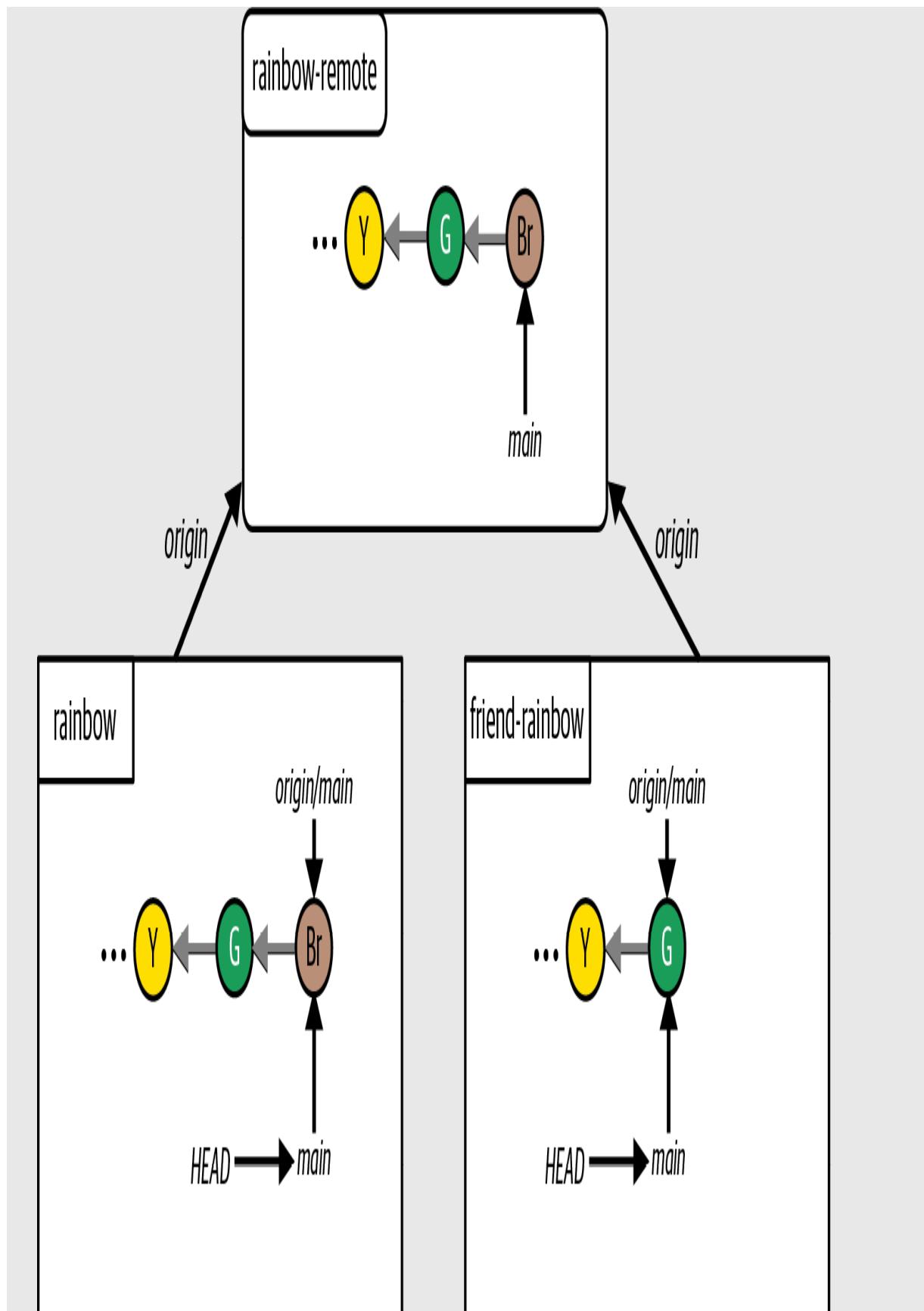
Go to the `rainbow-remote` repository on your hosting service and refresh the page. You should see the brown commit.

What to notice:

- You pushed your local `main` branch to the remote repository.

This is illustrated in [Visualize it 9-4](#).

[VISUALIZE IT 9-4]



The Rainbow project after you push your local `main` branch to the `rainbow-remote` repository

You have just learned how to define upstream branches in a local repository, and you have made the brown commit in the `rainbow` repository and pushed it to the remote repository. To end up in a situation where you will have to carry out a three-way merge, there must be divergent development histories between two branches.

Next, your friend will continue working on the local `main` branch in their local repository *without* fetching the changes you pushed to the remote `main` branch, which will cause the local `main` branch in the `friend-rainbow` repository and the `main` branch in the `rainbow-remote` repository to diverge.

While your friend is working on the local `main` branch, you are also going to learn about some characteristics of modified files in the working directory and what happens when you edit a file multiple times between commits.

Editing the Same File Multiple Times Between Commits

Up until now, you've been editing files just once and adding them to the staging area. However, it is important to understand that if you add a file to the staging area and then make *another change* to the file, Git will interpret this as a new version of the file and it will mark the file as modified. Then, if you want the latest version of the file to be included in your next commit, you will have to add the updated version of the file to the staging area *again*.

To see this in action, in this section your friend is going to add the color blue to the list of colors in `rainbowcolors.txt` and add the edited file to their staging area—but they will make a typo, writing “Bloo is the fifth color of the rainbow.” When they notice this they will edit the file again to fix the typo, and you will see how the file will need to be added to the staging area a second time in order for their latest changes to be included in the next commit.

The upcoming Visualize It diagrams will use the Git Diagram we created in [Chapter 2](#), and we will zoom in on the `friend-rainbow` repository to illustrate all of this happening. First, go to [Follow Along 9-4](#) to check the state of the `friend-rainbow` working directory and staging area at the moment.

[FOLLOW ALONG 9-4]

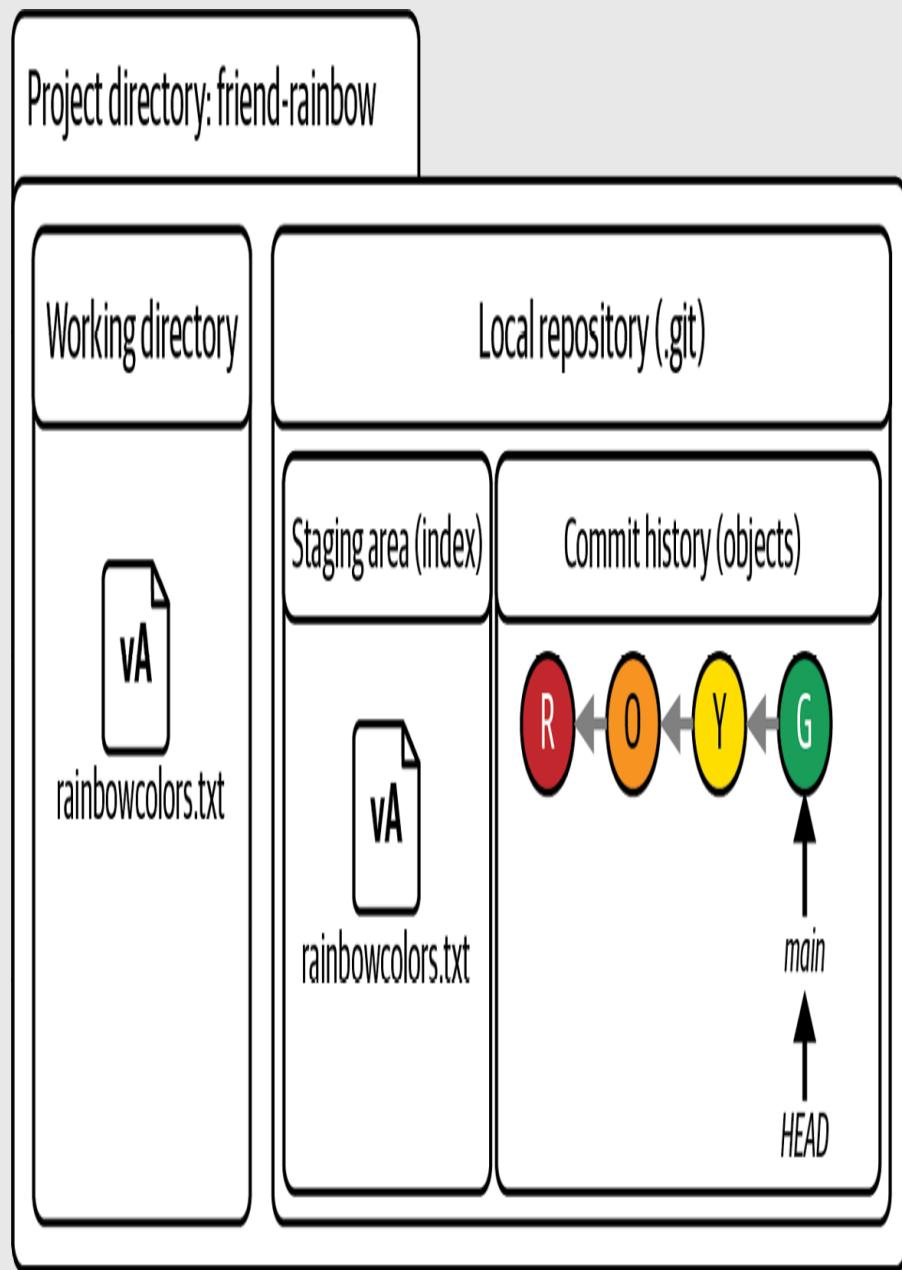
```
1 friend-rainbow $ git status
   on branch main
     Your branch is up to date with 'origin/main'.
       nothing to commit, working tree clean
```

What to notice:

- The `git status` output states `nothing to commit, working tree clean`, which indicates that the state of the working directory and staging area are the same. There are no modified files in the working directory and there are no newly staged files in the staging area.

This observation is illustrated in [Visualize it 9-5](#).

[VISUALIZE IT 9-5]



Git Diagram showing the current state of the `friend-rainbow` project directory

What to notice:

- The version of the `rainbowcolors.txt` file in the working directory and staging area mentions the colors red, orange, yellow, and green. We represent it as version A (vA).

Next, in [Follow Along 9-5](#), your friend is going to edit the `rainbowcolors.txt` file and add the color blue.

[FOLLOW ALONG 9-5]

1 Open the `rainbowcolors.txt` in the `friend-rainbow` project directory in a text editor window, add “Bloo is the fifth color of the rainbow.” on line 5, and save the file. Note that you are including the sentence with a typo here on purpose for learning purposes.

2

```
friend-rainbow $ git status
on branch main

Your branch is up to date with 'origin/main'.

Changes not staged for commit:

  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   rainbowcolors.txt

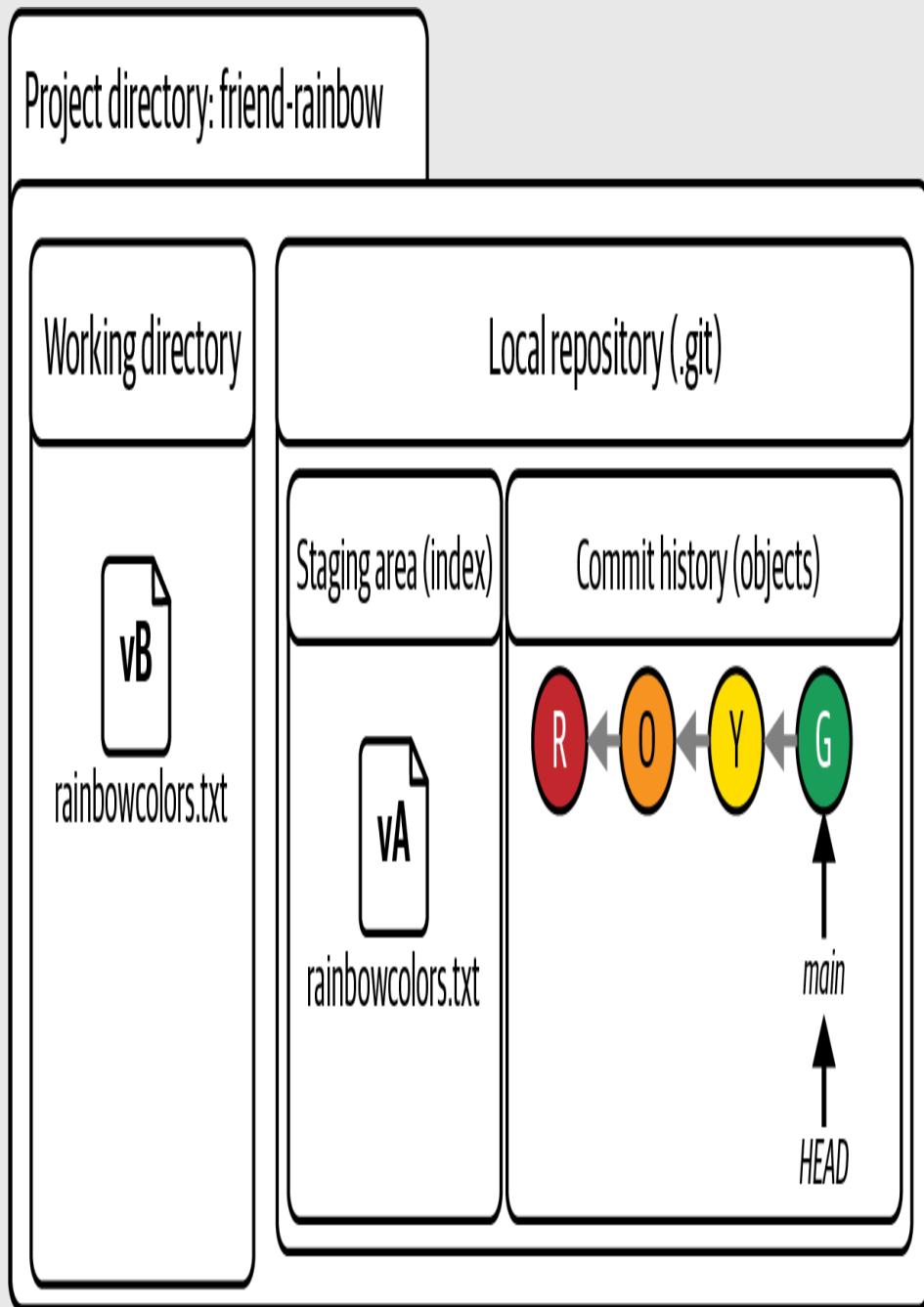
no changes added to commit (use "git add" and/or "git commit -a")
```

What to notice:

- In step 2, the `git status` output indicates that there is now one modified file in the working directory, which is not yet staged for commit.

This observation is illustrated in [Visualize it 9-6](#).

[VISUALIZE IT 9-6]



The `friend-rainbow` project directory after your friend edits the `rainbowcolors.txt` file in the working directory

What to notice:

- The version of the `rainbowcolors.txt` file in the staging area is still vA; it hasn't changed.
- The version of the `rainbowcolors.txt` file in the working directory has changed: it mentions the colors red, orange, yellow, green, and blue. We represent it as version B (vB).

Next, in [Follow Along 9-6](#), let's see what happens when your friend adds the file to the staging area.

[FOLLOW ALONG 9-6]

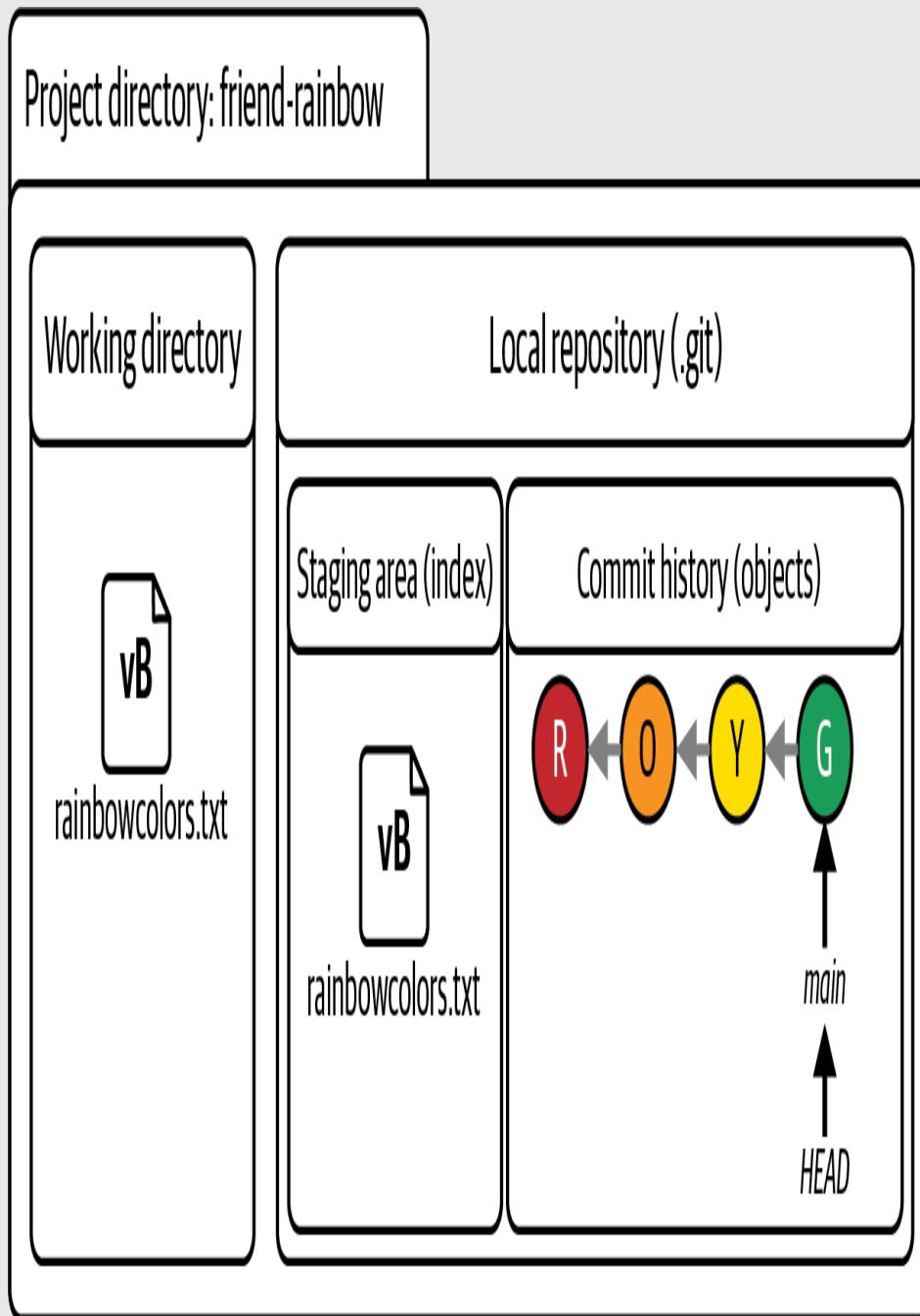
```
1 friend-rainbow $ git add rainbowcolors.txt  
  
2 friend-rainbow $ git status  
on branch main  
Your branch is up to date with 'origin/main'.  
Changes to be committed:  
(use "git restore --staged <file>..." to unstage)  
modified:   rainbowcolors.txt
```

What to notice:

- In step 2, the `git status` output indicates that there is now one modified file in the working directory that is also staged for commit.

This is illustrated in [Visualize it 9-7](#).

[VISUALIZE IT 9-7]



The `friend-rainbow` project directory after your friend adds the modified `rainbowcolors.txt` file to the staging area

What to notice:

- The version of the `rainbowcolors.txt` file in the staging area has changed from vA to vB.
- The version of the `rainbowcolors.txt` file in the staging area is the same as the one in the working directory.

Your friend has added the updated version of the `rainbowcolors.txt` file to the staging area, and it is ready to be committed. Now, go to [Follow Along 9-7](#) to see what happens when your friend goes back to edit the file again to fix the typo, changing it from “Bloo” to “Blue”.

[FOLLOW ALONG 9-7]

1 In the `friend-rainbow` project directory in your text editor, fix the typo on line 5 of the `rainbowcolors.txt` file so the sentence reads “Blue is the fifth color of the rainbow.” Save the file.

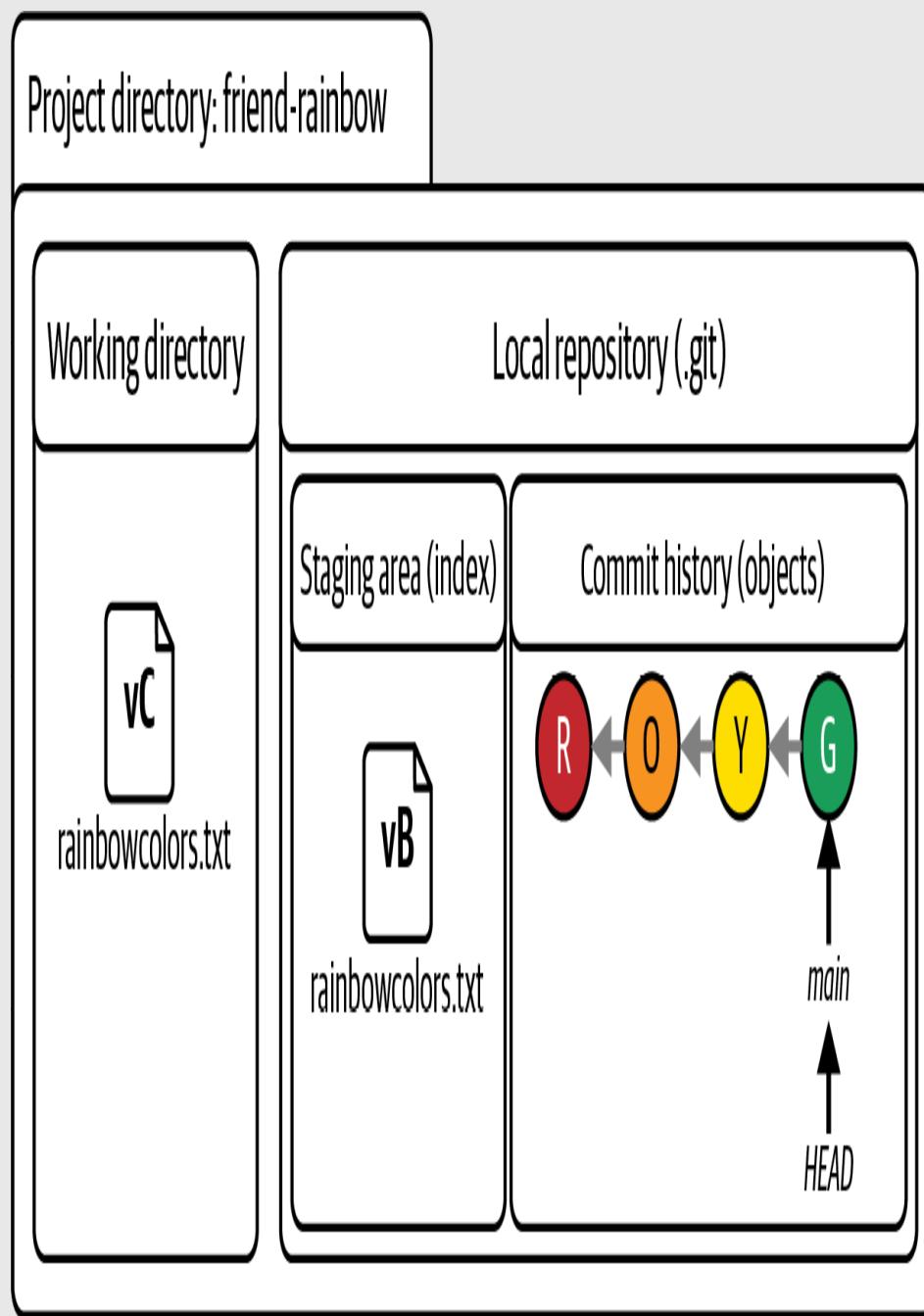
2 friend-rainbow \$ **git status**
on branch main
Your branch is up to date with 'origin/main'.
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
modified: `rainbowcolors.txt`
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working directory)
modified: `rainbowcolors.txt`

What to notice:

- In the `git status` output in step 2, one version of the `rainbowcolors.txt` file is listed as a modified file that is staged for commit and another version of the `rainbowcolors.txt` file is listed as a modified file that is *not* staged for commit.

This is illustrated in [Visualize it 9-8](#).

[VISUALIZE IT 9-8]



The `friend-rainbow` project directory after your friend edits the `rainbowcolors.txt` file for the second time

What to notice:

- The version of the `rainbowcolors.txt` file in the staging area is vB. This is the version of the file with the typo.
- The version of the `rainbowcolors.txt` file in the working directory has changed from vB to version C (vC). The vC file is the version without the typo.

From these observations, you can see that just because a file with a particular name is staged for commit *doesn't mean* it is automatically updated with any other changes you make to it. You need to explicitly add each updated version of a file to the staging area to include the latest changes in your next commit. Go to [Follow Along 9-8](#) to see this in practice.

[FOLLOW ALONG 9-8]

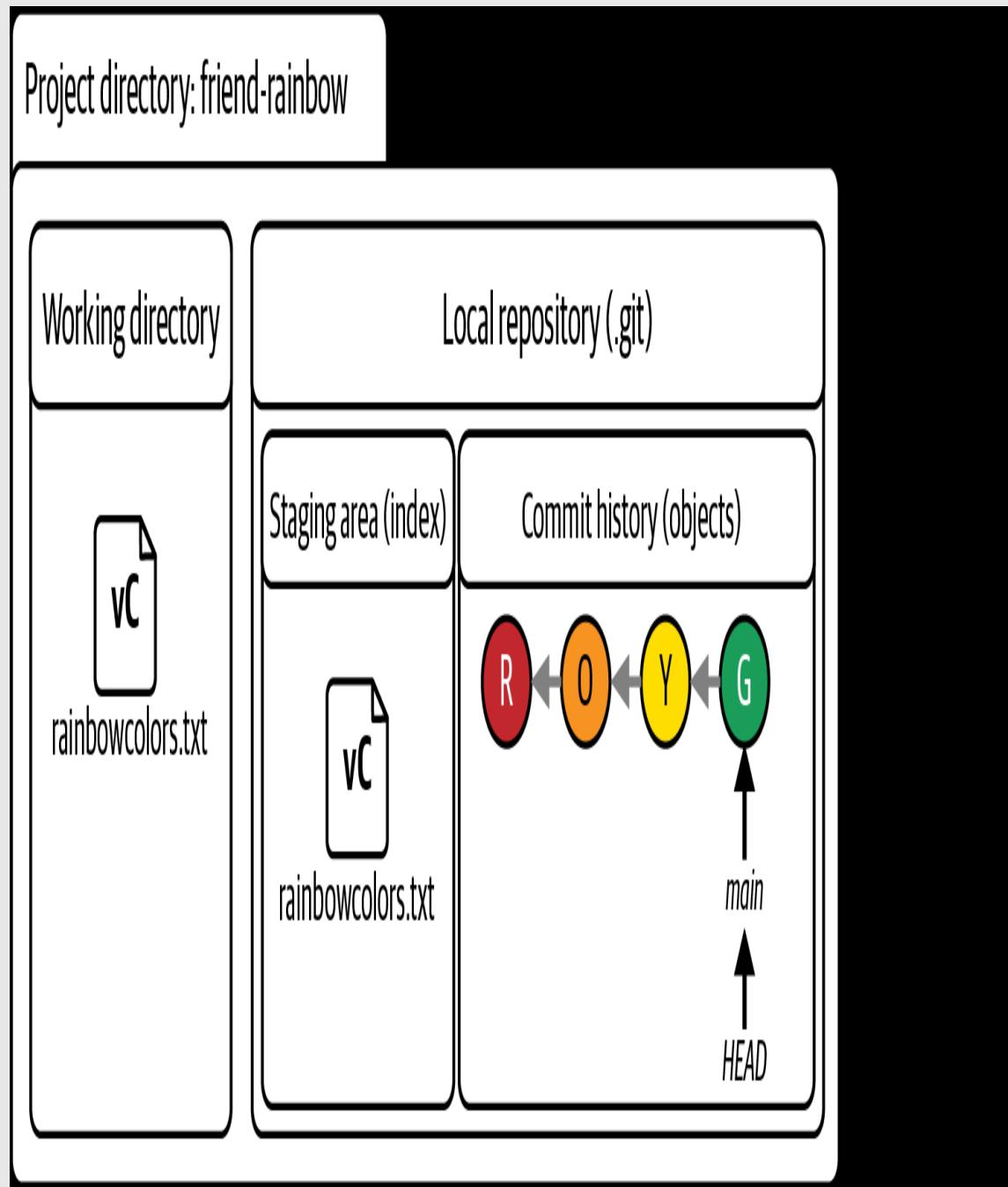
```
1 friend-rainbow $ git add rainbowcolors.txt  
2 friend-rainbow $ git status  
On branch main  
Your branch is up to date with 'origin/main'.  
Changes to be committed:  
(use "git restore --staged <file>..." to unstage)  
modified:   rainbowcolors.txt
```

What to notice:

- In the `git status` output in step 2, the `rainbowcolors.txt` file is listed as a modified file staged for commit. There are no more modified files that can be added to the staging area.

This is illustrated in [Visualize it 9-9](#).

[VISUALIZE IT 9-9]



The `friend-rainbow` project directory after your friend adds the `rainbowcolors.txt` file to the staging area again

What to notice:

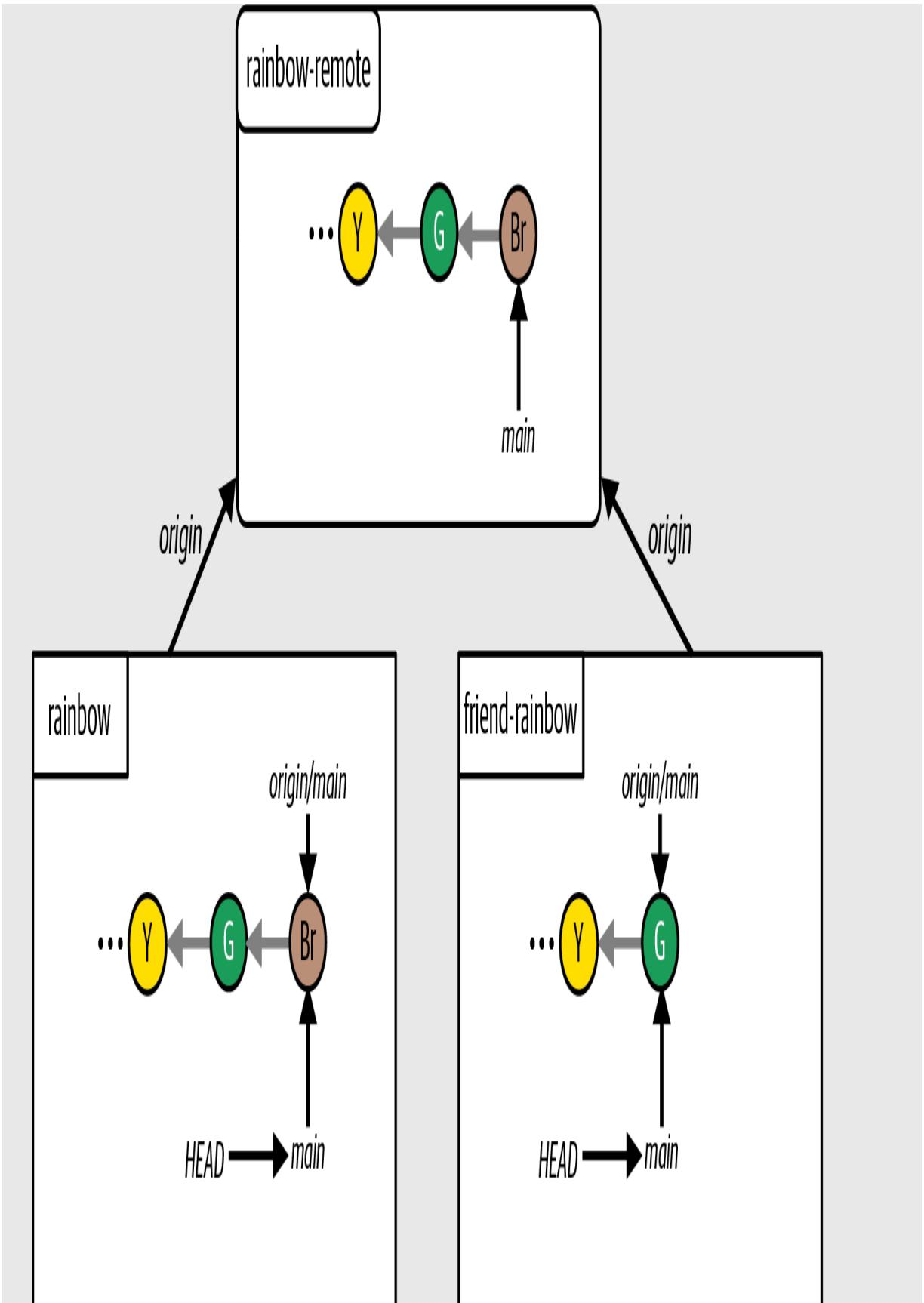
- The version of the `rainbowcolors.txt` file in the staging area has changed from vB to vC, indicating that the latest version of the `rainbowcolors.txt` file has been added to the staging area.

Next, your friend is going to finish making the blue commit and attempt to push their work to the remote repository. However, since their local `main` branch is out of sync with the remote `main` branch, they will come across an error. You'll see this in the next section.

Working at the Same Time as Others on Different Files

In [Visualize it 9-10](#), you can see that your friend has not fetched the brown commit that you made on the `main` branch in the `rainbow` repository and pushed to the remote repository.

[VISUALIZE IT 9-10]



The current state of the local and remote repositories

The local `main` branch in the `friend-rainbow` repository is out of sync with the remote `main` branch. Your friend is ready to make their blue commit and to try to share their work, but when they try to push their work to the remote repository they will get an error. Let's see this in action in [Follow Along 9-9.](#)

[FOLLOW ALONG 9-9]

1 friend-rainbow \$ **git commit -m "blue"**

[main 342bbfc] blue

1 file changed, 2 insertions(+), 1 deletion(-)

2 friend-rainbow \$ **git push**

To github.com:gitlearningjourney/rainbow-remote.git

! [rejected] main -> main (fetch first)

error: failed to push some refs to 'github.com:gitlearningjou
rney/rainbow-remote.git'

hint: Updates were rejected because the remote contains work that you do

hint: not have locally. This is usually caused by hint: another

repository pushing

hint: to the same ref. You may want hint: to first integrate the remote
changes

hint: (e.g., 'git pull hint: ...') before pushing again.

hint: See the 'Note about fast-forwards' in 'git push --help' for
details.

3 friend-rainbow \$ **git log**

commit 342bbfc96bb03053f23ea7f7564ca207c58ceab2 (**HEAD -> main**)

Author: annaskoulikari <gitlearningjourney@gmail.com>

Date: Sat Feb 19 13:00:56 2022 +0100

blue

commit 6987cd2996e245ec24ee9c5ea99874f0a01a31cd (**origin/main,**
origin/HEAD)

Author: annaskoulikari <gitlearningjourney@gmail.com>

Date: Sat Feb 19 11:49:03 2022 +0100

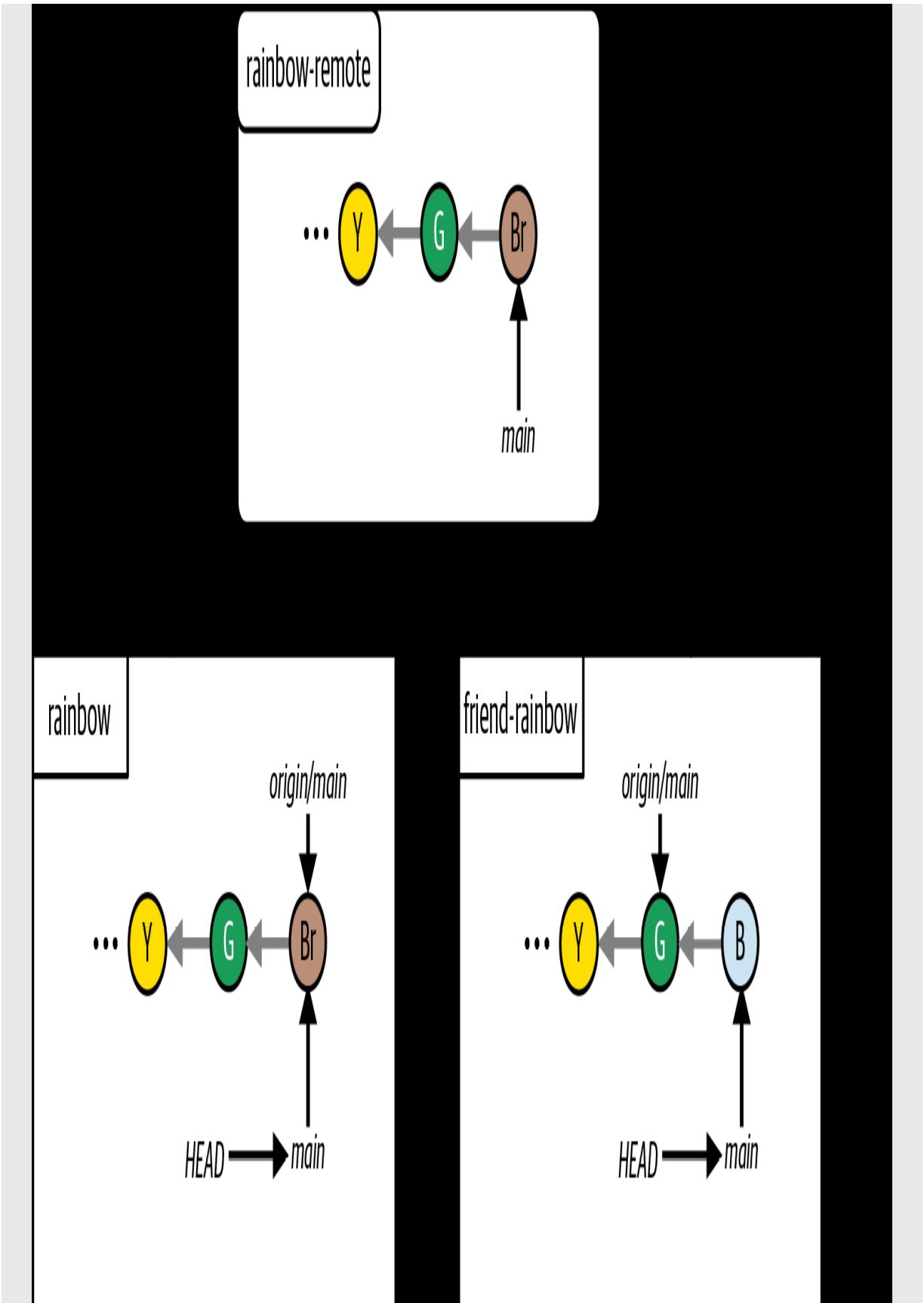
green

What to notice:

- In step 2, the output of the `git push` command shows that your friend gets an error. They are not able to push their changes to the remote repository.

The development histories of the local `main` branch and the remote `main` branch have diverged, and Git is not able to merge changes from one into the other with a simple fast-forward merge. The addition of the blue commit and the divergent development histories are illustrated in [Visualize it 9-11](#).

[VISUALIZE IT 9-11]



The Rainbow project after your friend makes the blue commit in the `friend-rainbow` repository

What to notice:

- In the `friend-rainbow` repository, the local `main` branch points to the blue commit.
- In the `rainbow-remote` repository, the `main` branch points to the brown commit.

Let's take a closer look at the error message your friend received:

```
error: failed to push some refs to  
'https://github.com/gitlearningjourney/rainbow-remote.git'  
hint: Updates  
were rejected because the remote contains work that you do not have  
locally. This is usually caused by another repository pushing to the same  
ref. You may want to first integrate the remote changes (e.g., 'git pull ...')  
before pushing again. See the 'Note about fast-forwards' in 'git push --help'  
for details.
```

Git is telling your friend that there are commits on the remote `main` branch that they have not yet fetched (or pulled). It advises your friend that they have to fetch (or pull) the remote work and integrate it into their local `main` branch before they can push to the remote repository. Note that pulling is similar to fetching, with some differences; we'll cover pulling later in this chapter. For now, to integrate the remote changes, your friend is going to execute a three-way merge.

Three-Way Merge in Practice

In [Chapter 8](#), you learned that incorporating changes from a remote repository is a two-step process:

1. First, you fetch the changes from the remote repository.
2. Second, you integrate the changes into the local branch in the local repository.

Your friend is going to follow the same steps right now, except at step 2, they'll end up carrying out a three-way merge instead of a fast-forward merge because the development histories of the two branches involved in the merge have diverged.

When performing a three-way merge, Git will create a merge commit. To do that, it will enter a text editor in the command line. Let's learn a bit more about how you're going to work with this command line text editor.

INTRODUCING VIM, THE COMMAND LINE TEXT EDITOR

Git, by default, uses a text editor in the command line called *Vim* to write commit messages. If a commit is being made (whether a regular commit or a merge commit) and a commit message is not specified, Git will enter Vim in the command line. You have not yet come across Vim in this book because every time you have made a commit using the `git commit` command in the Rainbow project you have passed in the `-m` option and explicitly specified a commit message.

[NOTE]

Git allows you to change the default command line text editor it uses from Vim to another text editor. For example, you can set it to the same text editor you use to edit files in your project.

Vim may appear scary because the command line will look different, and unless you know some Vim basics it is not intuitive to use. Let's cover the necessary basics to navigate Vim.

When Git enters Vim in the command line, you have the choice to enter text, edit text, or approve and save text. In the case of a three-way merge, Git will draft a default commit message for the merge commit and Vim will present it to you, as seen in [Figure 9-5](#).

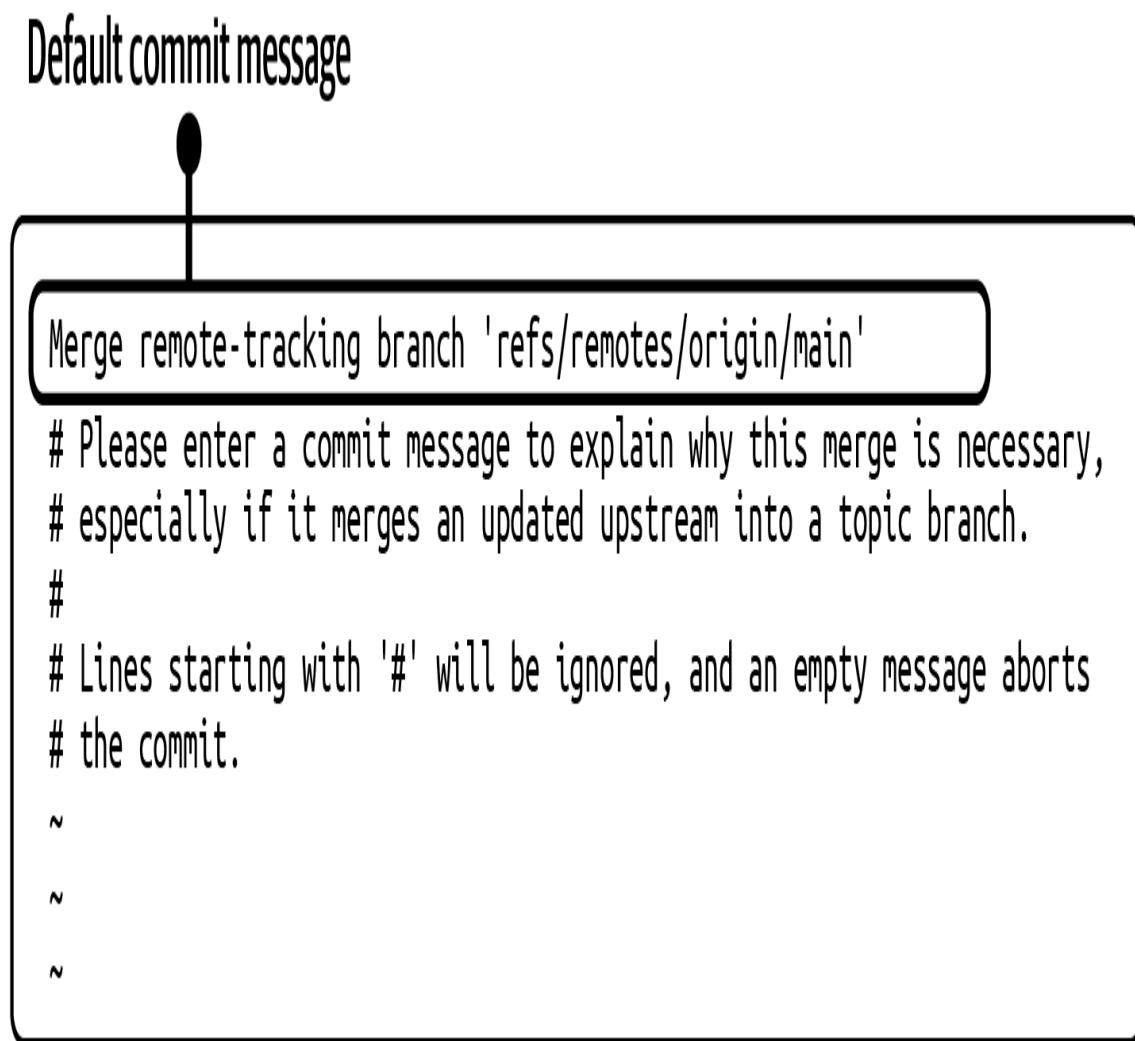


FIGURE 9-5

An example of a default commit message drafted by Git and presented in Vim in the

command line

For the purposes of this example, we'll simply cover how to accept and save the default commit message. To do that you must press the Escape key, type a colon (:), then press the W key, the Q key, and the Enter key one after another in order, as seen in [Figure 9-6](#). The `w` stands for “write” in Vim, which means save, and the `q` stands for “quit,” which means exit. (If anything goes wrong, you can press Escape to try again.)

```
Merge remote-tracking branch 'refs/remotes/origin/main'  
# Please enter a commit message to explain why this merge is necessary,  
# especially if it merges an updated upstream into a topic branch.  
#  
# Lines starting with '#' will be ignored, and an empty message aborts  
# the commit.  
~  
~  
:wq
```

Command to enter to save and quit Vim

FIGURE 9-6

The command to enter to save and quit Vim

This saves the commit message and exits the Vim command line text editor. These instructions will be repeated in every Follow Along that requires you to exit Vim.

Now that we've covered some basics of how to work with Vim, your friend is ready to carry out the two steps to incorporate the remote changes into

their local branch.

EXECUTING THE THREE-WAY MERGE

Having read the error message Git presented, your friend is now going to fetch the changes from the remote `main` branch into their local repository.

This will update the `origin/main` remote-tracking branch. Then they will merge the `origin/main` remote-tracking branch into their local `main` branch.

Go to [Follow Along 9-10](#) to carry out the two steps to incorporate the remote changes into the local `main` branch in the `friend-rainbow` repository.

[FOLLOW ALONG 9-10]

1 friend-rainbow \$ **git fetch**
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 290 bytes | 145.00 KiB/s, done.
From github.com:gitlearningjourney/rainbow-remote
6987cd2..7f0a87a main -> origin/main

2 friend-rainbow \$ **git merge origin/main**

3 The command line will enter Vim after the `git merge` command is executed.
The default commit message that Git drafts for you will either be `Merge remote-tracking branch 'refs/remotes/origin/main'` or `Merge remote-tracking branch 'origin/main'`. To accept the default message in the Vim editor and exit the editor, you must press the Escape key, type `:wq`, and then press the Enter key. You will see the following output:

```
Merge made by the 'ort' strategy.  
othercolors.txt | 1 +  
1 file changed, 1 insertion(+)  
create mode 100644 othercolors.txt
```

[FOLLOW ALONG 9-10]

4

```
friend-rainbow $ git log
commit 225839938563c7458af81daca7beb782dfcbfb27 (HEAD -> main)
Merge: 342bbfc 7f0a87a
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 13:05:46 2022 +0100
        Merge remote-tracking branch 'refs/remotes/origin/main'
commit 342bbfc96bb03053f23ea7f7564ca207c58ceab2
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 13:00:56 2022 +0100
        blue
commit 7f0a87a318e50638eec50a484bf8dfa76b76d08e (origin/main,
origin/HEAD)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 12:46:29 2022 +0100
        brown
```

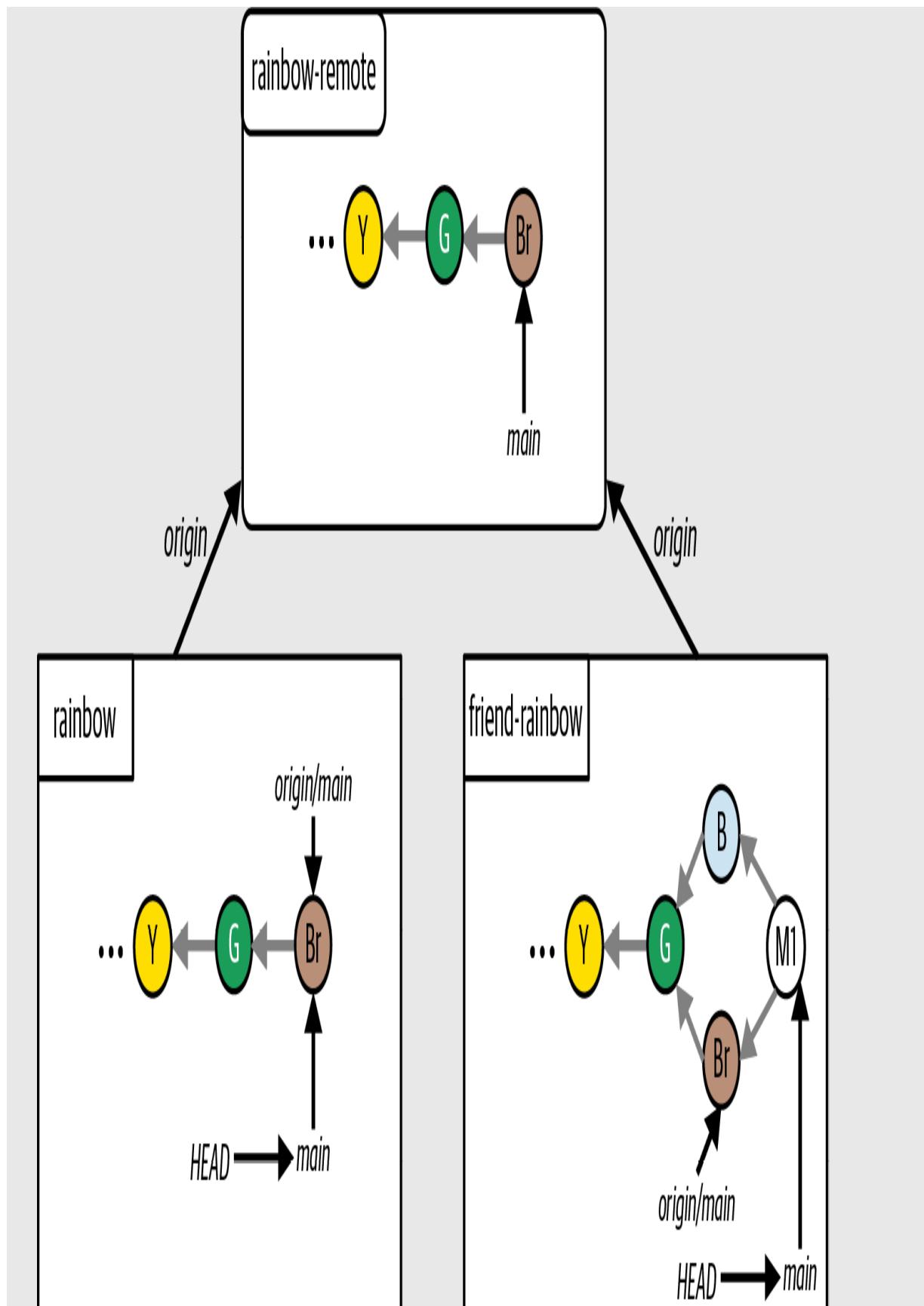
What to notice:

- In step 3, the `git merge` output says `Merge made by the 'ort' strategy`. This indicates that this was a three-way merge instead of a fast-forward merge. (If you have an older version of Git, your output may say that the merge was made by the “recursive” strategy; this is also correct.)
- Git drafted a default commit message for you which was either `Merge remote-tracking branch 'refs/remotes/origin/main'` OR `Merge remote-tracking branch 'origin/main'`. Both are correct.

- In step 4, the `git log` output shows the merge commit `2258399` and lists the two parent commits in the `Merge` field: `342bbfc` (the blue commit) and `7f0a87a` (the brown commit). The commit hashes for your commits will be different from the ones in this book, because commit hashes are unique.

[Visualize it 9-12](#) shows the state of the Rainbow project after [Follow Along 9-10.](#)

[VISUALIZE IT 9-12]



The Rainbow project after your friend fetches the brown commit and carries out a three-way merge

What to notice:

- In the `friend-rainbow` repository, we illustrate the merge commit as M1 and show how it ties the two development histories together.

The M1 merge commit has two parent commits. Recall that in [Chapter 4](#), you used the command `git cat-file -p <commit_hash>` to view the parent commits of a commit. In the upcoming Follow Along, your friend will use the same command, passing in the commit hash of the M1 merge commit, to see its parent commits (represented by their commit hashes). To retrieve the commit hash of the M1 merge commit, you can use the output of the `git log` command. Recall that you must use the commit hash of *your* M1 merge commit, not the commit hash in this book (because commit hashes are unique).

After that, your friend will push their changes to the remote repository to make sure it's updated. Go to [Follow Along 9-11](#) to perform these actions now.

[FOLLOW ALONG 9-11]

1 Retrieve the commit hash for the M1 merge commit (you can copy this from the `git log` output in Follow Along 9-10). You must pass this commit hash as an argument to the `git cat-file -p` command in step 2 of this Follow Along. You may copy and paste the entire commit hash or just enter the first seven characters, as shown here.

2 friend-rainbow \$ **git cat-file -p 2258399**

```
tree 45330906e6041a0cd07849617a25443a9a5b08bd
parent 342bbfc96bb03053f23ea7f7564ca207c58ceab2
parent 7f0a87a318e50638eec50a484bf8dfa76b76d08e
author annaskoulikari <gitlearningjourney@gmail.com> 1645272346 +0100
committer annaskoulikari <gitlearningjourney@gmail.com> 1645272346 +0100
Merge remote-tracking branch 'refs/remotes/origin/main'
```

3 friend-rainbow \$ **git push**

```
Enumerating objects: 9, done.

Counting objects: 100% (8/8), done.

Delta compression using up to 4 threads

Compressing objects: 100% (4/4), done.

Writing objects: 100% (5/5), 586 bytes | 586.00 KiB/s, done.

Total 5 (delta 1), reused 0 (delta 0), pack-reused 0

remote: Resolving deltas: 100% (1/1), completed with 1 local object.

To github.com:gitlearningjourney/rainbow-remote.git

    7f0a87a..2258399  main -> main
```

[FOLLOW ALONG 9-11]

4

```
friend-rainbow $ git log
commit 225839938563c7458af81daca7beb782dfcbfb27 (HEAD -> main,
origin/main, origin/HEAD)
Merge: 342bbfc 7f0a87a
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 13:05:46 2022 +0100
        Merge remote-tracking branch 'refs/remotes/origin/main'
commit 342bbfc96bb03053f23ea7f7564ca207c58ceab2
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 13:00:56 2022 +0100
        blue
```

5

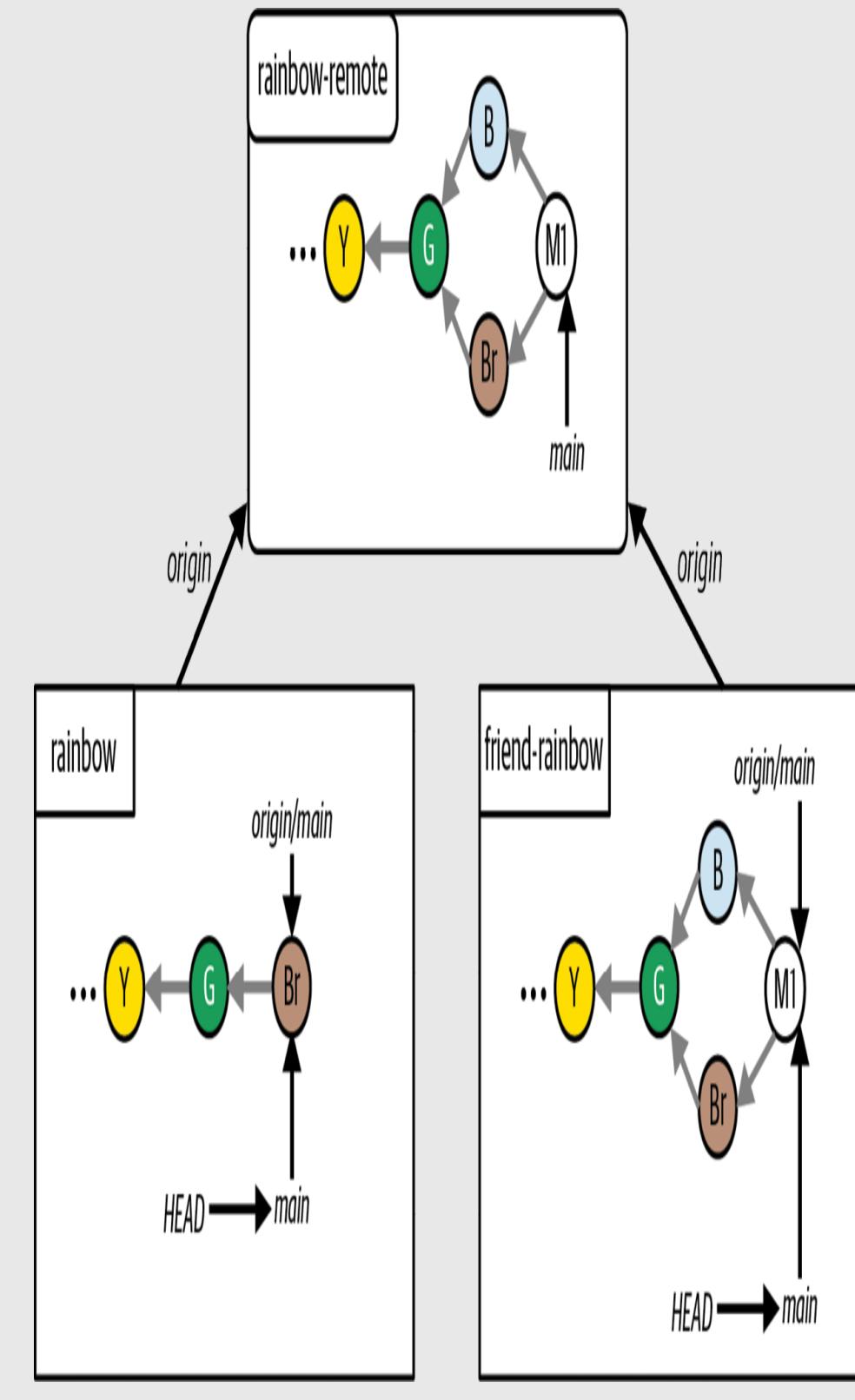
Go to the `rainbow-remote` repository on your hosting service and refresh the page. The merge commit should be there.

What to notice:

- In step 2, the `git-cat file -p` output shows the two commits listed as parents: `342bbfc96bb03053f23ea7f7564ca207c58ceab2` (the blue commit) and `7f0a87a318e50638eec50a484bf8dfa76b76d08e` (the brown commit).
- In step 5, you can see that the merge commit is in the remote repository.

These observations are illustrated in [Visualize it 9-13](#).

[VISUALIZE IT 9-13]



The Rainbow project after your friend pushes their changes to the remote repository

The M1 merge commit is now in the `friend-rainbow` repository and the `rainbow-remote` repository. The next step is for you to sync your `rainbow` repository with the remote repository, updating your local `main` branch with the merge commit as well. To do that, you will learn about pulling in Git.

Pulling Changes from a Remote Repository

Up until now, in the Rainbow project, when you wanted to update your local repository with changes from the remote repository you did it in two steps: first you fetched the data from the remote repository (using the `git fetch` command), and then you merged the data (using the `git merge` command) into the local branch. Pulling data allows you to do both in one go.

In Git, we use the term *pull* or *pulling* to refer to the process of fetching data from a remote repository and integrating it into a branch in a local repository in one go, and the command we use to do it is `git pull`. If you don't have an upstream branch defined for your local branch, then you must specify the shortname of the remote repository and the name of the branch that you want to update. If you do have an upstream branch defined, you can just use `git pull` without any arguments.

[SAVE THE COMMAND]

git pull <shortname> <branch_name>

Fetch and integrate changes from the `<shortname>` remote repository for the specified `<branch_name>`

git pull

If an upstream branch is defined for the current branch, fetch and integrate changes from the defined upstream branch

There's one more thing you need to know about the `git pull` command. As you've learned, in Git there are two ways to integrate changes: merging and rebasing. Which method the `git pull` command uses will depend on whether the development histories of the branches have diverged and, if so, on the option you choose when entering the command:

- If the development histories of the local branch and remote branch in a `git pull` have *not* diverged, then by default a fast-forward merge will occur.
- If the development histories of the local branch and the remote branch in a `git pull` *have* diverged, then you must tell Git whether you want to integrate the changes by merging or rebasing (otherwise, you'll get an error). To tell Git to integrate the changes by merging, you must pass in the `--no-rebase` option. To tell Git to integrate the changes by rebasing, you must pass in the `--rebase` option.

As illustrated in [Figure 9-7](#), the `git pull` command effectively combines the `git fetch` command and either the `git merge` or the `git rebase` command.

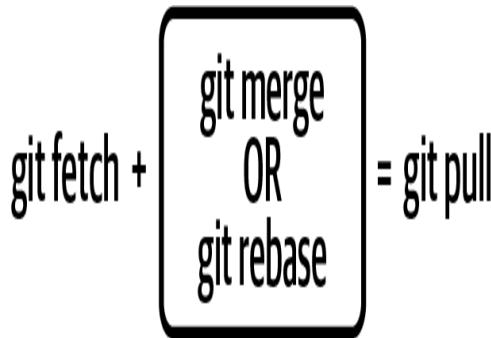


FIGURE 9-7

The `git pull` command fetches and integrates changes in one go

Now the question is, when should you fetch and integrate changes in two steps by using the `git fetch` command and then either the `git merge` or the `git rebase` command, and when should you carry out both of those steps in one go by just using the `git pull` command?

It is common for Git users to use the `git pull` command when the development histories of the local and remote branches have not diverged, and therefore a simple fast-forward merge will happen. If the development histories of the local and remote branches have diverged, Git users often prefer to use the `git fetch` command then choose whether to rebase or merge in a separate step. By carrying out the process in two steps, they give themselves more time to look at what is going to change in their local branch and to prepare for the integration process.

In this book you will follow this common practice, and therefore you will use only the `git pull` command when a fast-forward merge will update your local branch.

Go to [Follow Along 9-12](#) to practice using the `git pull` command by pulling the changes from the remote `main` branch to the local `main` branch in the `rainbow` repository.

[FOLLOW ALONG 9-12]

```
1 rainbow $ git pull
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 5 (delta 1), reused 5 (delta 1), pack-reused 0
Unpacking objects: 100% (5/5), 566 bytes | 141.00 KiB/s, done.
From github.com:gitlearningjourney/rainbow-remote
    7f0a87a..2258399  main      -> origin/main
Updating 7f0a87a..2258399
Fast-forward
  rainbowcolors.txt | 3 +-+
  1 file changed, 2 insertions(+), 1 deletion(-)
```

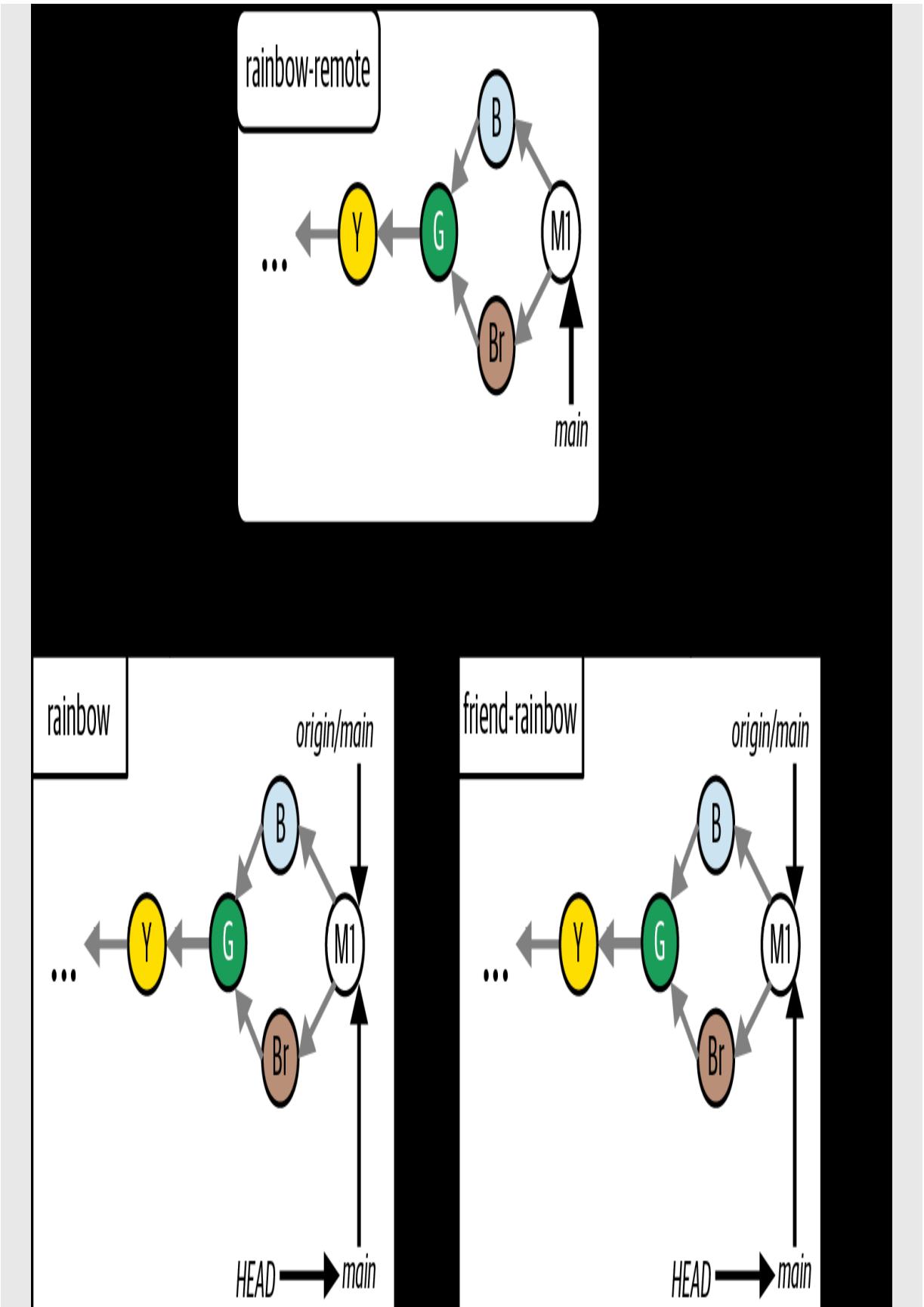
```
2 rainbow $ git log
commit 225839938563c7458af81daca7beb782dfcbfb27 (HEAD -> main,
origin/main)
Merge: 342bbfc 7f0a87a
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 13:05:46 2022 +0100
        Merge remote-tracking branch 'refs/remotes/origin/main'
commit 342bbfc96bb03053f23ea7f7564ca207c58ceab2
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 13:00:56 2022 +0100
        blue
```

What to notice:

- The first half of the `git pull` output in step 1 shows the data is being fetched or downloaded. The second half of the output indicates that a `Fast-forward` merge was carried out.
- The `git log` output in step 2 shows that you now have the blue commit and the merge commit in the `rainbow` repository, and the `origin/main` remote-tracking branch and the `main` branch have been updated to point to the merge commit.

[Visualize it 9-14](#) illustrates these observations.

[VISUALIZE IT 9-14]



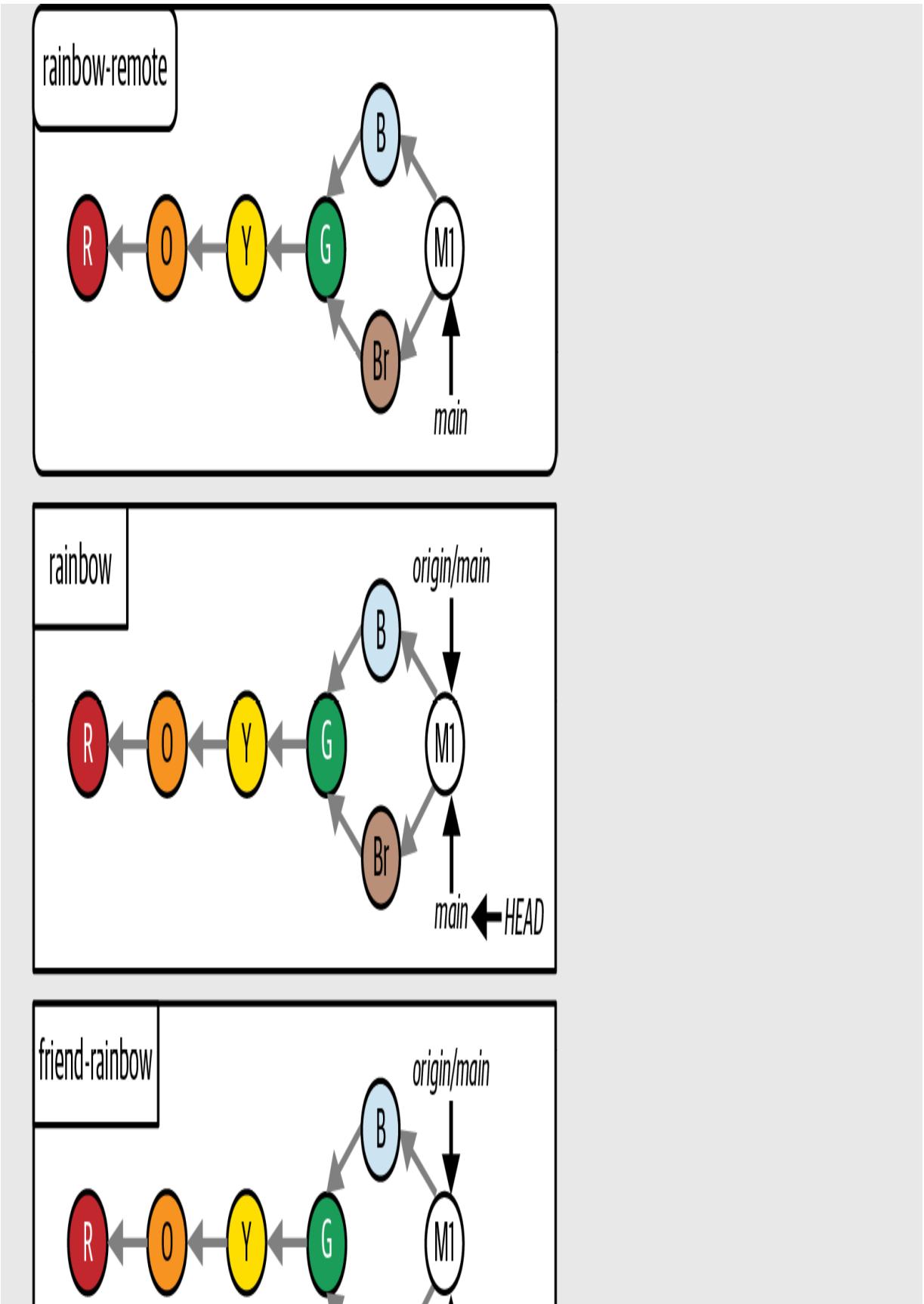
The Rainbow project after you pull changes from the remote repository into the `main` branch in the `rainbow` repository

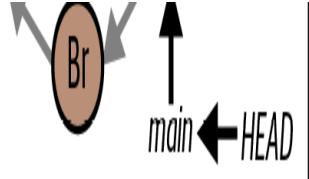
All three repositories in the Rainbow project are now in sync.

State of the Local and Remote Repositories

[Visualize it 9-15](#) shows the state of the local and remote repositories in the Rainbow project with all the commits that were made from [Chapter 1](#) all the way to the end of this chapter.

[VISUALIZE IT 9-15]





The Rainbow project at the end of [Chapter 9](#) with all the commits since [Chapter 1](#)

Summary

In this chapter you learned about three-way merges. To see how they work, you made changes in the Rainbow project that resulted in the histories of the local `main` branch in the `friend-rainbow` repository and the remote `main` branch diverging.

In the process of creating these divergent histories you learned how to define upstream branches, and you saw how after defining them you were able to use commands such as `git push` without specifying any additional arguments.

You also learned an important detail about editing files in your working directory multiple times between commits: namely, every time you make a change to a file and save it, Git will interpret this as a new version of the file and will mark the file as modified. To include the latest version of the file in the next commit, you will need to add that version of the file to the staging area.

When it came time to carry out the three-way merge, you learned about the Vim command line text editor and covered the basics of how to save Git's proposed commit message and exit Vim. At the end of the merge process, you saw how the merge commit tied the two divergent development histories together.

Finally, you learned about the process of pulling and how it allows you to carry out the two steps of incorporating the changes from a remote repository into a local repository in one go.

The three-way merge you carried out in this chapter didn't have any merge conflicts. In [Chapter 10](#), you're going to learn about merge conflicts and how to resolve them.

Merge Conflicts

In [Chapter 9](#), you carried out a three-way merge in which you did not experience any merge conflicts.

In this chapter, you’re going to learn about merge conflicts, how they arise, and how to resolve them by walking through a hands-on example of a three-way merge with a merge conflict.

State of the Local and Remote Repositories

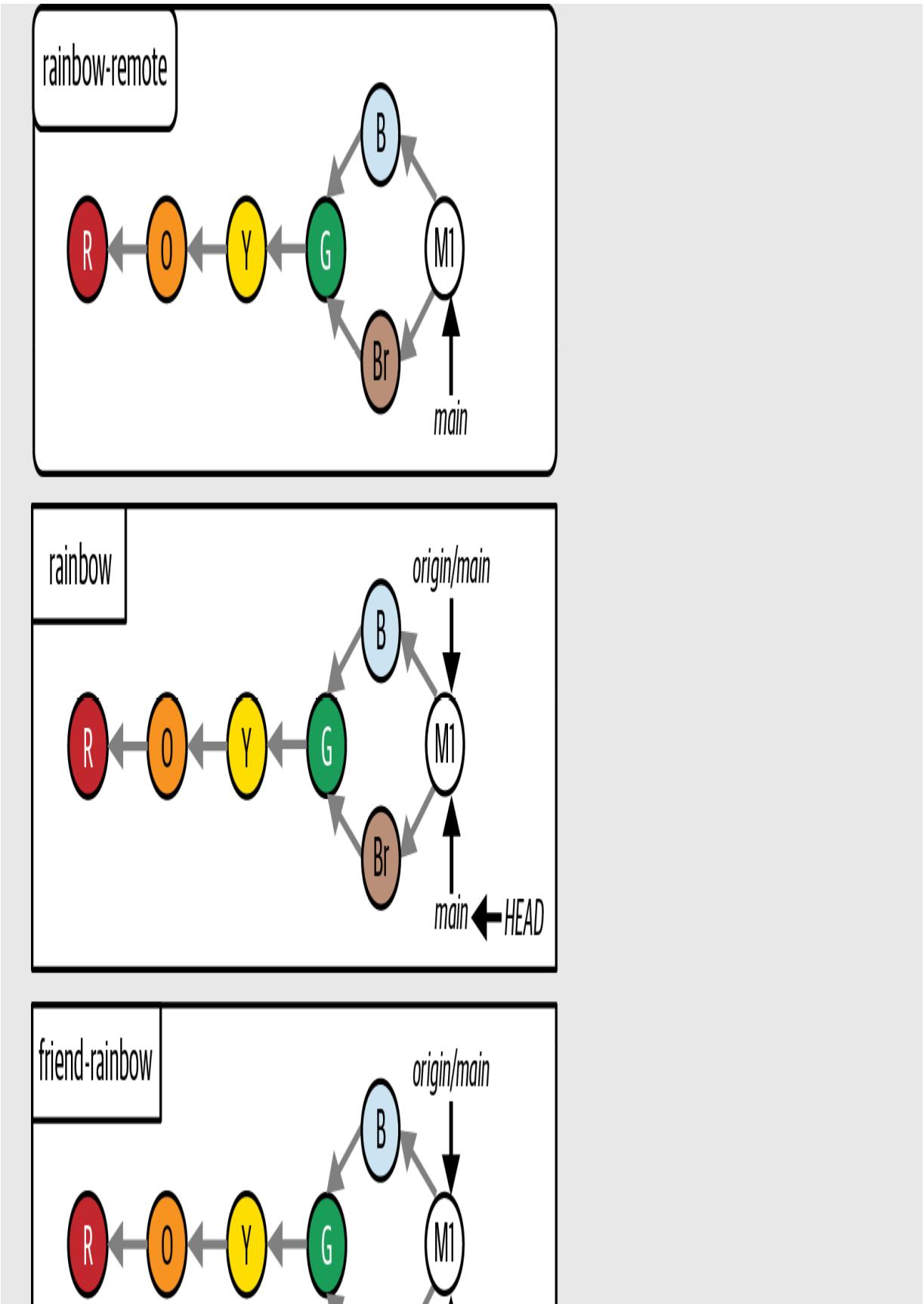
At the start of this chapter, you should have two local repositories called `rainbow` and `friend-rainbow` and one remote repository called `rainbow-remote`. All three of these repositories should be in sync, with the same commits and branches. As usual, I recommend that you use two separate text editor windows and command line windows for working with the `rainbow` repository and the `friend-rainbow` repository.

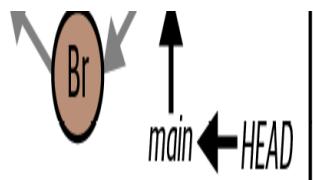
[Visualize it 10-1](#) shows the state of the local and remote repositories in the Rainbow project with all the commits that were made from [Chapter 1](#) through [Chapter 9](#).

To focus on the commits you are going to make in this chapter, from here on I’ll simplify the Visualize It diagrams and show only the last three commits that are part of the `main` branch in all the repositories: the blue

commit, the brown commit, and the M1 merge commit. This representation is shown in [Visualize it 10-2](#).

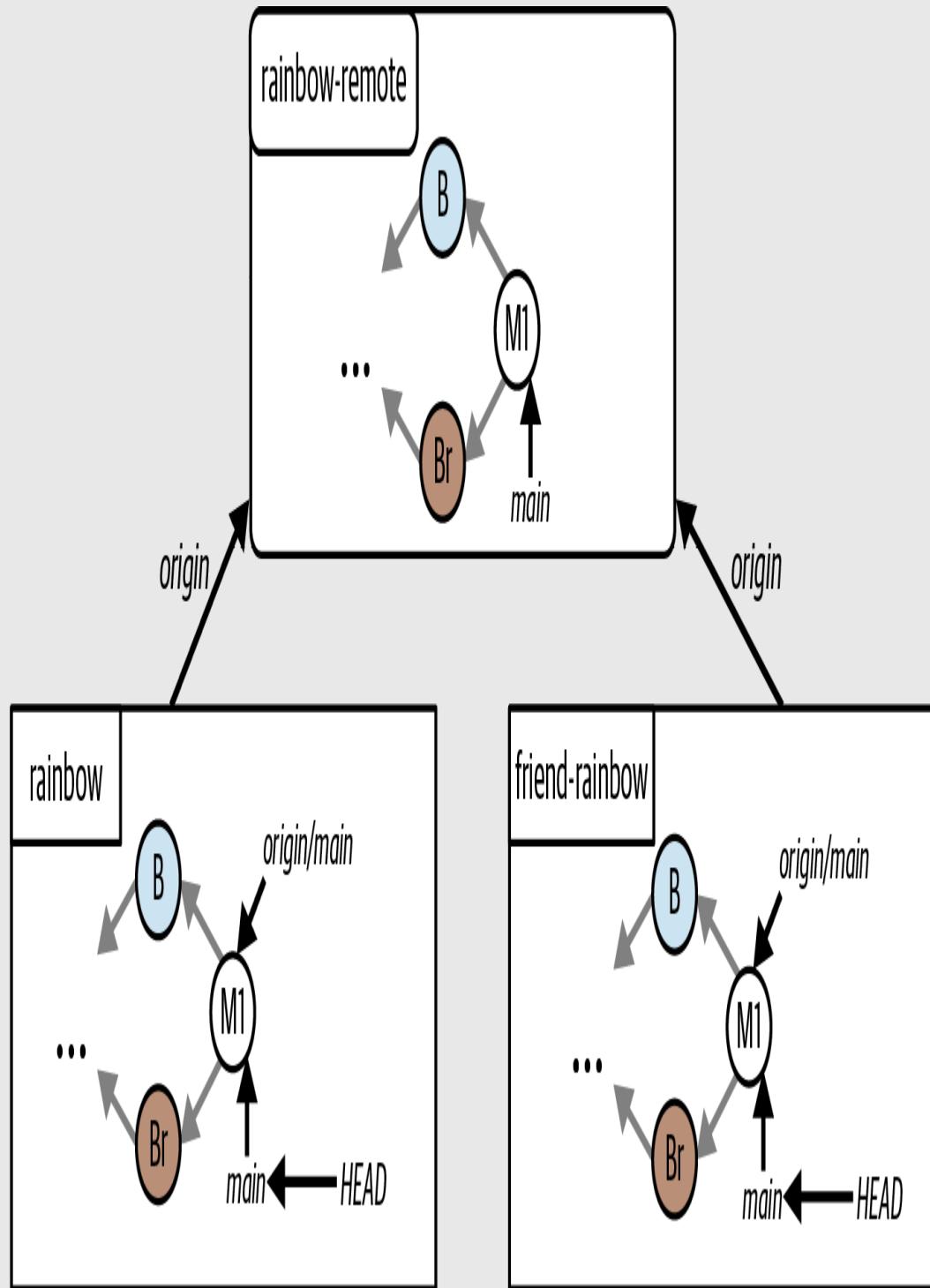
[VISUALIZE IT 10-1]





The Rainbow project at the start of [Chapter 10](#) with all the commits since [Chapter 1](#)

[VISUALIZE IT 10-2]



A simplified representation of the Rainbow project at the start of [Chapter 10](#), showing just the last three commits on the `main` branch in all the repositories

Introducing Merge Conflicts

In [Chapter 9](#), we covered what happens when you work on a project at the same time as someone else but you each make changes to different files: you edited the `othercolors.txt` file, while your friend edited the `rainbowcolors.txt` file. This meant that you were able to do a three-way merge without any merge conflicts.

Now, we will cover what happens when you want to integrate work where there will be a merge conflict. As you learned in the previous chapter, merge conflicts arise when you merge two branches where different changes have been made to the same parts in the same file(s), or if in one branch a file was deleted that was edited in the other branch. In these cases Git is unable to automatically merge the files, and therefore you must do it manually.

Merge conflicts can arise during the process of merging as well as the process of rebasing. In this chapter we will cover an example of a merge conflict in the Rainbow project while merging, and in [Chapter 11](#) we will cover an example of a merge conflict in the Rainbow project while rebasing.

Bear in mind that it is normal for merge conflicts to arise, and if they do, it does not mean that someone did something wrong while working on a project. Resolving merge conflicts is a regular part of working on Git projects. Let's take a look at [Example Book Project 10-1](#) to see an example of how such a situation might occur.

Example Book Project 10-1

Let's imagine a couple of scenarios where I may experience merge conflicts in my Book project. As you know, the project consists of 10 text files, one for each chapter of the book, called `chapter_one.txt`,

`chapter_two.txt`, and so on. My primary line of development is the `main` branch, and I've decided together with my editor and my coauthor that whenever I work on a chapter I will make a branch off the `main` branch; then, after they have both approved the work I have done on that branch, I can merge the branch back into the `main` branch. I've also agreed with my coauthor that only one of us should work on any one chapter at a time, and this way we can often avoid merge conflicts.

But now, let's imagine what happens if my coauthor and I accidentally miscommunicate. We both decide to edit chapter 3 at the same time, and we both make branches off the `main` branch to work on this chapter at the same time. I make the `chapter_three` branch, and my coauthor makes the `chapter_three_coauthor` branch. In this case, we are both going to edit the same file, `chapter_three.txt`.

Now, let's assume that my coauthor merges their work on the `chapter_three_coauthor` branch into the `main` branch first. When I go to merge my `chapter_three` branch into the most up-to-date version of the `main` branch, I will find that not only do I have to carry out a three-way merge, but I will also have to resolve merge conflicts.

This is because the version of the `chapter_three.txt` file in my `chapter_three` branch and the version of the file in the `main` branch have both been

edited in different ways since I made my branch off the `main` branch. Git isn't able to automatically merge the work. It needs me to go in and decide exactly what the final version of the `chapter_three.txt` file that will be included in the merge commit should look like.

This is one scenario in which I may experience merge conflicts. It occurs when the same file has been edited in different ways in the two branches involved in the merge.

Next, let's explore another scenario in which a merge conflict might arise. Let's imagine that my coauthor and I both agree that the last chapter in the book, chapter 10, isn't any good. However, we again miscommunicate about what we should do about it.

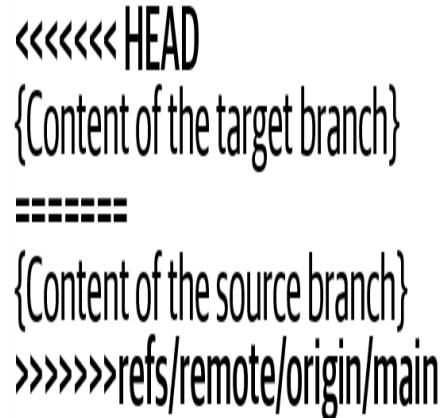
We accidentally both make branches off the `main` branch at the same time: I make the `chapter_ten` branch and my coauthor makes the `chapter_ten_coauthor` branch. In my `chapter_ten` branch, I decide to delete the `chapter_ten.txt` file because I think we should simply remove that chapter from our book. However, my coauthor, working on their `chapter_ten_coauthor` branch, instead decides to edit the `chapter_ten.txt` file to make it better.

Again, my coauthor merges their branch into the `main` branch first. And when I go to merge my branch into the most up-to-date `main` branch through a three-way merge, I will again encounter a merge conflict. This is another scenario where Git isn't able to automatically merge the work; it can't determine whether it should delete the `chapter_ten.txt` file or retain the edited version. To recap, this situation occurs when I am integrating two branches where in one branch a specific file has been edited and in the other branch that same file has been deleted.

[Example Book Project 10-1](#) described a couple of scenarios where merge conflicts might arise. Now let's take a look at how you can resolve them.

How to Resolve Merge Conflicts

When merge conflicts happen, you will see a set of special markers in each of the files involved that indicate where the conflicts occur. These markers, called *conflict markers*, consist of seven left angle brackets (`<<<<<`), seven equals signs (`=====`), and seven right angle brackets (`>>>>>`), as well as references to the branches involved in the merge. [Figure 10-1](#) shows an example. Above the row of equals signs you will see the content of the target branch, and below it you will see the content of the source branch.



```
<<<<<HEAD
{Content of the target branch}
=====
{Content of the source branch}
>>>>>refs/remote/origin/main
```

FIGURE 10-1

An example of the conflict markers that appear when there is a merge conflict

There are two steps to resolving merge conflicts:

1. Decide what to keep, edit the content, and remove the conflict markers.
2. Add the file(s) you have edited to the staging area and commit your changes.

We will cover each of these steps in detail when you practice resolving merge conflicts in the Rainbow project later in this chapter. To prepare for that, you need to create a situation with divergent development histories in your local repositories. You'll do that next.

Setting Up a Merge Conflict Scenario

To set up a three-way merge with merge conflicts, you and your friend have to make different changes to the same part in the same file. First, go to [Follow Along 10-1](#) to add the color indigo to the list of colors in the `rainbowcolors.txt` file in your `rainbow` repository and push the updated `main` branch to the remote repository.

[FOLLOW ALONG 10-1]

1 In the `rainbow` project directory in your text editor, in the `rainbowcolors.txt` file, add “Indigo is the sixth color of the rainbow.” on line 6. Then save the file.

2 `rainbow $ git add rainbowcolors.txt`

3 `rainbow $ git commit -m "indigo"`
[main 9b0a614] indigo
1 file changed, 2 insertions(+), 1 deletion(-)

4 `rainbow $ git push`
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 326 bytes | 326.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:gitlearningjourney/rainbow-remote.git
2258399..9b0a614 main -> main

[FOLLOW ALONG 10-1]

5

```
rainbow $ git log
commit 9b0a61461c8e8d74ed358e65b2662e3697b94de6 (HEAD -> main,
origin/main)

Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sun Feb 20 08:36:11 2022 +0100

    indigo

commit 225839938563c7458af81daca7beb782dfcbfb27

Merge: 342bbfc 7f0a87a
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sat Feb 19 13:05:46 2022 +0100

    Merge remote-tracking branch 'refs/remotes/origin/main'
```

6

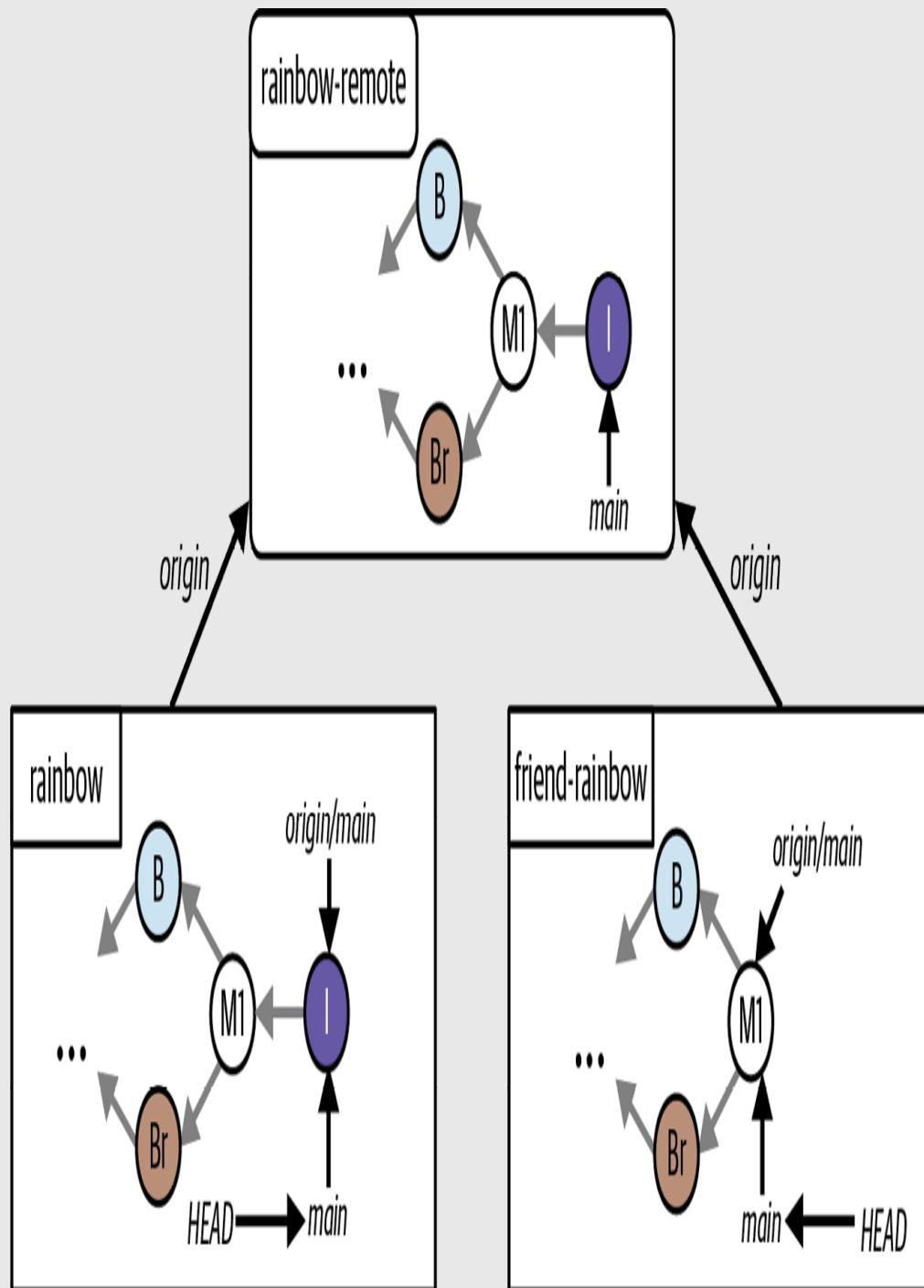
Go to the `rainbow-remote` repository on your hosting service and refresh the page. You should see the indigo commit there.

What to notice:

- In the `rainbow` repository and in the `rainbow-remote` repository, there is an indigo commit.

These changes are illustrated in [Visualize it 10-3](#).

[VISUALIZE IT 10-3]



The Rainbow project after you make the indigo commit in your `rainbow` repository and push the updated `main` branch to the remote repository

Now, to create a situation where you will have a merge conflict, in [Follow Along 10-2](#) your friend will add the color violet to the same line in the `rainbowcolors.txt` file and make a commit without first pulling the changes you made from the `rainbow-remote` repository into their local `main` branch.

[FOLLOW ALONG 10-2]

1 In the the `rainbowcolors.txt` file in the `friend-rainbow` project directory in your text editor, add “Violet is the seventh color of the rainbow.” on line 6. Save the file.

2 `friend-rainbow $ git add rainbowcolors.txt`

3 `friend-rainbow $ git commit -m "violet"`
[main 6ad5c15] violet
1 file changed, 2 insertions(+), 1 deletion(-)

4 `friend-rainbow $ git log`
commit 6ad5c15f033b68ad27f2c9bce8bfa93329b3c23e (`HEAD -> main`)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date: Sun Feb 20 08:41:25 2022 +0100

violet

commit 225839938563c7458af81daca7beb782dfcbfb27 (`origin/main`,
`origin/HEAD`)
Merge: 342bbfc 7f0a87a

Author: annaskoulikari <gitlearningjourney@gmail.com>
Date: Sat Feb 19 13:05:46 2022 +0100

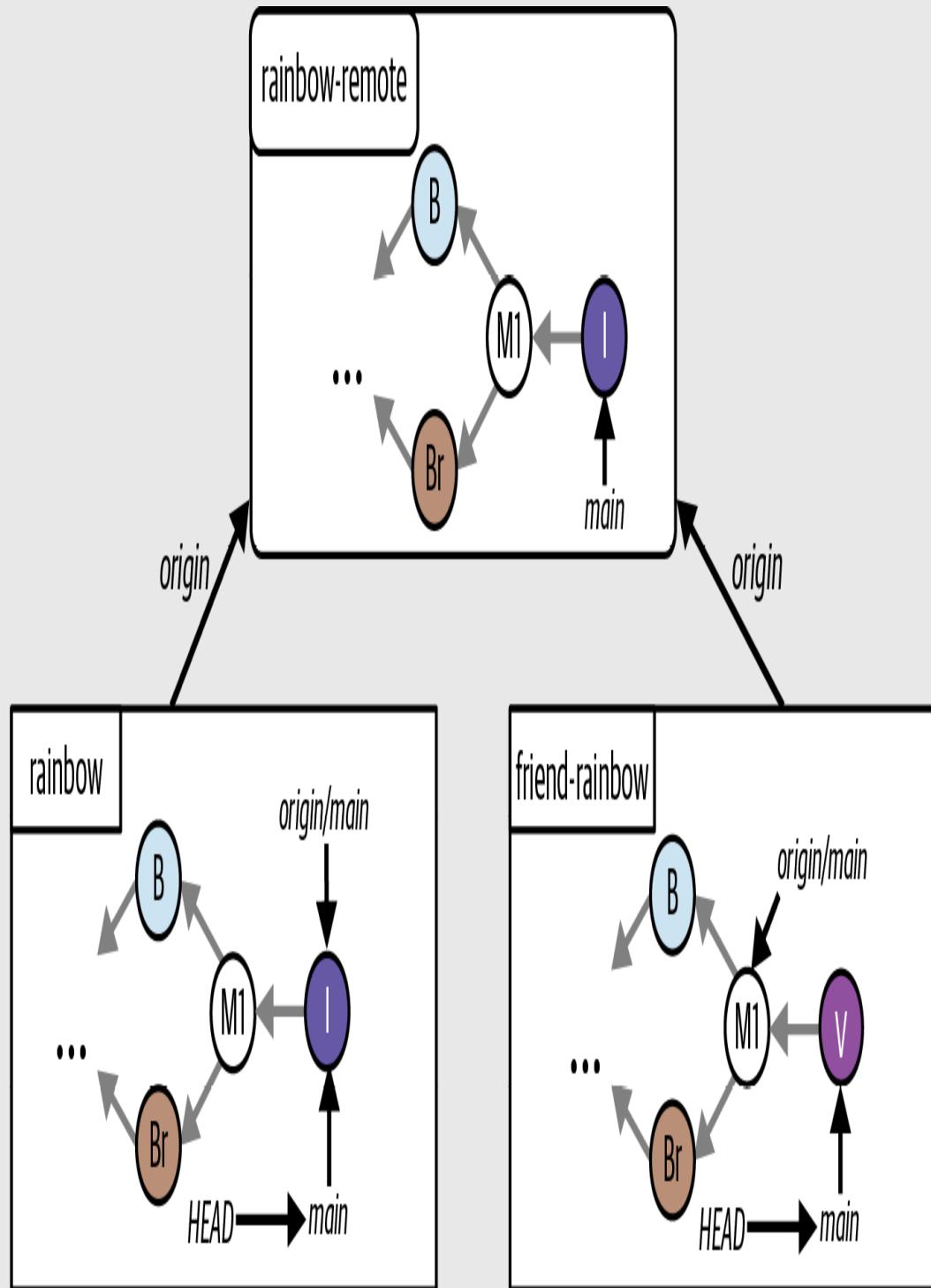
Merge remote-tracking branch 'refs/remotes/origin/main'

What to notice:

- Your friend made the violet commit.

This is illustrated in [Visualize it 10-4](#).

[VISUALIZE IT 10-4]



The Rainbow project after your friend makes the violet commit in the `friend-rainbow` repository without first pulling the latest work on the `main` branch in the remote

repository

What to notice:

- In the `friend-rainbow` repository, the local `main` branch points to the violet commit.
- In the `rainbow` repository, the local `main` branch points to the indigo commit.

At this point, your friend will need to fetch your changes from the remote repository and integrate them before they can push their changes to the remote repository. As you learned in [“Pushing to the Remote Repository” on page 129](#), they can use the `git status` command after fetching your changes to check whether their local `main` branch has diverged from the remote `main` branch. Go to [Follow Along 10-3](#) to see this in action.

[FOLLOW ALONG 10-3]

1 friend-rainbow \$ **git fetch**

```
remote: Enumerating objects: 5, done.  
remote: Counting objects: 100% (5/5), done.  
remote: Compressing objects: 100% (2/2), done.  
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0  
Unpacking objects: 100% (3/3), 322 bytes | 53.00 KiB/s, done.  
From github.com:gitlearningjourney/rainbow-remote  
 c5941f8..6f2ea44  main      -> origin/main
```

2 friend-rainbow \$ **git status**

```
On branch main  
Your branch and 'origin/main' have diverged,  
and have 1 and 1 different commits each, respectively.  
(use "git pull" to merge the remote branch into yours)  
nothing to commit, working tree clean
```

3 friend-rainbow \$ **git log --all**

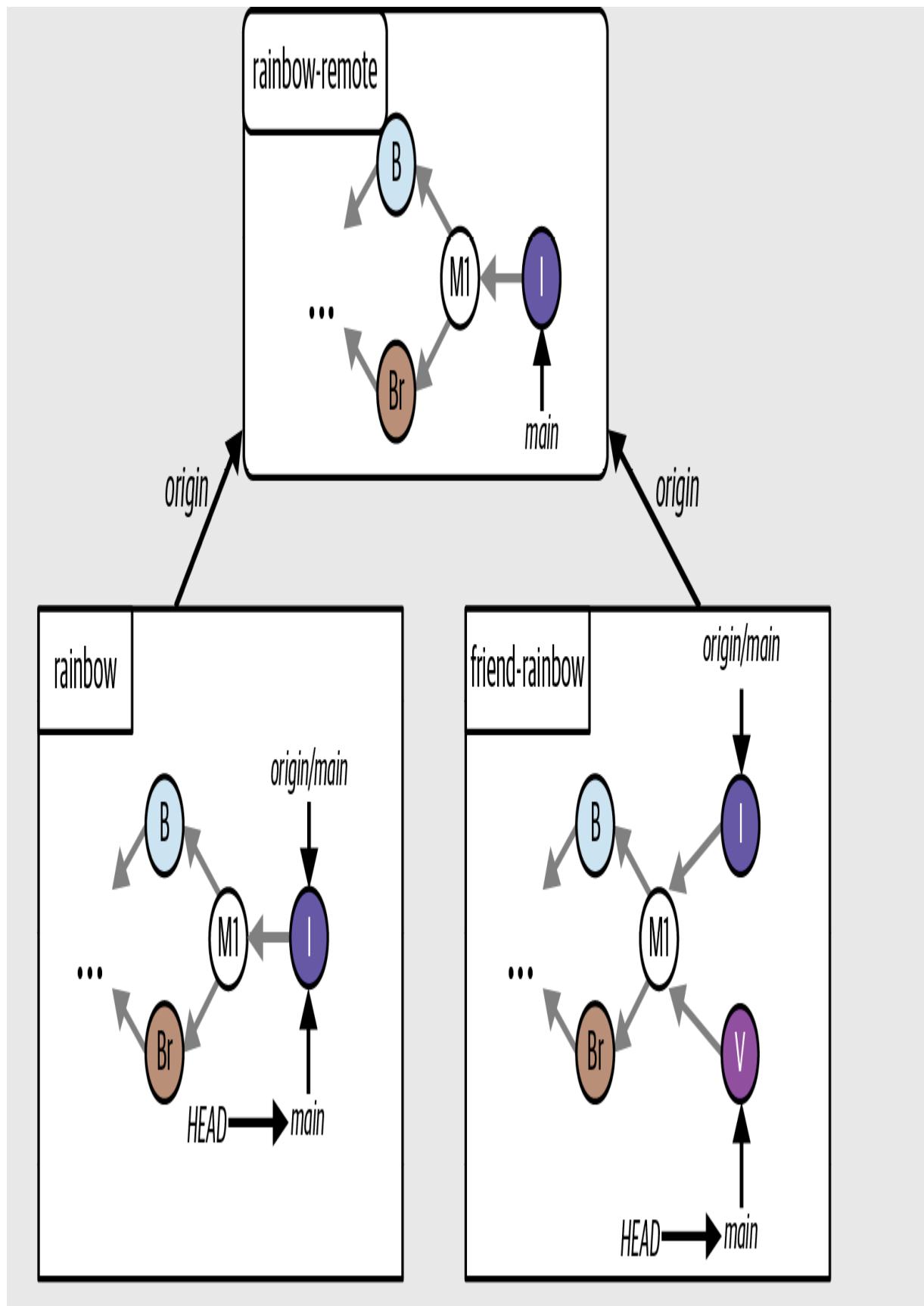
```
commit 6ad5c15f033b68ad27f2c9bce8bfa93329b3c23e (HEAD -> main)  
Author: annaskoulikari <gitlearningjourney@gmail.com>  
Date:   Sun Feb 20 08:41:25 2022 +0100  
    violet  
commit 9b0a61461c8e8d74ed358e65b2662e3697b94de6 (origin/main,  
origin/HEAD)  
Author: annaskoulikari <gitlearningjourney@gmail.com>  
Date:   Sun Feb 20 08:36:11 2022 +0100  
    indigo
```

What to notice:

- In step 2, the output of the `git status` command states that `Your branch` and '`origin/main`' have diverged, and have 1 and 1 different commits each, respectively.
- In the `friend-rainbow` repository, the `origin/main` remote-tracking branch points to the `indigo` commit; however, the remote changes have not yet been merged into the local `main` branch.

These observations are shown in [Visualize it 10-5](#).

[VISUALIZE IT 10-5]



The Rainbow project after your friend fetches the changes from the remote repository to the `friend-rainbow` repository

Next, your friend is going to carry out a three-way merge to integrate the latest changes on the remote `main` branch into their local `main` branch so they can push the updated `main` branch to the remote repository. Before they perform the merge and resolve the merge conflicts in the `friend-rainbow` repository, let's take a closer look at what that process entails.

The Merge Conflict Resolution Process

As mentioned earlier in this chapter, there are two steps to resolving merge conflicts. In this section we'll walk through them in more detail. As a reminder, the first step is to decide what to keep, edit the content, and remove the conflict markers, and the second is to add all the changes to the staging area and commit. Let's go over each of them in turn.

STEP 1

After you execute the `git merge` command in a situation where a merge conflict arises, Git will identify the conflict and will insert conflict markers indicating the location(s) of the conflicting content. The first step in resolving merge conflicts is to decide what to keep, edit the content in the file(s) where the conflicts occur, and remove the conflict markers. Once you're done making the necessary changes, you must also make sure to save the file or files in your text editor. In the Rainbow project there will be a merge conflict in only one file, but in other Git projects you work on there may be merge conflicts in multiple files.

To resolve the merge conflict in the Rainbow project, your friend will keep both the sentence about the indigo color and the sentence about the violet color. They want to make sure they’re in the correct order, so they will put the sentence about indigo first (which will end up being on line 6) and the sentence about violet second (this will end up on line 7). See [Figure 10-2](#) for an example of what the `rainbowcolors.txt` file will look like before and after your friend makes this change.

Before	After
<p>rainbowcolors.txt</p> <p>Red is the first color of the rainbow. Orange is the second color of the rainbow. Yellow is the third color of the rainbow. Green is the fourth color of the rainbow. Blue is the fifth color of the rainbow.</p> <p><<<<< HEAD</p> <p>Violet is the seventh color of the rainbow.</p> <p>=====</p> <p>Indigo is the sixth color of the rainbow.</p> <p>>>>> refs/remote/origin/main</p>	<p>rainbowcolors.txt</p> <p>Red is the first color of the rainbow. Orange is the second color of the rainbow. Yellow is the third color of the rainbow. Green is the fourth color of the rainbow. Blue is the fifth color of the rainbow.</p> <p><<<<< HEAD</p> <p>Indigo is the sixth color of the rainbow. Violet is the seventh color of the rainbow.</p> <p>=====</p> <p>>>>> refs/remote/origin/main</p>

FIGURE 10-2

The `rainbowcolors.txt` file before and after your friend decides what to keep and edits the content

Next, your friend will need to remove the conflict markers from the `rainbowcolors.txt` file, as seen in [Figure 10-3](#), and save the file.

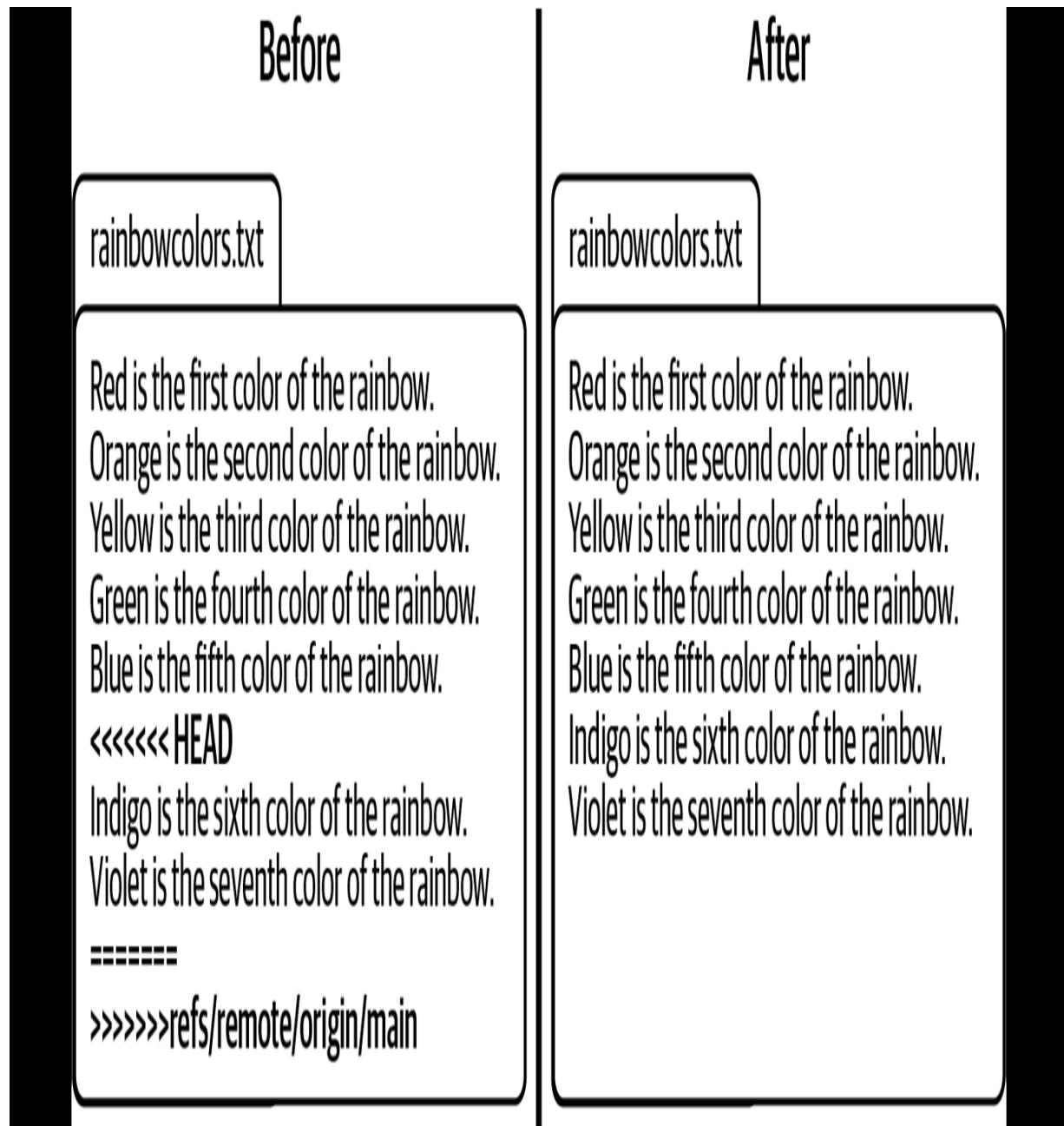


FIGURE 10-3

The `rainbowcolors.txt` file before and after your friend removes the conflict markers

STEP 2

After your friend has finished editing the content, they're ready for the second step: adding the updated file(s) to the staging area and making a commit. In the Rainbow project there is only one file with a merge conflict, so they'll need to add only one file to the staging area. However, as mentioned previously, it's common to have merge conflicts in multiple files, in which case you have to add all the modified files to the staging area.

In [Figure 10-4](#), you can see an example of what the commit history in the `friend-rainbow` repository will look like before and after your friend completes the merge. This merge will result in the M2 merge commit, because this is a three-way merge.

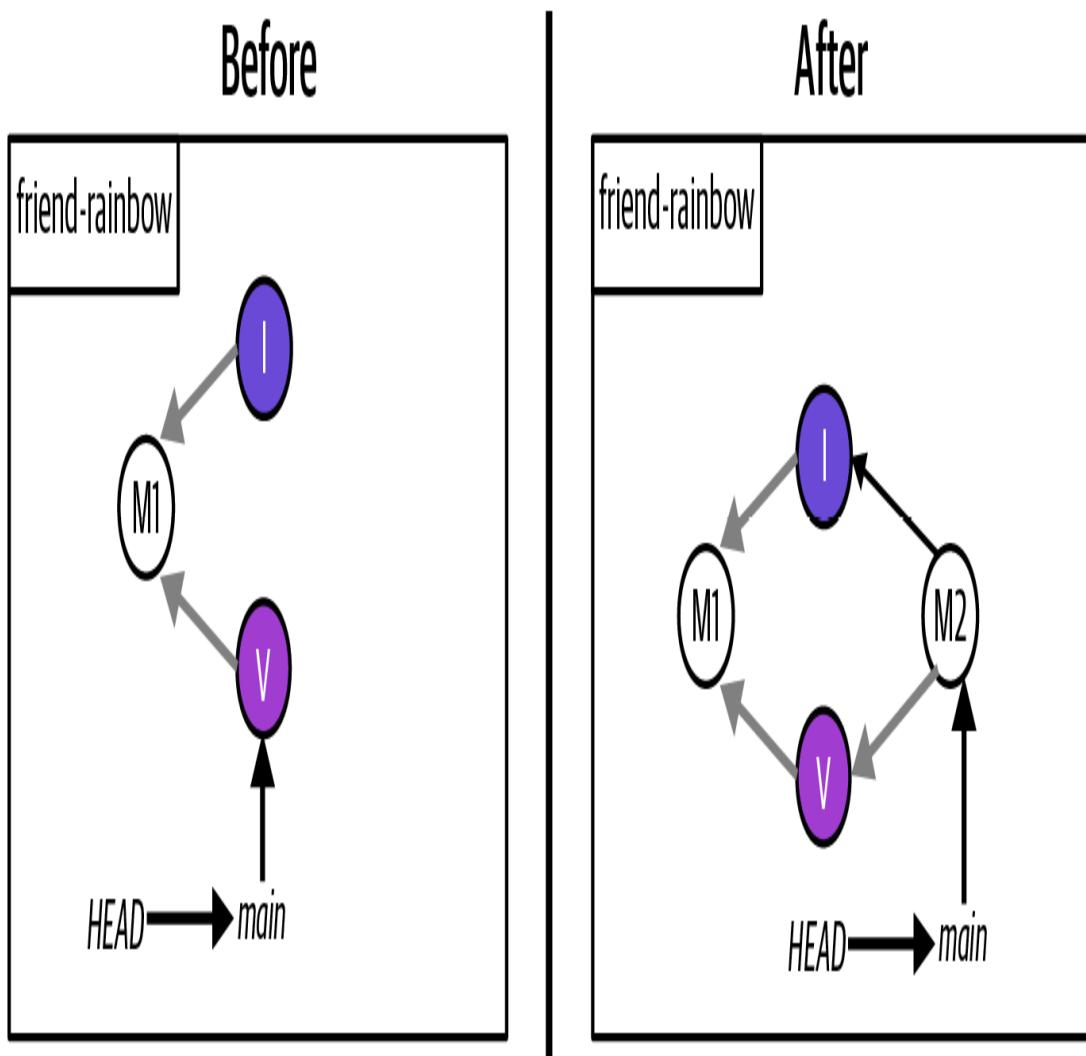


FIGURE 10-4

The commit history before and after your friend completes the three-way merge

Now that you're familiar with the two steps involved in resolving merge conflicts, let's briefly go over what to do if you decide you don't want to continue with a merge.

ABORTING A MERGE

If at any point during a merge with conflicts you decide that you don't want to continue integrating two branches, you can choose to stop or "abort" the merge by using the `git merge` command with the `--abort` option. This will return all your files to the state they were in before the merge.

[SAVE THE COMMAND]

`git merge --abort`

Stop the merge process and go back to the state before the merge

Now that you know how to resolve merge conflicts and what to do if you ever want to abort a merge and the conflict resolution process, let's see an example of resolving merge conflicts in practice.

Resolving Merge Conflicts in Practice

In [Follow Along 10-4](#), your friend is going to carry out the three-way merge and resolve the merge conflicts. Throughout the merge operation, they will use the `git status` command to see information about the conflict resolution process.

[FOLLOW ALONG 10-4]

1 friend-rainbow \$ **git merge origin/main**

Auto-merging rainbowcolors.txt

CONFLICT (content): Merge conflict in rainbowcolors.txt

Automatic merge failed; fix conflicts and then commit the result.

2 friend-rainbow \$ **git status**

On branch main

Your branch and 'origin/main' have diverged,

and have 1 and 1 different commits each, respectively.

(use "git pull" to merge the remote branch into yours)

You have unmerged paths.

(fix conflicts and run "git commit")

(use "git merge --abort" to abort the merge)

Unmerged paths:

(use "git add <file>..." to mark resolution)

both modified: rainbowcolors.txt

no changes added to commit (use "git add" and/or "git commit -a")

3 Take a deep breath, go to the `friend-rainbow` project directory in your text editor, and look at the `rainbowcolors.txt` file. Find the conflict markers and the conflicting content.

4 Carry out step 1 of resolving merge conflicts, which is to choose what to keep, edit the content, and remove the conflict markers. In your case, you are going to keep all of the content; that is, the changes from violet commit and the changes from the indigo commit. The comment about the color indigo will go on line 6, and the comment about the color violet will go on line 7. Make sure to save the `rainbowcolors.txt` file when you're done making your edits.

[FOLLOW ALONG 10-4]

5

```
friend-rainbow $ git add rainbowcolors.txt
```

6

```
friend-rainbow $ git status
```

On branch main

Your branch and 'origin/main' have diverged,
and have 1 and 1 different commits each, respectively.

(use "git pull" to merge the remote branch into yours)

All conflicts fixed but you are still merging.

(use "git commit" to conclude merge)

Changes to be committed:

modified: rainbowcolors.txt

7

```
friend-rainbow $ git commit -m "merge commit 2"
```

[main f10f972] merge commit 2

[FOLLOW ALONG 10-4]

8

```
friend-rainbow $ git log
commit f10f9725e3319af840a3d891ca8950436a219eb0 (HEAD -> main)
Merge: 6ad5c15 9b0a614
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sun Feb 20 09:11:06 2022 +0100

    merge commit 2

commit 6ad5c15f033b68ad27f2c9bce8bfa93329b3c23e
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sun Feb 20 08:41:25 2022 +0100

    violet

commit 9b0a61461c8e8d74ed358e65b2662e3697b94de6 (origin/main,
origin/HEAD)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sun Feb 20 08:36:11 2022 +0100

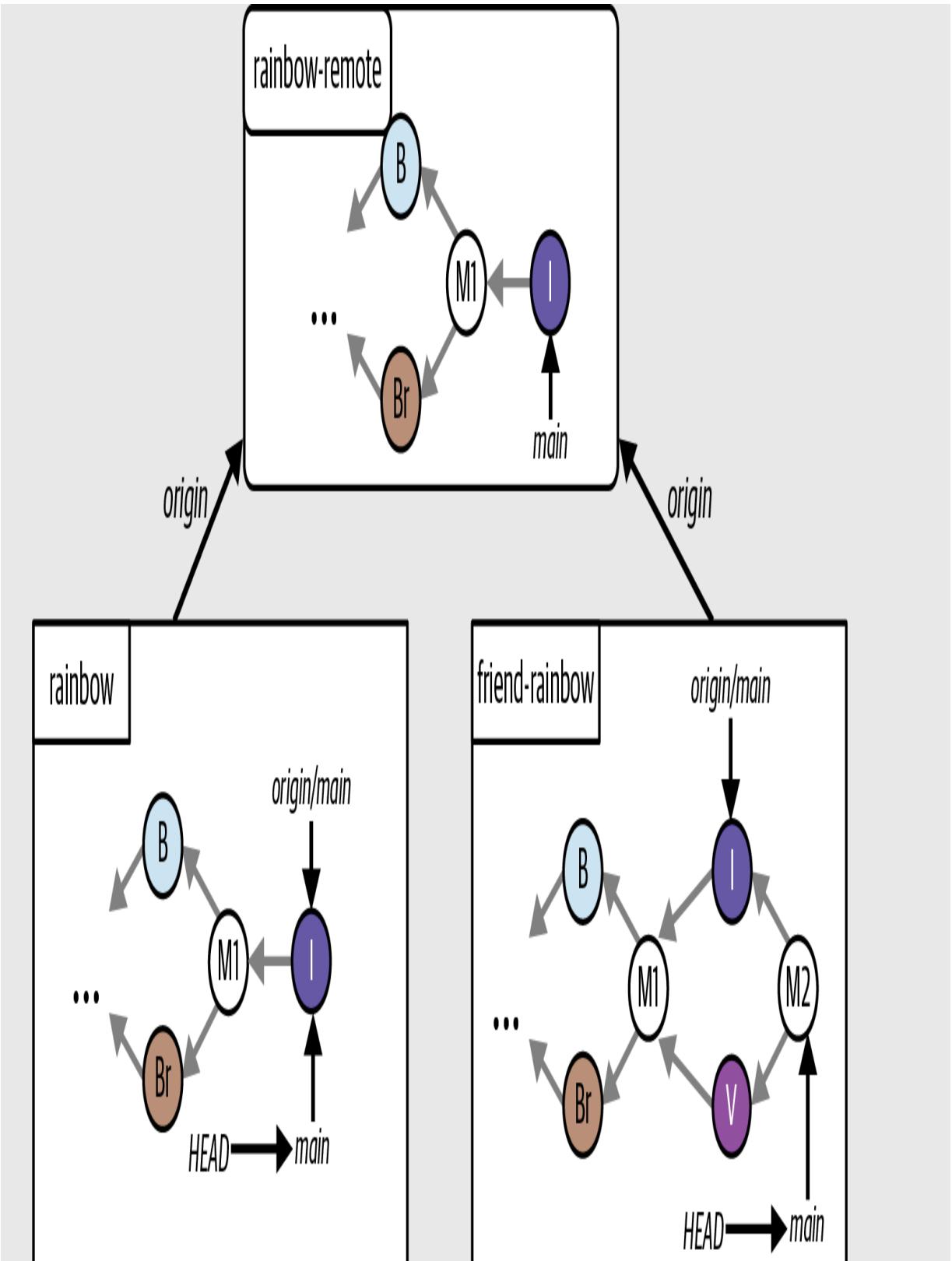
    indigo
```

What to notice:

- In step 8, the `git log` output shows:
 - The local `main` branch points to merge commit 2.
 - The two parent commits of merge commit 2 are `6ad5c15` (the violet commit) and `9b0a614` (the indigo commit). Recall that the commit hashes in your `friend-rainbow` repository will be different from the ones in this book because commit hashes are unique.

These observations are illustrated in [Visualize it 10-6](#), where the new merge commit is represented as M2.

[VISUALIZE IT 10-6]



The Rainbow project after your friend resolves the merge conflicts in the `friend-`

`rainbow` repository and makes another merge commit

Your friend successfully carried out the three-way merge and resolved the merge conflicts. Next, let's discuss the importance of staying up to date with a remote repository.

Staying Up to Date with a Remote Repository

It is more time-consuming to integrate changes from one branch into another when there are merge conflicts. In the Rainbow project example, there was only one file with a small merge conflict. However, in real-world projects there may be many more files with way more complicated merge conflicts to resolve. Staying up to date with changes made on relevant branches in a remote repository is important because it can help limit the number of potential merge conflicts you may have to resolve later down the road.

Whenever you create a new branch, it is recommended that you base it off the most up-to-date version of the relevant remote branch in the remote repository. Often this will be the `main` branch (or whichever branch your team uses as the primary line of development). However, depending on your team's Git workflow, it may be another branch.

If you're working on an existing branch on your own, it is recommended to update your branch with the changes made to the relevant remote branch in the remote repository by merging. And if you find yourself in a situation where you're working on the same branch as someone else, then you must always make sure to fetch and merge any changes that have been made on that branch in the remote repository before continuing to work on it.

With this reminder about staying up to date in mind, let's now make sure that your friend pushes the work they've done on their local `main` branch to the remote repository and that you sync your local `main` branch in the `rainbow` repository with the remote `main` branch.

Syncing the Repositories

For all the repositories to be in sync, your friend will need to push the new commits on their local `main` branch to the remote repository, and you will need to pull down all the changes into your local `main` branch in the `rainbow` repository. Go to [Follow Along 10-5](#) to complete these steps now.

[FOLLOW ALONG 10-5]

1

```
friend-rainbow $ git push
Enumerating objects: 10, done.

Counting objects: 100% (10/10), done.

Delta compression using up to 4 threads

Compressing objects: 100% (6/6), done.

Writing objects: 100% (6/6), 614 bytes | 614.00 KiB/s, done.

Total 6 (delta 2), reused 0 (delta 0), pack-reused 0

remote: Resolving deltas: 100% (2/2), completed with 1 local object.

To github.com:gitlearningjourney/rainbow-remote.git

  9b0a614..f10f972 main -> main
```

2

```
friend-rainbow $ git log
commit f10f9725e3319af840a3d891ca8950436a219eb0 (HEAD -> main,
origin/main, origin/HEAD)

Merge: 6ad5c15 9b0a614

Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sun Feb 20 09:11:06 2022 +0100

    merge commit 2

commit 6ad5c15f033b68ad27f2c9bce8bfa93329b3c23e

Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sun Feb 20 08:41:25 2022 +0100

    violet
```

3

Go to the command line window where you're in the `rainbow` repository to execute the following command.

[FOLLOW ALONG 10-5]

4 rainbow \$ **git pull**

```
remote: Enumerating objects: 10, done.
remote: Counting objects: 100% (10/10), done.
remote: Compressing objects: 100% (4/4), done.
Unpacking objects: 100% (6/6), 594 bytes | 99.00 KiB/s, done.
remote: Total 6 (delta 2), reused 6 (delta 2), pack-reused 0
From github.com:gitlearningjourney/rainbow-remote
   9b0a614..f10f972  main      -> origin/main
Updating 9b0a614..f10f972
Fast-forward
  rainbowcolors.txt | 5 +----
  1 file changed, 4 insertions(+), 1 deletion(-)
```

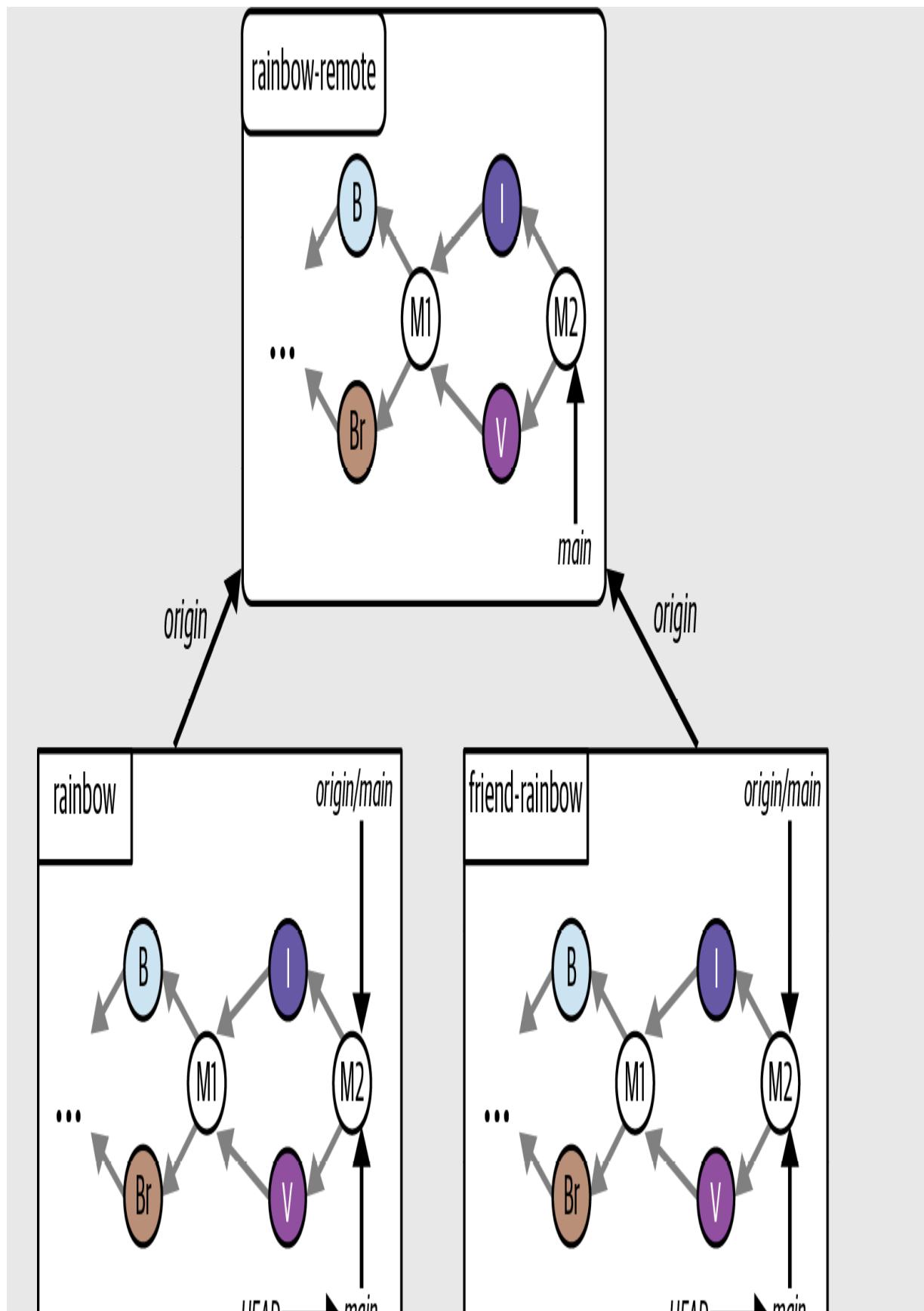
5 rainbow \$ **git log**

```
commit f10f9725e3319af840a3d891ca8950436a219eb0 (HEAD -> main,
origin/main)
Merge: 6ad5c15 9b0a614
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sun Feb 20 09:11:06 2022 +0100
       merge commit 2

commit 6ad5c15f033b68ad27f2c9bce8bfa93329b3c23e
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sun Feb 20 08:41:25 2022 +0100
       violet
```

All three repositories in the Rainbow project are now in sync, as illustrated in [Visualize it 10-7](#).

[VISUALIZE IT 10-7]





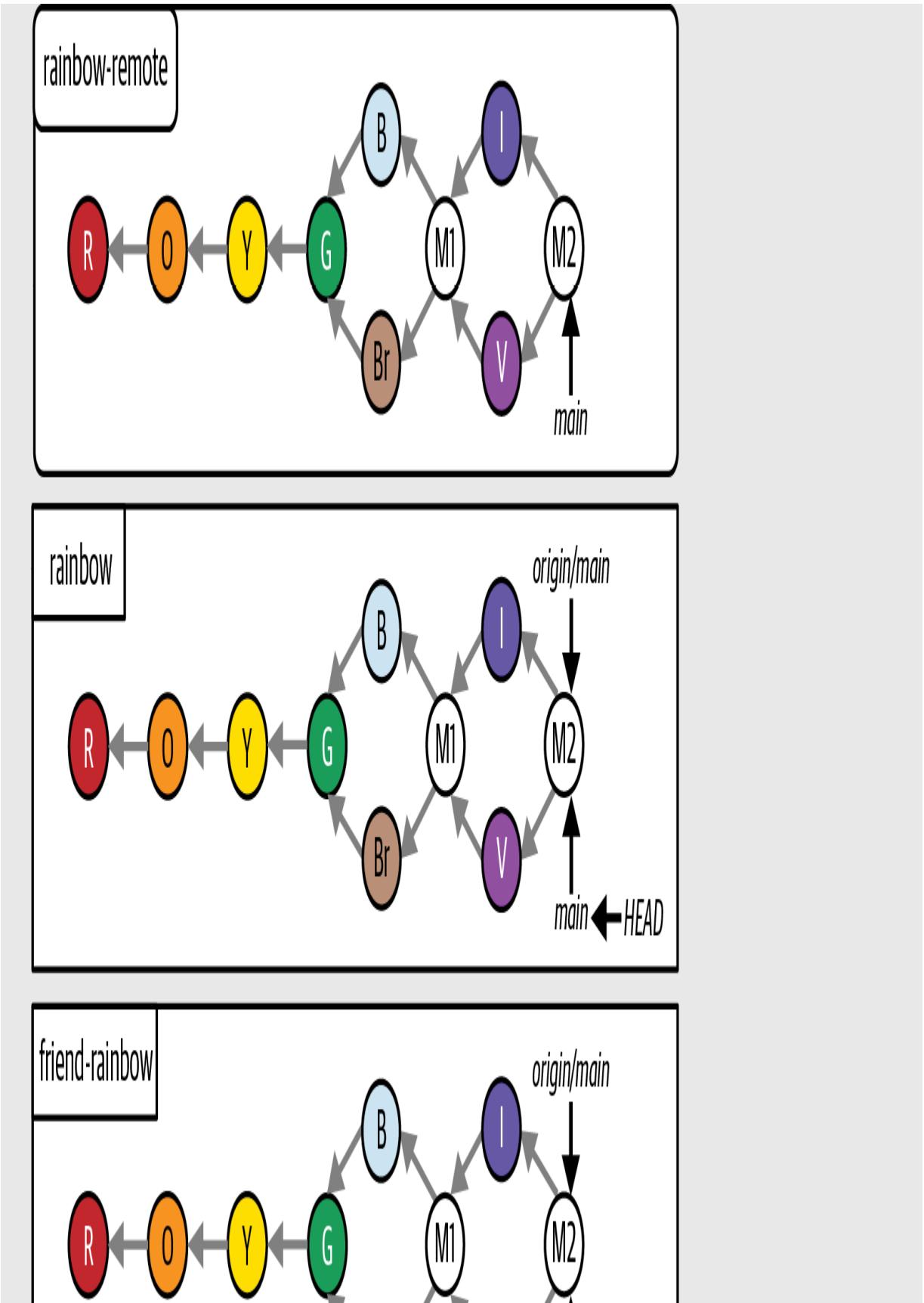
What to notice:

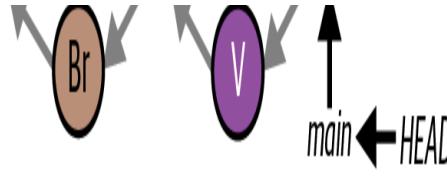
- In the `rainbow-remote` repository the remote `main` branch points to the M2 merge commit.
- In the `rainbow` repository, the local `main` branch and the `origin/main` remote-tracking branch point to the M2 merge commit.
- In the `friend-rainbow` repository, the `origin/main` remote-tracking branch has updated to point to the M2 merge commit.

State of the Local and Remote Repositories

[Visualize it 10-8](#) shows the state of the local and remote repositories in the Rainbow project with all the commits that were made from [Chapter 1](#) all the way to the end of this chapter.

[VISUALIZE IT 10-8]





The Rainbow project at the end of [Chapter 10](#) with all the commits since [Chapter 1](#)

Summary

In this chapter you learned about merge conflicts, which arise when you merge two branches where different changes have been made to the same parts in the same file(s), or if in one branch a file was deleted that was edited in the other branch. You also learned about the two steps involved in resolving them: deciding what content to keep, removing the conflict markers, and saving the files, then adding the edited files to the staging area and making a commit.

Merge conflicts can arise either during a three-way merge or during the process of rebasing. Up until now we have focused on merging; in [Chapter 11](#) you will learn about rebasing and you will work through a hands-on example in the Rainbow project.

Rebasing

As you know, Git offers two main ways to integrate changes from one branch to another: merging and rebasing. In Chapters [5](#) and [9](#) you learned about fast-forward merges and three-way merges, respectively. In [Chapter 10](#), you learned about merge conflicts and found out that they can arise both during a three-way merge and during a rebase.

In this chapter, you will learn about rebasing and walk through an example. I'll introduce the five stages of the rebase process and show you how to resolve merge conflicts during this process. We'll look at how merging and rebasing differ, and in which cases you may prefer to rebase instead of merge. I'll also introduce the golden rule of rebasing, which will help you determine when not to rebase. Additionally, you will learn a bit more about the staging area and how to use it as a rough draft space for organizing what you want to include in your next commit.

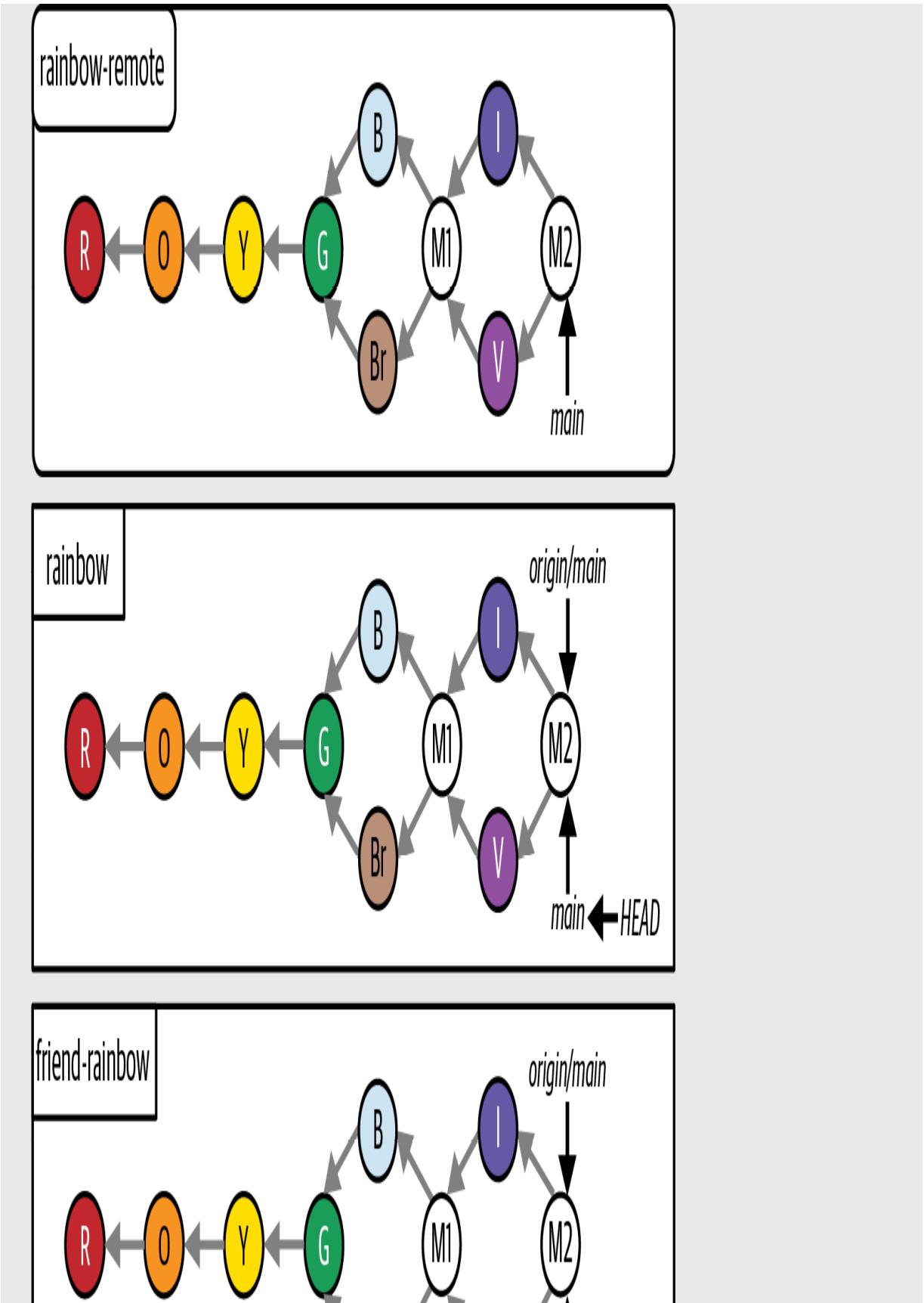
State of the Local and Remote Repositories

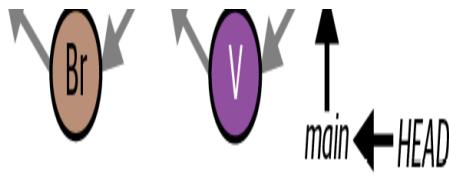
At the start of this chapter, you should have two local repositories called `rainbow` and `friend-rainbow` and one remote repository called `rainbow-remote`. All

three of these repositories should contain the same commits and branches. As usual, when working through the examples in this chapter I recommend that you use two separate text editor windows and command line windows for the `rainbow` repository and the `friend-rainbow` repository.

[Visualize it 11-1](#) shows the state of the local and remote repositories in the Rainbow project with all the commits that were made from [Chapter 1](#) to [Chapter 10](#).

[VISUALIZE IT 11-1]

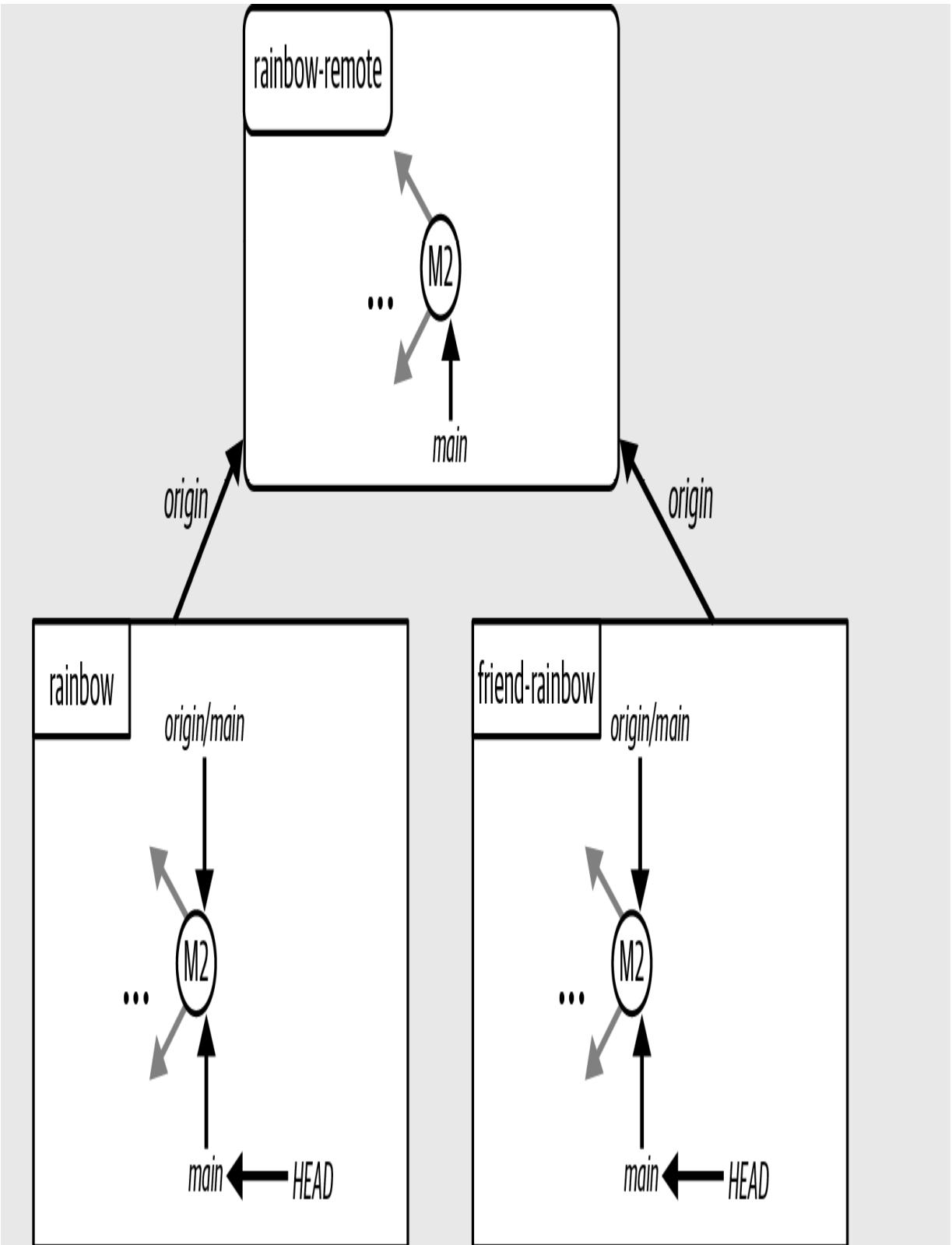




The Rainbow project at the start of [Chapter 11](#) with all the commits since [Chapter 1](#)

To focus on the commits you are going to make in this chapter, from here on I will simplify the Visualize It diagrams and show only the last commit that was made on the `main` branch in all the repositories, which is the M2 merge commit. This representation is shown in [Visualize it 11-2](#).

[VISUALIZE IT 11-2]



A simplified representation of the Rainbow project at the start of [Chapter 11](#), showing

just the last commit on the `main` branch in all the repositories

Integrating Changes in Git

Up until now, we have focused on merging as a way of integrating changes in Git. You have seen in practice that a fast-forward merge simply moves the branch pointer of the target branch to point to the latest commit, whereas a three-way merge creates a merge commit that ties the development histories of the source branch and the target branch together (and may sometimes also lead to merge conflicts).

If you look at the state of the various repositories in [Visualize it 11-1](#), you can see that up until the green commit you had a linear project history, but after the green commit the commit history is nonlinear due to the merge commits created by the three-way merges. Some teams and individuals prefer to maintain a linear project history because they find it is more organized and simpler. You can use the process of rebasing to avoid three-way merges and merge commits and maintain a linear project history.

Rebasing takes all the work you have done in the commits on one branch and reapplies the work on another branch, creating entirely new commits. This can make it appear as though you have created a branch from an entirely different commit than the original commit you created it from.

To carry out a rebase, you need to be on the branch you want to rebase. You use the `git rebase` command and pass in the name of the branch that you want to rebase onto.

[SAVE THE COMMAND]

git rebase <branch_name>

Reapply commits on top of another branch

Given that rebasing creates entirely new commits, this means it changes the commit history. It's important to be careful when using Git operations that do this. We'll discuss situations where it is not recommended to rebase a branch in [“The Golden Rule of Rebasing” on page 223](#). First, however, I'm going to show you how rebasing may be used in a Git project, and you'll work through a hands-on example in the Rainbow project. Let's get started by exploring why you might want to take this approach.

Why Is Rebasing Helpful?

To illustrate how rebasing can help you avoid three-way merges and maintain a linear project history, let's revisit the example introduced in [Example Book Project 9-1](#). Take a look at [Example Book Project 11-1](#).

Example Book Project 11-1

Using the example scenario from [Chapter 9](#), suppose my coauthor and I each make a different branch based on the `main` branch when it is pointing to commit B, to work on separate chapters. I make the `chapter_five` branch and my coauthor makes the `chapter_six` branch, as seen in [Figure 11-1](#).

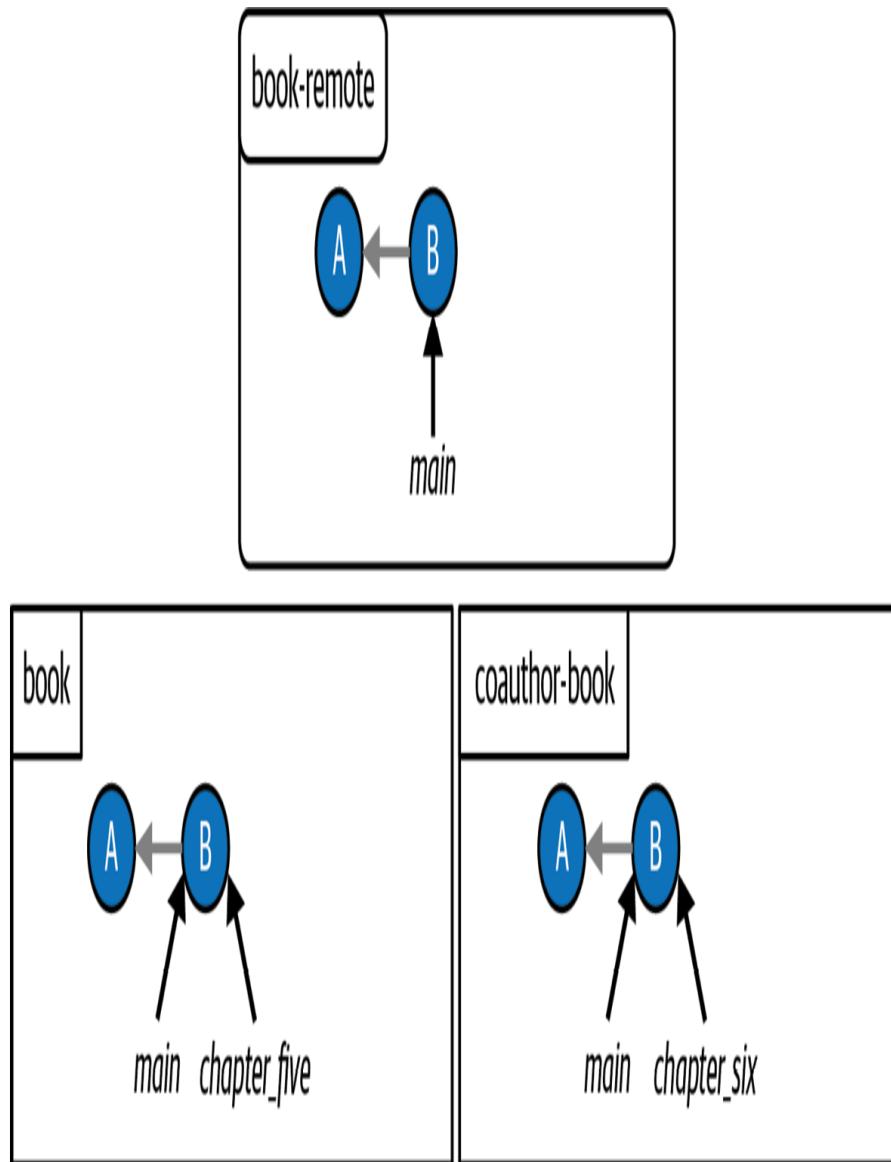


FIGURE 11-1

The Book project when my coauthor and I each make branches to work on different chapters

My coauthor works on chapter 6 of the book and makes commit C. They merge their work into the `main` branch of their local repository and push the updated `main` branch to the remote repository. At the same time, I work on chapter 5 and make commit D in my local repository. However, I have not yet merged or shared my work. These developments are illustrated in [Figure 11-2](#).

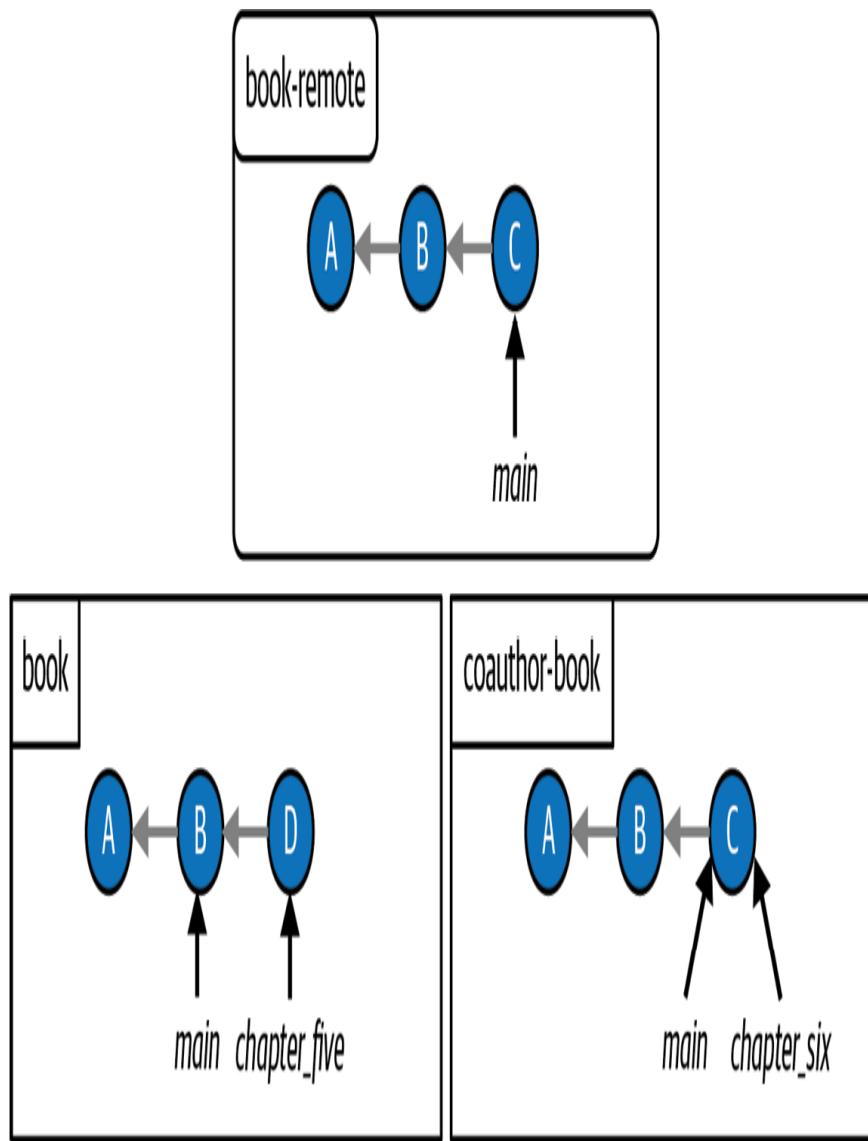


FIGURE 11-2

The Book project after my coauthor pushes their changes to the remote repository

When I decide that my changes to chapter 5 are ready to be merged into my local `main` branch, I now have a couple of options. One option is to fetch the changes on the remote

`main` branch from the remote repository and then do a three-way merge to

merge `chapter_five` into `main`, which will ultimately produce a merge commit (represented as commit M) in [Figure 11-3](#). This is the option presented in [Chapter 9](#).

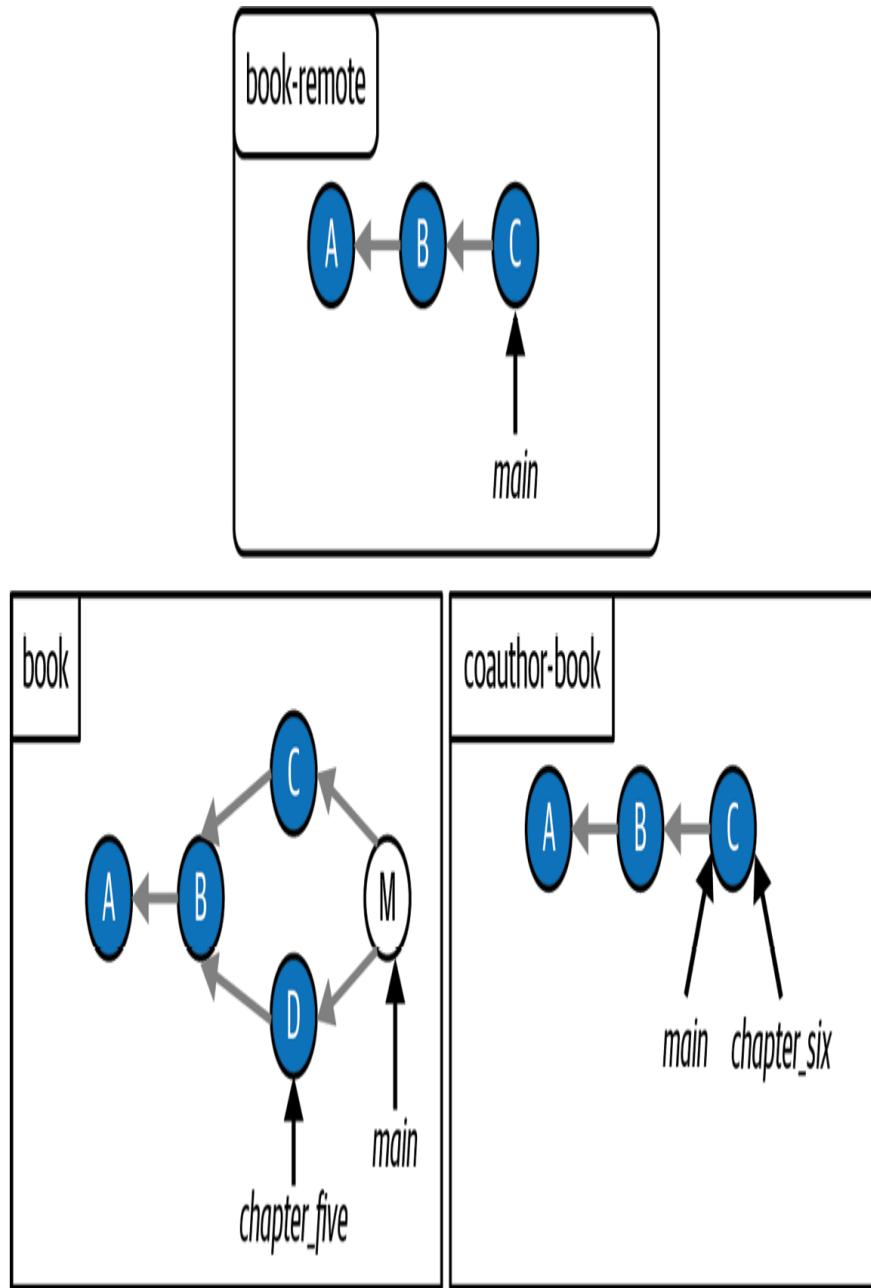


FIGURE 11-3

The Book project if I were to integrate the changes from the remote repository using a

three-way merge

Another option is to pull the changes from the remote repository, which means my local `main` branch will update to point to commit C, with the intent of rebasing my branch afterward. [Figure 11-4](#) shows the state of the Book project after I pull the changes from the remote `main` branch.

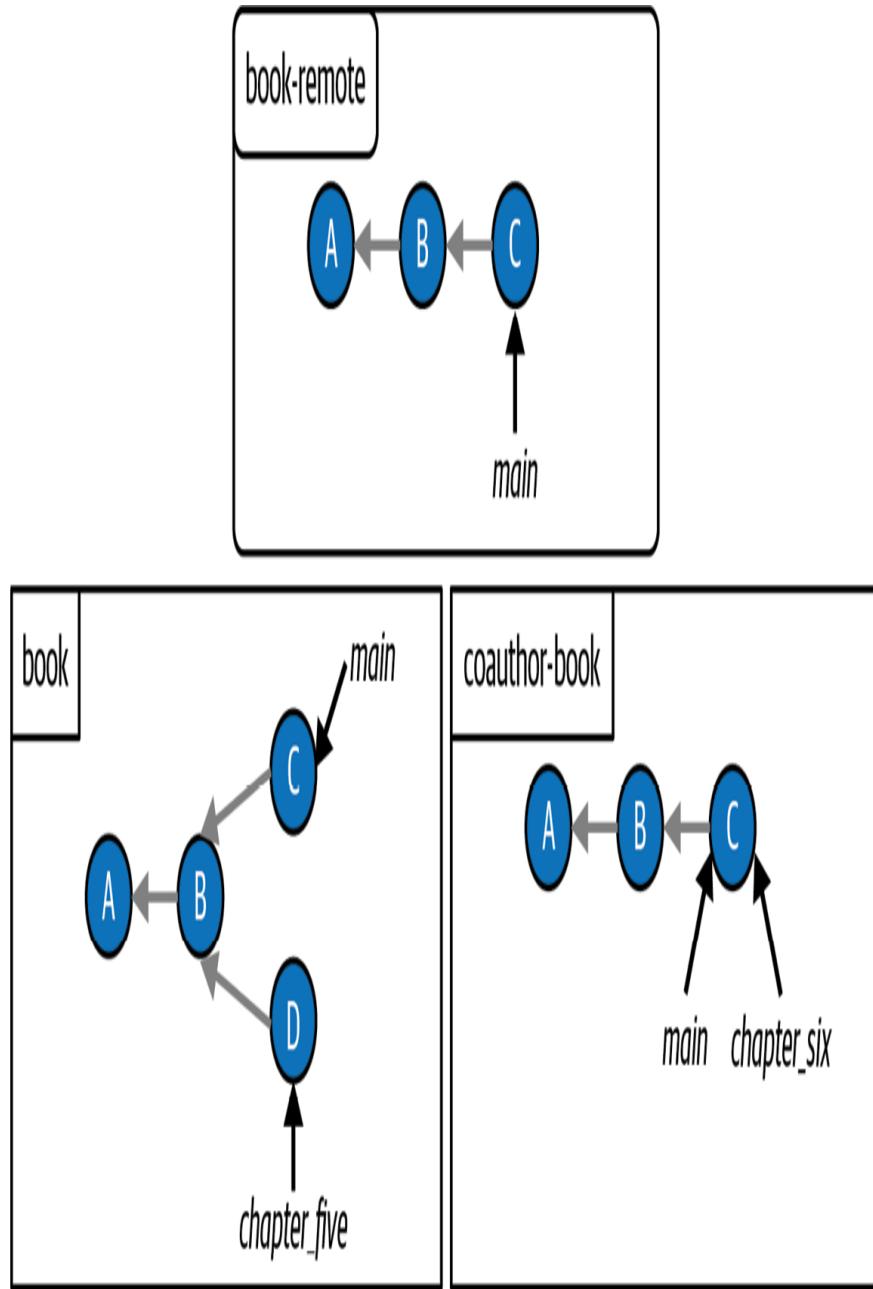


FIGURE 11-4

The Book project after I pull the changes from the remote `main` branch

Next, I can rebase my `chapter_five` branch on top of the updated `main` branch. As mentioned previously, rebasing will take all the commits I

have made on the `chapter_five` branch and reapply them onto the `main` branch, creating entirely new commits. In this example, the only commit that I made on the `chapter_five` branch since it diverged from the `main` branch is commit D. When the branch is rebased, a new D' commit will be created, as seen in [Figure 11-5](#). The apostrophe ('') in the identifier indicates that this is a rebased commit.

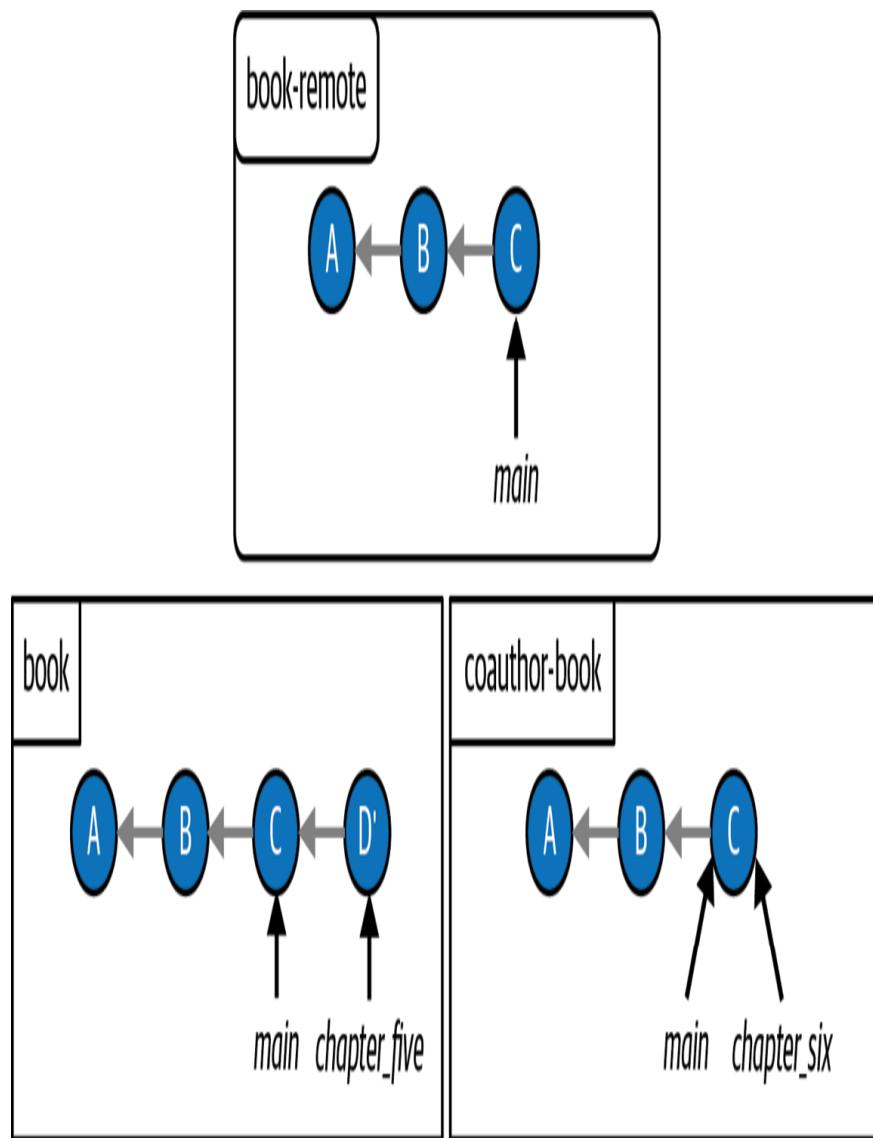


FIGURE 11-5

The Book project after I integrate the changes from the remote repository by rebasing

my `chapter_five` branch onto the `main` branch

The D' commit represents the new version of commit D that was created during the rebase process. It includes all the changes I made in the original D commit, but it is an entirely new commit with a new commit hash.

After I have rebased my `chapter_five` branch, I can merge it into `main` with a simple fast-forward merge, as seen in [Figure 11-6](#).

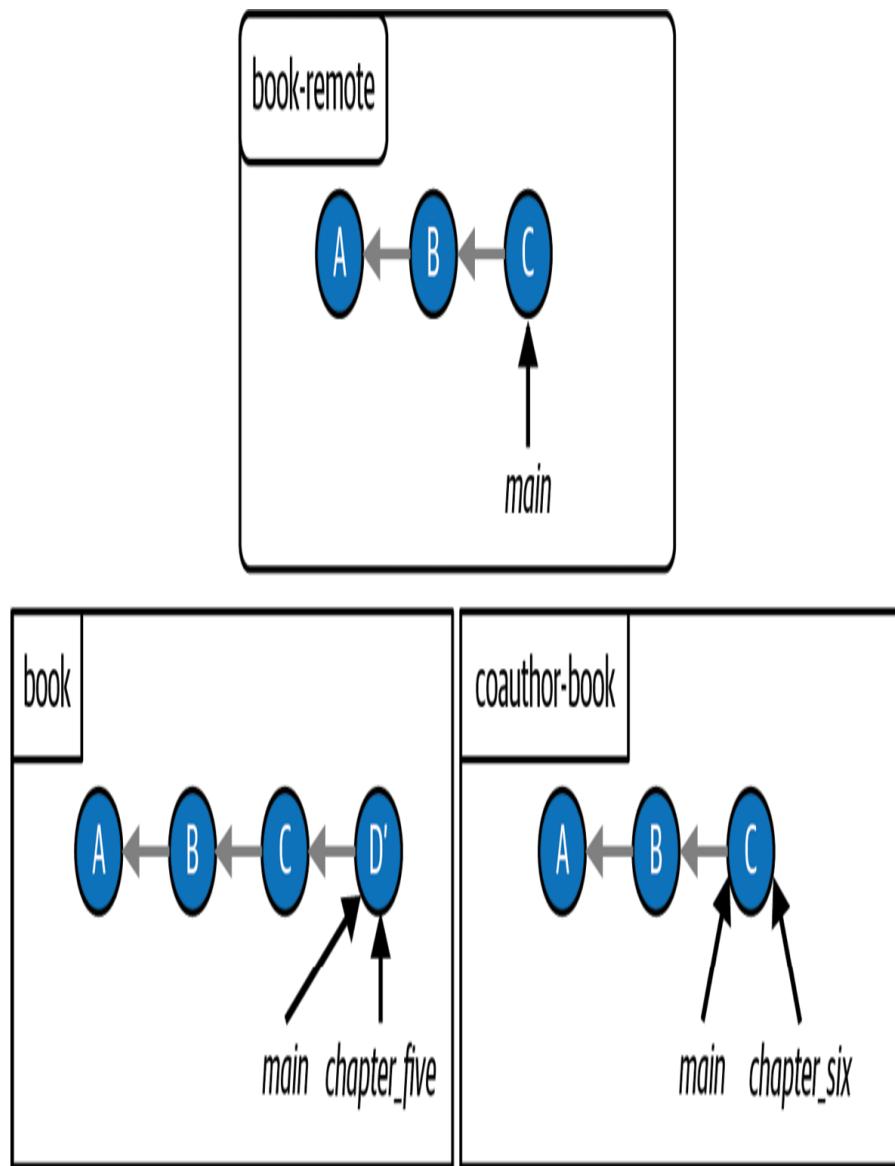


FIGURE 11-6

The Book project after I merge the `chapter_five` branch into the `main` branch with a simple fast-forward merge

By rebasing my branch I'm able to maintain a linear commit history and avoid making additional merge commits.

Note that the original D commit still exists in the commit history; however, it is no longer part of any of the branches in the [book](#) repository, which is why it is not shown in [Figure 11-6](#). Rebasing a branch does not delete the commits on that branch; it simply re-creates them. The old commits still exist in the commit history.

Now that you've seen a hypothetical example of how rebasing can be used to maintain a linear project history, let's create a situation in the Rainbow project where you have divergent development histories between two branches so you can go over a hands-on example.

Setting Up the Rebasing Example

To practice rebasing, you'll need to create divergent histories. You are going to make one commit in the [rainbow](#) repository and push it to the remote repository. Then your friend, without fetching the changes from the remote repository, is going to make two commits in their [friend-rainbow](#) repository. After they have made their commits, they will fetch the changes from the remote repository and decide to rebase their branch.

Go to [Follow Along 11-1](#) to make a commit in the [rainbow](#) repository.

[FOLLOW ALONG 11-1]

1 Open the `othercolors.txt` in the `rainbow` project directory in a text editor window. Add “Gray is not a color in the rainbow.” on line 2, and save the file.

2 `rainbow $ git add othercolors.txt`

3 `rainbow $ git commit -m "gray"`
[main 6f2cf36] gray
1 file changed, 2 insertions(+), 1 deletion(-)

4 `rainbow $ git push`
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 327 bytes | 327.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To `github.com:gitlearningjourney/rainbow-remote.git`
`f10f972..6f2cf36 main -> main`

[FOLLOW ALONG 11-1]

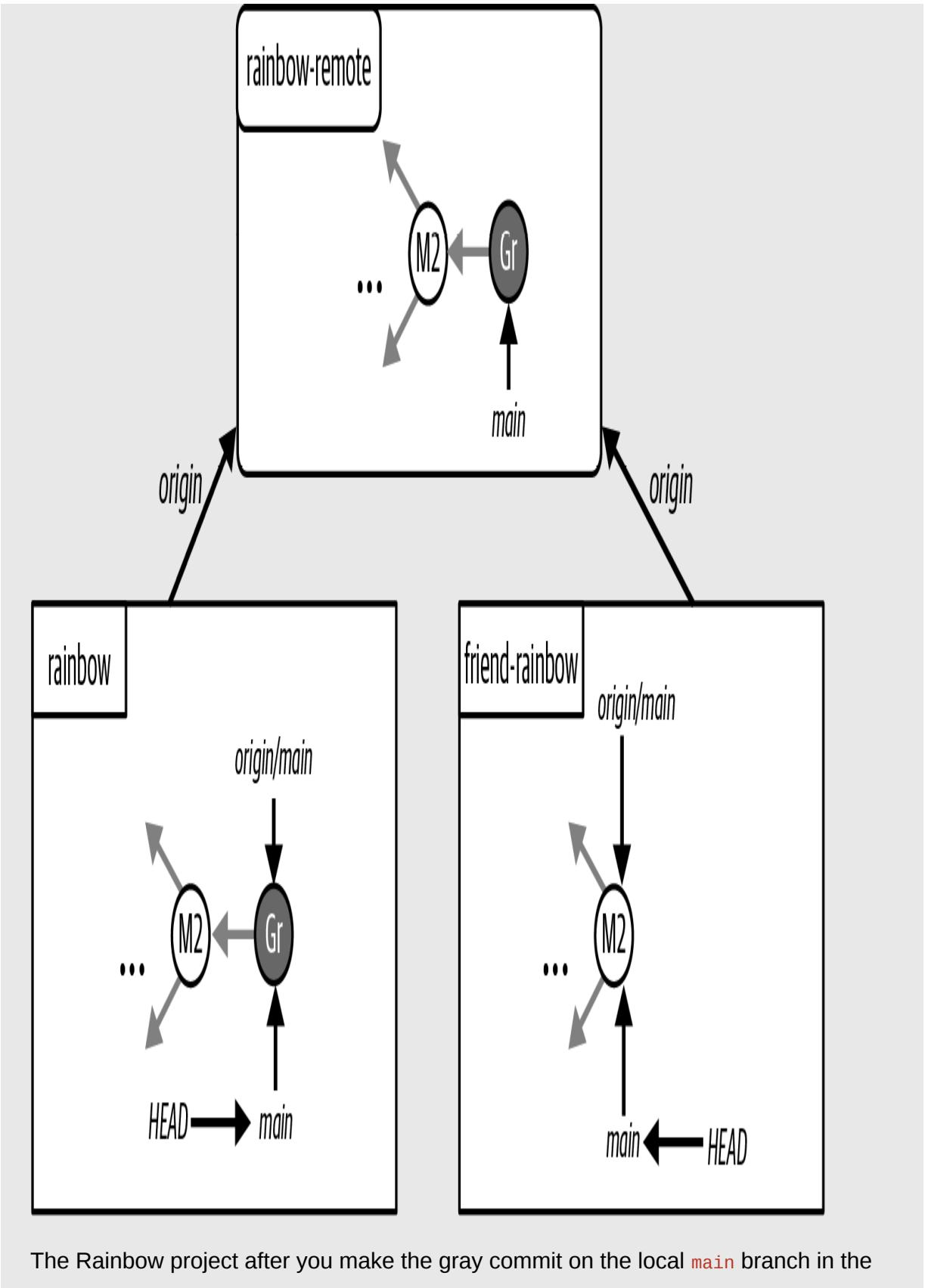
```
rainbow $ git log  
5 commit 6f2cf3698e6bf9078e8e0340ec9948f590405091 (HEAD -> main,  
origin/main)  
Author: annaskoulikari <gitlearningjourney@gmail.com>  
Date:   Sun Feb 20 09:27:58 2022 +0100  
gray  
commit f10f9725e3319af840a3d891ca8950436a219eb0  
Merge: 6ad5c15 9b0a614  
Author: annaskoulikari <gitlearningjourney@gmail.com>  
Date:   Sun Feb 20 09:11:06 2022 +0100  
merge commit 2
```

What to notice:

- You have made the gray commit and pushed it to the remote repository.

This is illustrated in [Visualize it 11-3](#).

[VISUALIZE IT 11-3]



`rainbow` repository and push it to the remote repository

Next, your friend will add some work in the `friend-rainbow` repository on the same branch, which will lead to divergent histories between your `main` branch and your friend's `main` branch. This also provides a good opportunity to introduce some helpful features of the staging area.

Unstaging and Staging Files

In this section, your friend is going to make changes to both of the files in the `friend-rainbow` repository and add them both to the staging area. But then they will realize they want to make two separate commits for the different pieces of work. Therefore, they will have to *unstage* a file.

Let's take a look at [Example Book Project 11-2](#), in which we revisit the scenario that we explored in [Example Book Project 3-2](#) to see why we might need to unstage a file.

Example Book Project 11-2

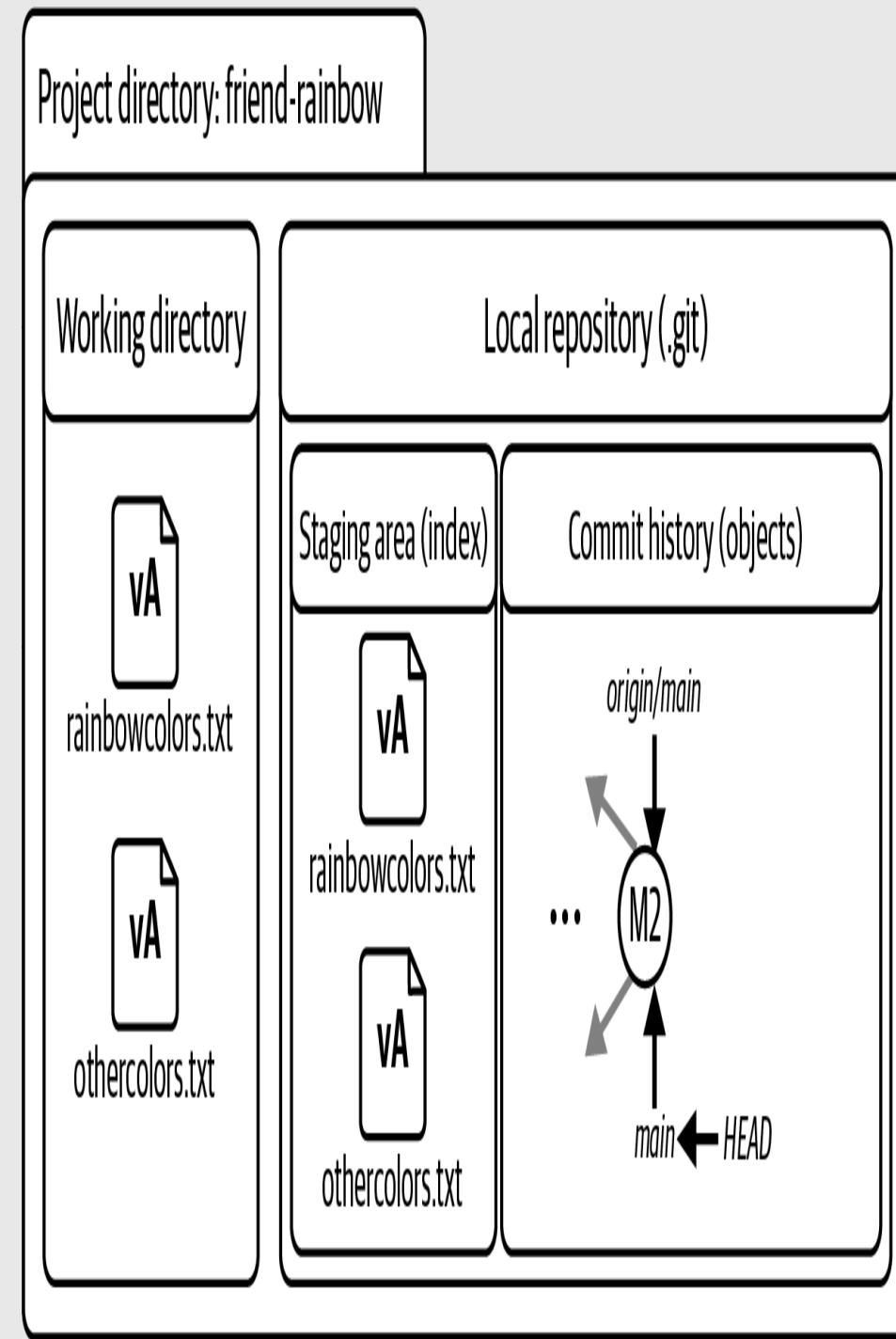
Suppose I work on chapters 1, 2, and 3, of my book, editing the files corresponding to those chapters: `chapter_one.txt`, `chapter_two.txt`, and `chapter_three.txt`. I add all the chapter files to the staging area, which means all the changes I have made to chapters 1, 2, and 3 will be saved in my next commit. My plan is for my editor to review all of my changes.

Before I make the commit, however, I realize that only the changes in chapter 2 are actually ready to be reviewed. In other words, I want only the changes that I made in `chapter_two.txt` to be included in my next commit. Since the staging area is a rough draft space where I can add and remove modified files in order to craft what will be included in the next commit, I'm free to remove the modified `chapter_two.txt` and `chapter_three.txt` files from this area, so that only the changes in the modified `chapter_two.txt` file are included in my next commit. This means my editor will review only those changes.

Now that you have an idea of why it is important to know how to unstage a file from the staging area, you're ready to practice unstaging a file in the Rainbow project.

To explore what happens in the upcoming example, we will focus on the `friend-rainbow` repository in the Visualize It diagrams, using the Git Diagram introduced in [Chapter 2](#). The current state of the `friend-rainbow` repository is shown in [Visualize it 11-4](#).

[VISUALIZE IT 11-4]



The `friend-rainbow` project directory before your friend edits any files

What to notice:

- The current versions of the `rainbowcolors.txt` file and `othercolors.txt` file that are in the working directory and staging area are represented as version A (vA).

Next, in [Follow Along 11-2](#), your friend is going to make some changes to the files.

[FOLLOW ALONG 11-2]

- 1 In the `friend-rainbow` project directory in your text editor:
 - In the `othercolors.txt` file, add "Black is not a color in the rainbow." on line 2 and save the file.
 - In the `rainbowcolors.txt` file, add "These are the colors of the rainbow." on line 8 and save the file.
- 2

```
friend-rainbow $ git status
on branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:

  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   othercolors.txt
    modified:   rainbowcolors.txt

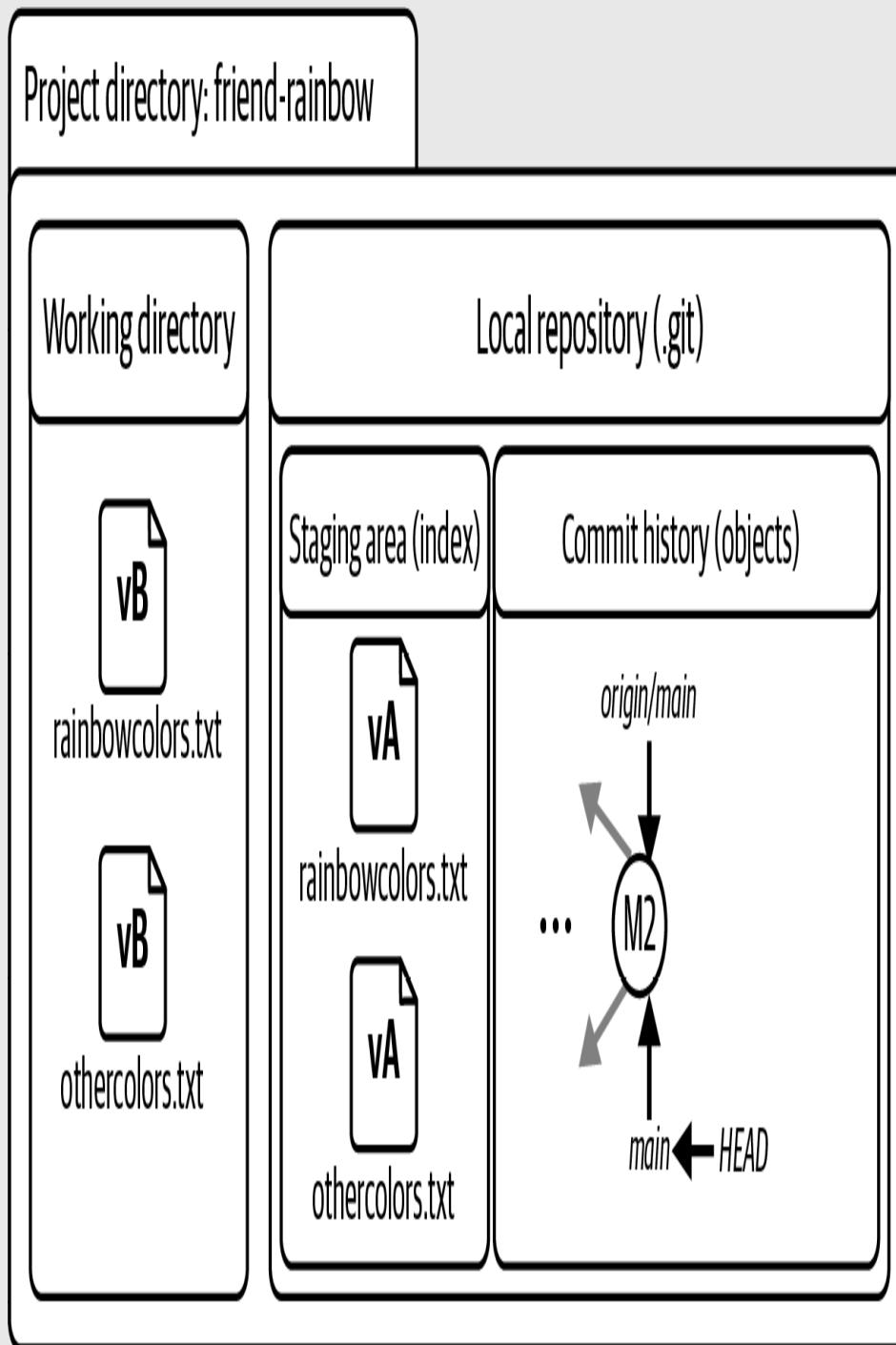
no changes added to commit (use "git add" and/or "git commit -a")
```

What to notice:

- Your friend edited both the `rainbowcolors.txt` file and the `othercolors.txt` file, and they are listed as modified files that have not been added to the staging area.

This is illustrated in [Visualize it 11-5](#).

[VISUALIZE IT 11-5]



The `friend-rainbow` project directory after your friend edits the `rainbowcolors.txt` and `othercolors.txt` files in the working directory

What to notice:

- The modified versions of the `rainbowcolors.txt` file and the `othercolors.txt` file that your friend edited are represented as version B (vB).
- The versions of the `othercolors.txt` and `rainbowcolors.txt` files in the staging area (vA) are different from the versions in the working directory (vB).

Now, in [Follow Along 11-3](#), your friend will add the updated files to the staging area to include them in the next commit.

[FOLLOW ALONG 11-3]

1 friend-rainbow \$ **git add rainbowcolors.txt othercolors.txt**

2 friend-rainbow \$ **git status**

on branch main

Your branch is up to date with 'origin/main'.

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

modified: othercolors.txt

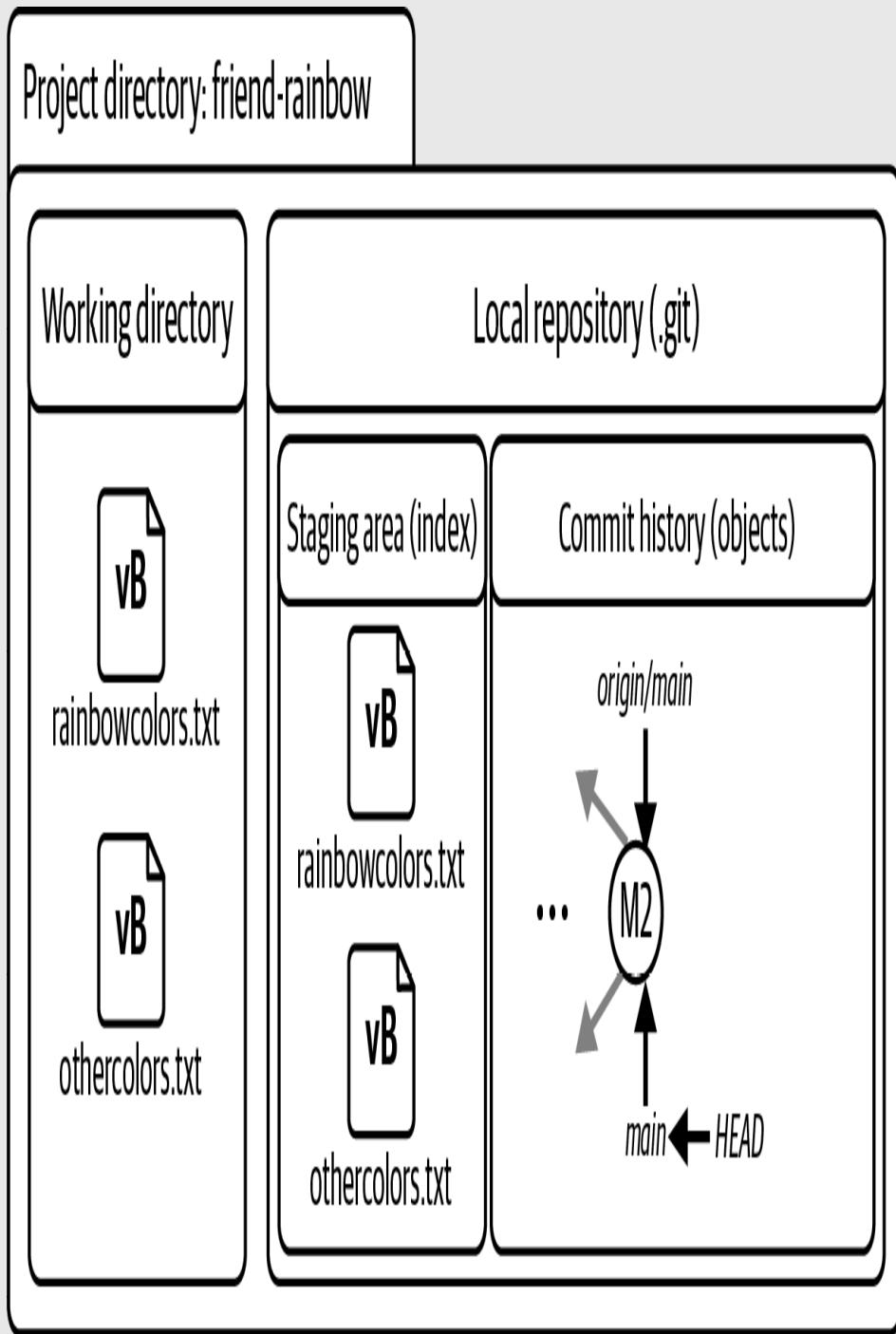
modified: rainbowcolors.txt

What to notice:

- Your friend added both the modified `rainbowcolors.txt` file and the modified `othercolors.txt` file to the staging area.

This is illustrated in [Visualize it 11-6](#).

[VISUALIZE IT 11-6]



The `friend-rainbow` project directory after your friend adds the updated versions of the `rainbowcolors.txt` and `othercolors.txt` files to the staging area

What to notice:

- The versions of the `othercolors.txt` and `rainbowcolors.txt` files in the staging area have changed from vA to vB.

Now, let's assume that your friend decides they want to make separate commits for the work they did adding the color black to the list of non-rainbow colors and the work they did commenting on the rainbow.

In [“Introducing the Staging Area” on page 27](#), I mentioned that the staging area is like a rough draft space where you can add and remove modified files to craft what will be included in your next commit. Up until now, you have been using the `git add` command to add files to the staging area. Now you're going to see how to remove a modified file from the staging area, or in other words, change the version of the file in the staging area. In the `git status` command output in [Follow Along 11-3](#), Git provides the instructions for how to unstage a file: (`use "git restore --staged <file>..." to unstage`). As this output indicates, to remove files from the staging area you can use the `git restore` command with the `--staged` option, passing in the names of any files you want to unstage, separated by spaces.

[SAVE THE COMMAND]

`git restore --staged <filename>`

Restore a file to another version of the file in the staging area

[NOTE]

If you have a version of Git that is older than version 2.23, then you won't have access to the `git restore` command. Your output may suggest you use the `git reset` command, which is another command that can unstage a file. In the upcoming Follow Along, in step 1, you will have to enter `git reset HEAD rainbowcolors.txt` instead of `git restore --staged rainbowcolors.txt` to unstage the `rainbowcolors.txt` file. Keep in mind that in Git there are often many different ways to achieve the same outcome.

In [Follow Along 11-4](#), your friend will use the `git restore` command with the `--staged` option, as directed in the `git status` output in [Follow Along 11-3](#), passing in the name of the `rainbowcolors.txt` file to unstage it. This means their next commit will include only the changes they made in the `othercolors.txt` file.

[FOLLOW ALONG 11-4]

1 friend-rainbow \$ **git restore --staged rainbowcolors.txt**

2 friend-rainbow \$ **git status**

On branch main

Your branch is up to date with 'origin/main'.

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

modified: othercolors.txt

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

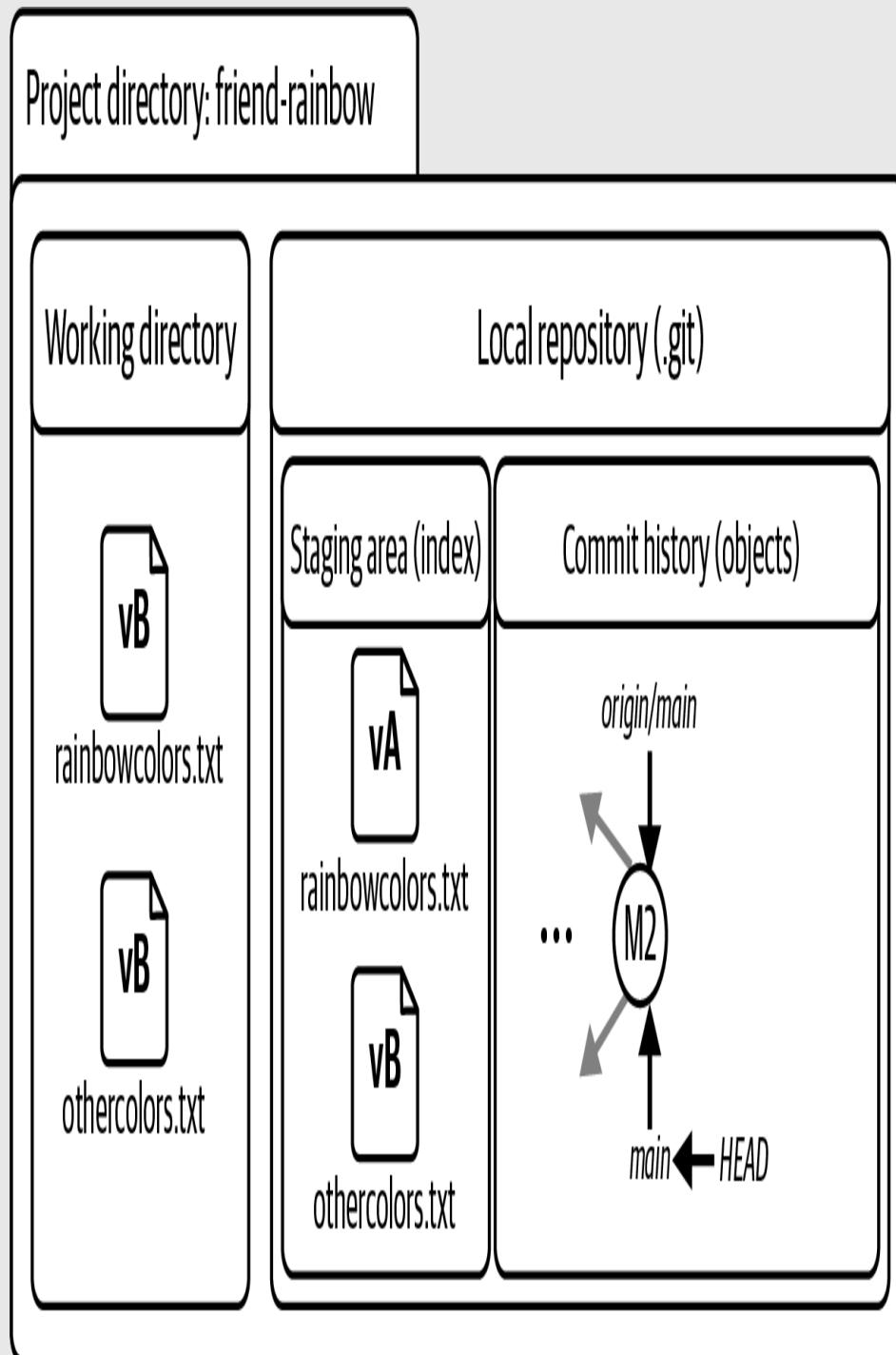
modified: rainbowcolors.txt

What to notice:

- The `git status` output shows that the updated version of the `othercolors.txt` file is still in the staging area, whereas the updated version of the `rainbowcolors.txt` file has been unstaged and is no longer in the staging area.

We illustrate this in [Visualize it 11-7](#).

[VISUALIZE IT 11-7]



The `friend-rainbow` project directory after your friend unstages the `rainbowcolors.txt` file

What to notice:

- Your friend unstaged the `rainbowcolors.txt` file, so the version in the staging area went back to vA.
- The updated version of `othercolors.txt` (vB) is still in the staging area.
- The version of `rainbowcolors.txt` with your friend's most recent changes (vB) is still in the working directory.

Next, in [Follow Along 11-5](#), your friend will make a commit that will include only the change to the `othercolors.txt` file.

[FOLLOW ALONG 11-5]

1 friend-rainbow \$ **git commit -m "black"**
[main 29bdadd] black
1 file changed, 2 insertions(+), 1 deletion(-)

2 friend-rainbow \$ **git status**
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
(use "git push" to publish your local commits)
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working directory)
modified: rainbowcolors.txt
no changes added to commit (use "git add" and/or "git commit -a")

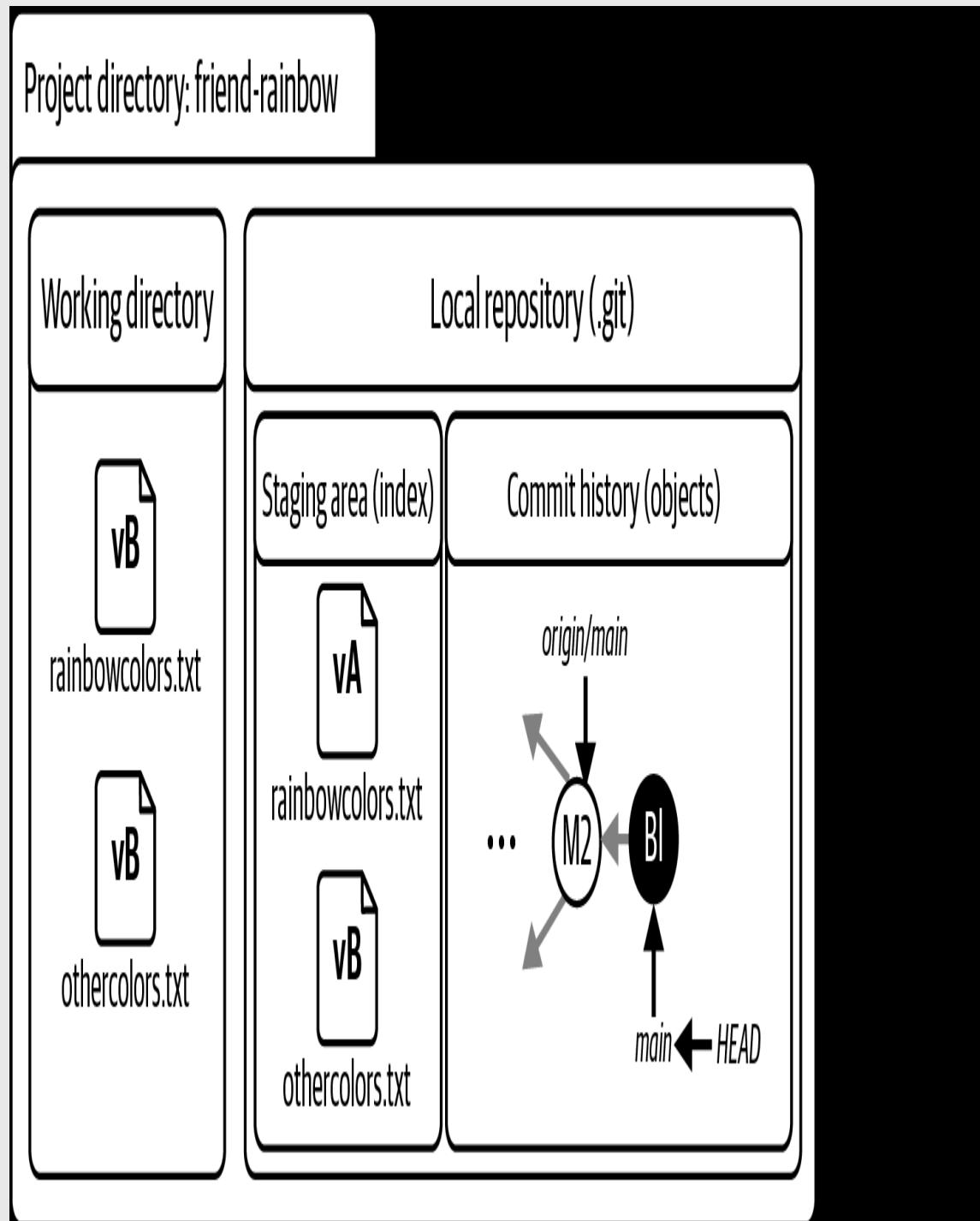
3 friend-rainbow \$ **git log**
commit 29bdadd50ddea41c75b476e776b6204a555b3d54 (**HEAD -> main**)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date: Sun Feb 20 10:07:38 2022 +0100
black
commit f10f9725e3319af840a3d891ca8950436a219eb0 (**origin/main**,
origin/HEAD)
Merge: 6ad5c15 9b0a614
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date: Sun Feb 20 09:11:06 2022 +0100
merge commit 2

What to notice:

- Your friend made the black commit.
- In step 2, the `git status` output mentions that the `rainbowcolors.txt` file is a modified file in the working directory.

[Visualize it 11-8](#) shows the state of the `friend-rainbow` project directory after [Follow Along 11-5](#).

[VISUALIZE IT 11-8]



The `friend-rainbow` project directory after your friend makes the black commit

Now, in [Follow Along 11-6](#), your friend will add the changes that they made to the

`rainbowcolors.txt` file to the staging area and make another commit.

[FOLLOW ALONG 11-6]

- 1** friend-rainbow \$ **git add rainbowcolors.txt**
- 2** friend-rainbow \$ **git commit -m "rainbow"**
[main 51dc6ec] rainbow
1 file changed, 1 insertion(+), 1 deletion(-)
- 3** friend-rainbow \$ **git log**
commit 51dc6ecb327578cca503abba4a56e8c18f3835e1 (**HEAD -> main**)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date: Sun Feb 20 10:10:11 2022 +0100

rainbow

commit 29bdadd50ddea41c75b476e776b6204a555b3d54
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date: Sun Feb 20 10:07:38 2022 +0100

black

commit f10f9725e3319af840a3d891ca8950436a219eb0 (**origin/main, origin/HEAD**)
Merge: 6ad5c15 9b0a614
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date: Sun Feb 20 09:11:06 2022 +0100

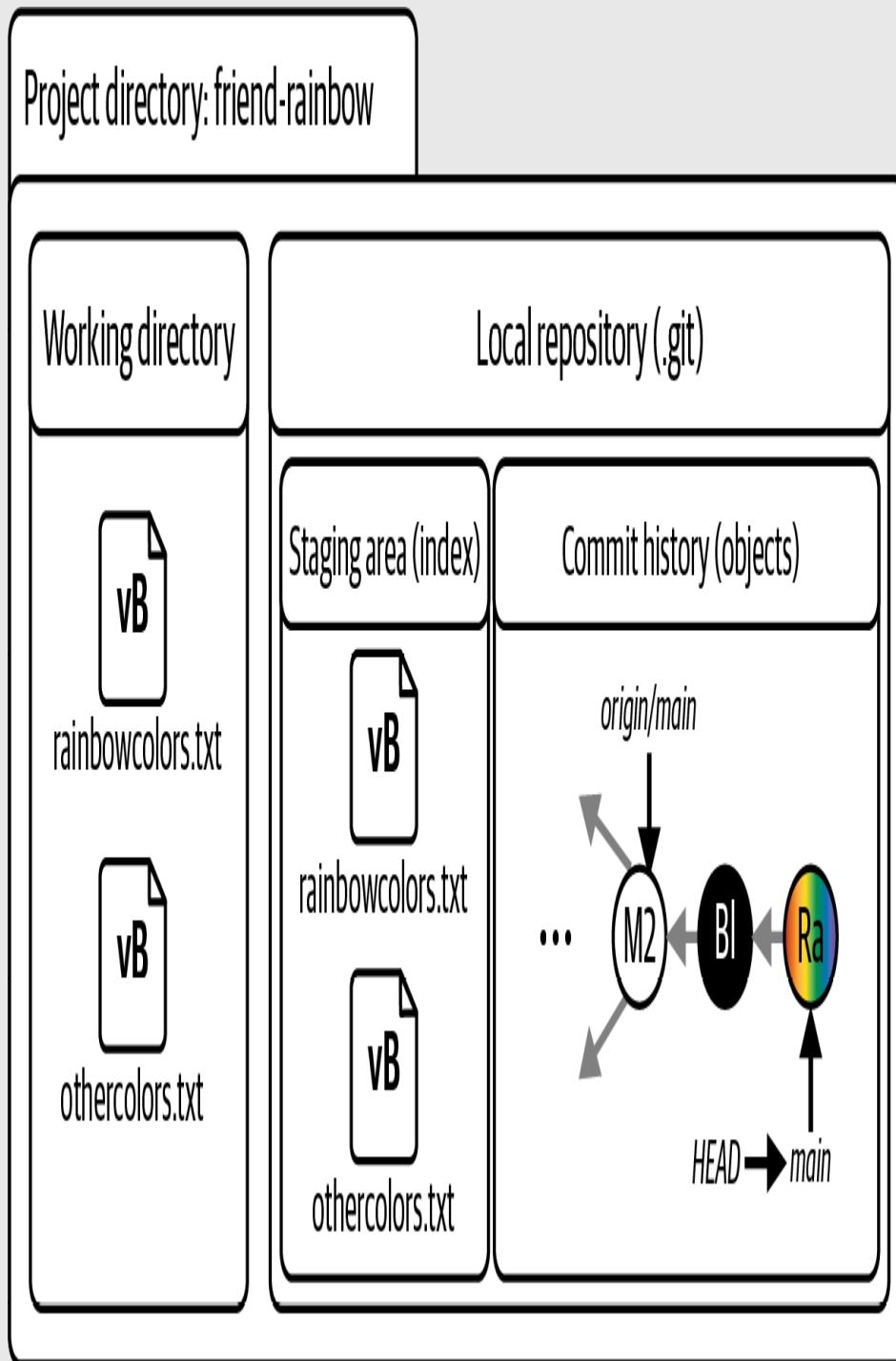
merge commit 2

What to notice:

- Your friend made the rainbow commit.

This is illustrated in [Visualize it 11-9](#).

[VISUALIZE IT 11-9]



The `friend-rainbow` project directory after your friend adds the `rainbowcolors.txt` file to the staging area and makes a commit

What to notice:

- The version of the `rainbowcolors.txt` file in the staging area is now vB, which is the version that is part of the rainbow commit.

You just saw how your friend was able to add files to and remove them from the staging area in order to craft exactly the commits they wanted. The local `main` branch in the `rainbow` repository and the local `main` branch in the `friend-rainbow` repository now have divergent development histories. Before your friend continues with the rebasing example, they need to make sure to fetch all the work you have pushed to the remote `main` branch from the remote repository.

Preparing to Rebase

As you saw in [Example Book Project 11-1](#) earlier in this chapter, to rebase a branch you first have to fetch all the work that has been done on the branch that you want to rebase onto. So, in preparation for rebasing their branch, in [Follow Along 11-7](#) your friend is going to fetch the latest updates from the remote repository.

[FOLLOW ALONG 11-7]

1 friend-rainbow \$ **git fetch**

```
remote: Enumerating objects: 5, done.  
remote: Counting objects: 100% (5/5), done.  
remote: Compressing objects: 100% (3/3), done.  
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0  
Unpacking objects: 100% (3/3), 307 bytes | 153.00 KiB/s, done.  
From github.com:gitlearningjourney/rainbow-remote  
f10f972..6f2cf36  main      -> origin/main
```

2 friend-rainbow \$ **git log --all**

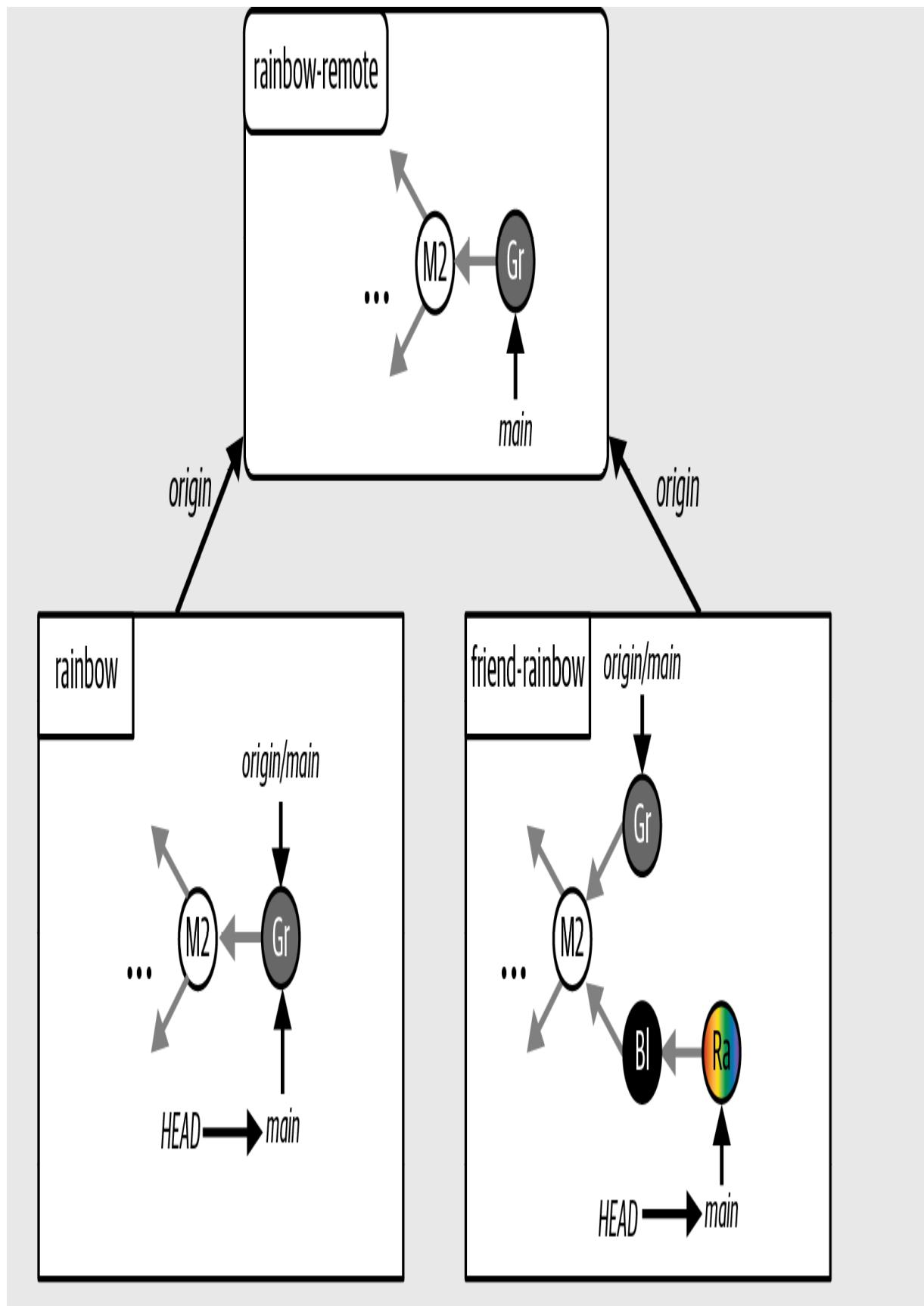
```
commit 51dc6ecb327578cca503abba4a56e8c18f3835e1 (HEAD -> main)  
Author: annaskoulikari <gitlearningjourney@gmail.com>  
Date:   Sun Feb 20 10:10:11 2022 +0100  
  
rainbow  
  
commit 29bdadd50ddea41c75b476e776b6204a555b3d54  
Author: annaskoulikari <gitlearningjourney@gmail.com>  
Date:   Sun Feb 20 10:07:38 2022 +0100  
  
black  
  
commit 6f2cf3698e6bf9078e8e0340ec9948f590405091 (origin/main,  
origin/HEAD)  
Author: annaskoulikari <gitlearningjourney@gmail.com>  
Date:   Sun Feb 20 09:27:58 2022 +0100  
  
gray
```

What to notice:

- Your friend fetched the gray commit from the remote repository, and the `origin/main` remote-tracking branch was updated to point to it.

This is illustrated in [Visualize it 11-10](#).

[VISUALIZE IT 11-10]



The Rainbow project after your friend updates the `friend-rainbow` repository by fetching your work from the remote repository

The `origin/main` remote-tracking branch in the `friend-rainbow` repository represents the latest version of the remote `main` branch. Your friend is now ready to rebase their local `main` branch onto the `origin/main` remote-tracking branch. Next, we'll go over the stages of the rebase process itself.

The Five Stages of the Rebase Process

This section will walk you through the five stages of the rebase process. To provide a visual illustration of the stages, I'll include Visualize It diagrams previewing the hands-on example that you will carry out later in this chapter, when your friend rebases their local `main` branch in the `friend-rainbow` repository onto the `origin/main` remote-tracking branch.

To initiate the rebase process, you use the `git rebase` command. Git will then carry out the five stages of the process itself; the only time you have to actively get involved is if there are any merge conflicts. We will discuss this situation in further detail in [“Rebasing and Merge Conflicts” on page 217](#).

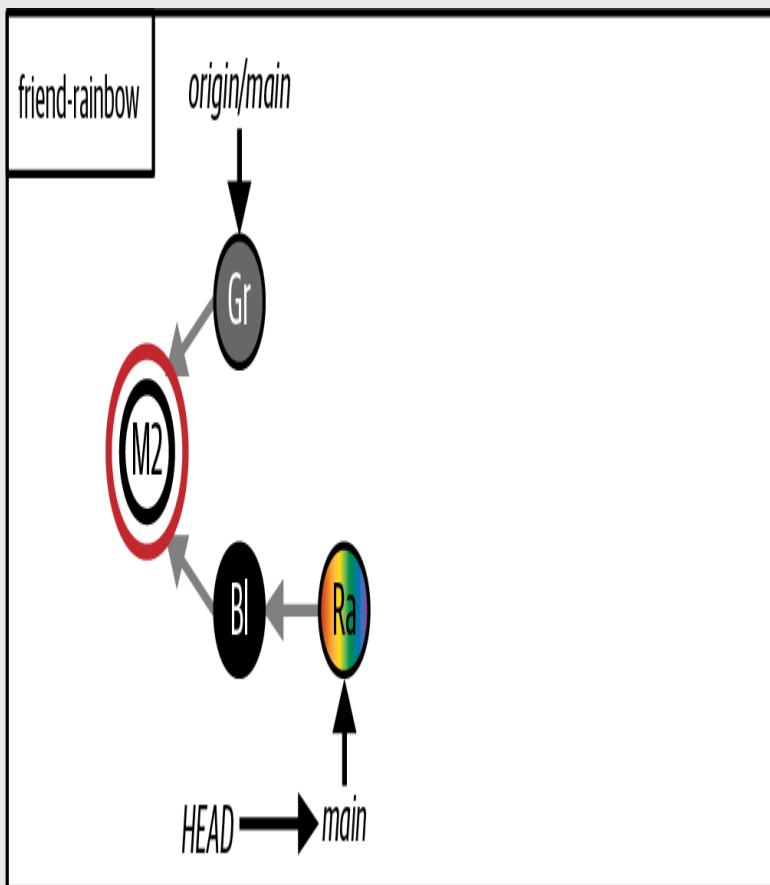
STAGE 1: FIND THE COMMON ANCESTOR

In the first stage of the process, Git will identify the common ancestor of the two branches involved in the rebase: the branch you're on and the branch you're rebasing onto.

In the Rainbow project example, the branch your friend will be on is the local `main` branch in the `friend-rainbow` repository and the branch they will be

rebasing onto will be the `origin/main` remote-tracking branch. The common ancestor will be the M2 merge commit, as shown in [Visualize it 11-11](#).

[VISUALIZE IT 11-11]



Stage 1: Locate the common ancestor of the branches involved in the rebase (here, the M2 merge commit)

STAGE 2: STORE INFORMATION ABOUT THE BRANCHES INVOLVED IN THE REBASE

In stage 2, Git will save the changes introduced by each commit of the branch you're on to a temporary area. It will also save additional

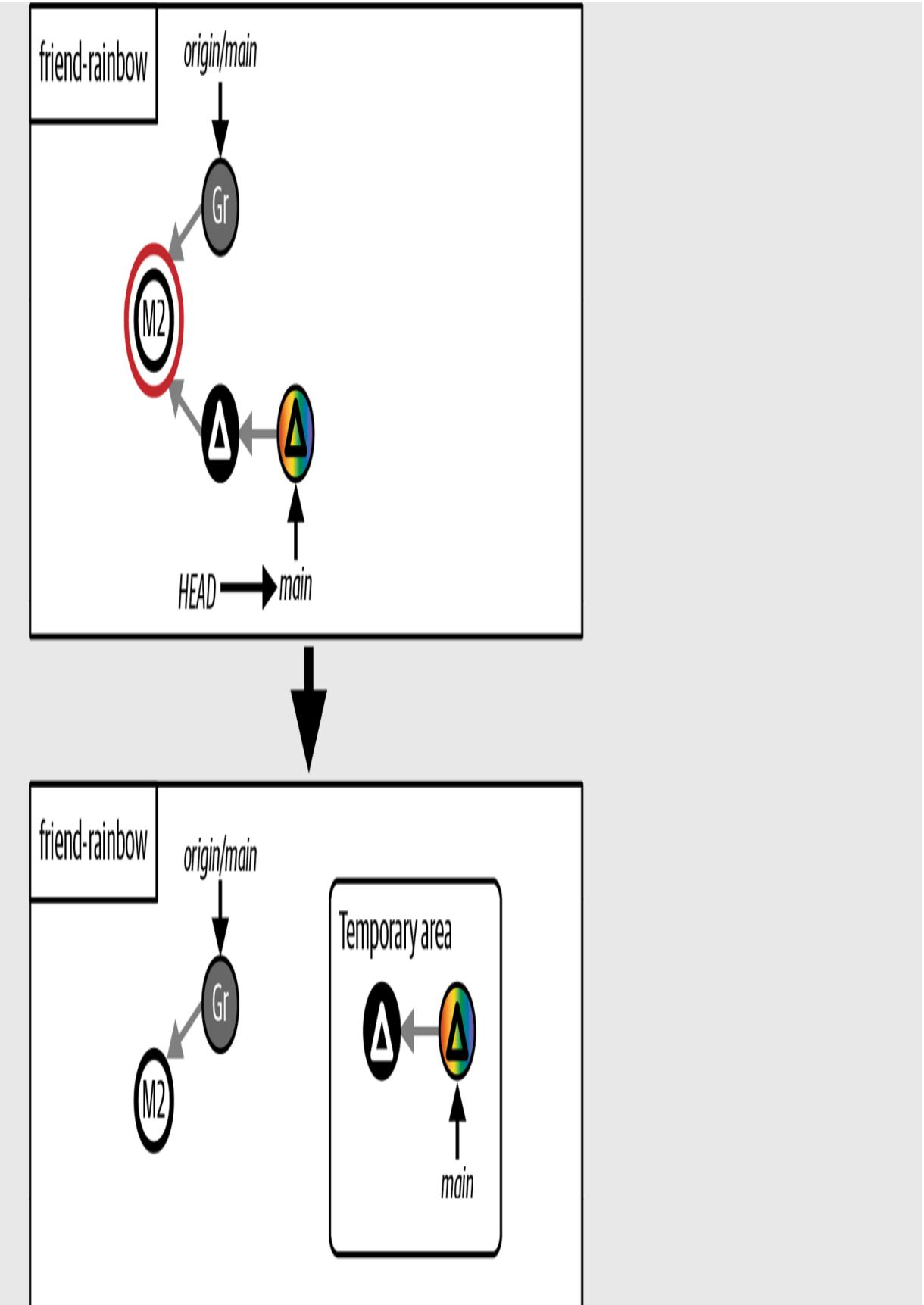
information in this temporary area, such as which branch you're rebasing onto and where it was pointing when you initiated the rebase.

In the Rainbow project example, the changes introduced by the black commit and the rainbow commit will be saved in the temporary area along with information about the remote `main` branch, as illustrated in [Visualize it 11-12](#).

[NOTE]

In the following Visualize It diagrams, triangles (Δ) are used to represent the changes introduced by commits.

[VISUALIZE IT 11-12]





HEAD →

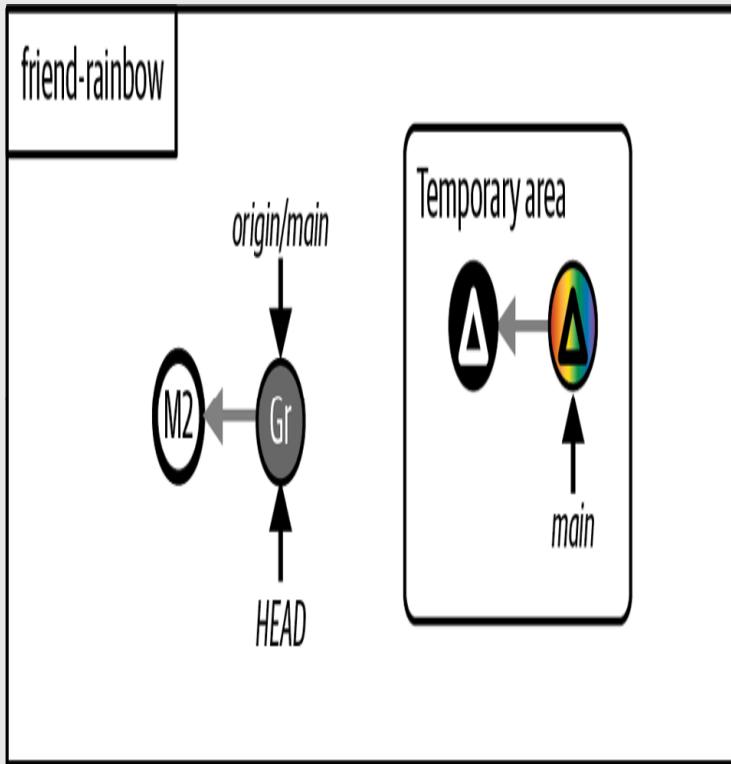
Stage 2: Save information about the rebase in a temporary area

STAGE 3: RESET HEAD

In stage 3, Git will reset `HEAD` to point to the same commit as the branch you are rebasing onto.

In the Rainbow project example, it will reset `HEAD` to the same commit that the `origin/main` remote-tracking branch is pointing to, which is the gray commit, as illustrated in [Visualize it 11-13](#).

[VISUALIZE IT 11-13]



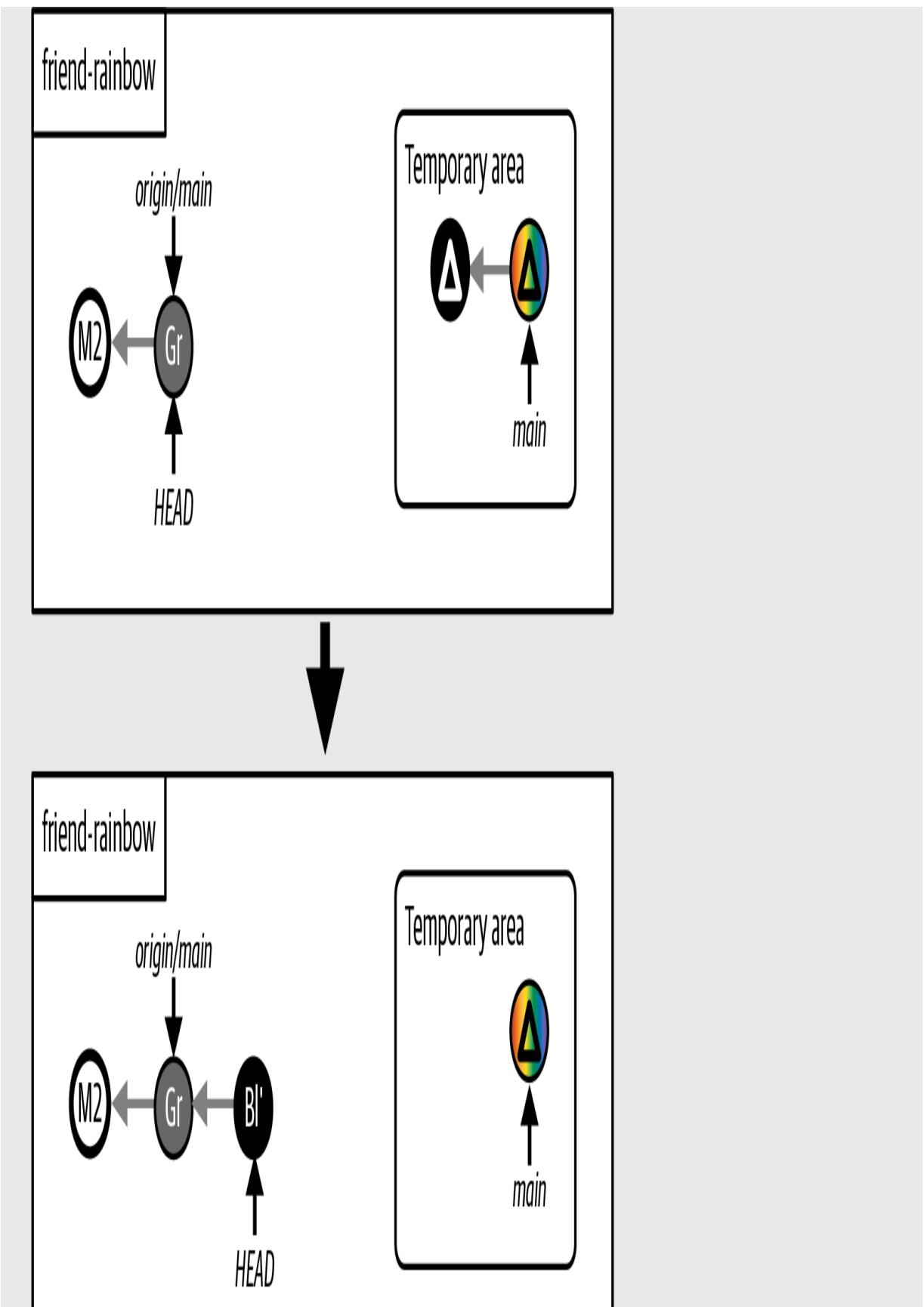
Stage 3: Reset `HEAD` to the same commit as the branch you are rebasing onto (here, the gray commit)

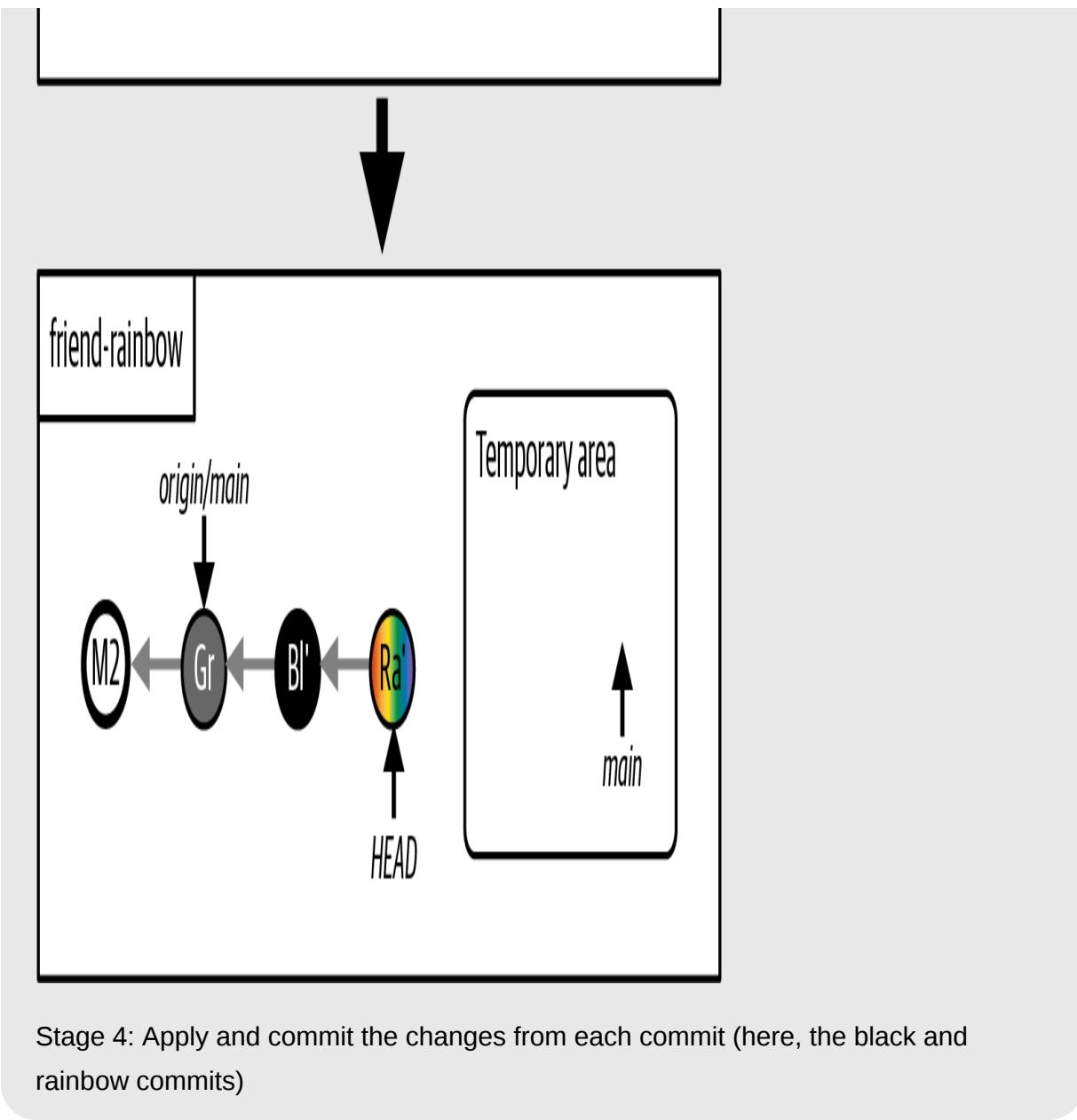
STAGE 4: APPLY AND COMMIT THE CHANGES

In stage 4, Git will apply the set of changes from each commit in turn, making a commit after it applies each set.

In the Rainbow project example, first it will apply the changes introduced by the black commit and create a new commit, and then it will apply the changes introduced by the rainbow commit and make a new commit, as shown in [Visualize it 11-14](#).

[VISUALIZE IT 11-14]





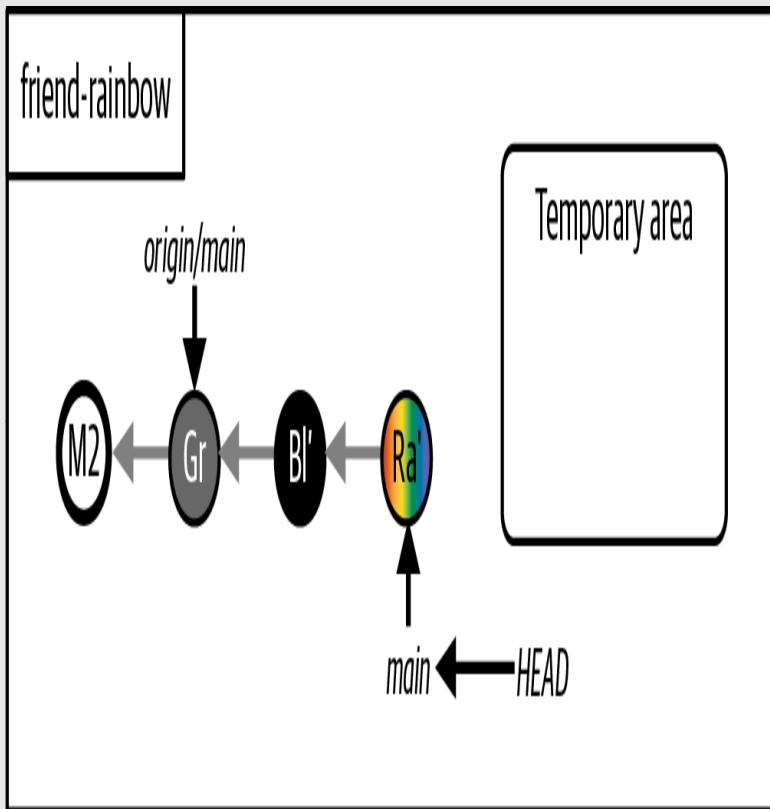
In [Visualize it 11-14](#), the new black commit and the new rainbow commit are represented as Bl' and Ra' (with an apostrophe), because they are entirely new commits.

STAGE 5: SWITCH ONTO THE REBASED BRANCH

In stage 5, Git will make the branch you rebased point to the last commit it reapplies, and it will check out that branch so that `HEAD` points to it.

In the Rainbow project example, the `main` branch will point to the new rainbow commit, as illustrated in [Visualize it 11-15](#).

[[VISUALIZE IT 11-15](#)]



Stage 5: Switch onto the branch you rebased

This concludes our walkthrough of the five stages of the rebase process. As mentioned at the beginning of this section, Git carries out the entire process itself; all you need to do is initiate it with the `git rebase` command. I also mentioned that the only time you'll need to get involved in the rebase

process is if Git encounters merge conflicts. We'll look briefly at that scenario in the next section.

Rebasing and Merge Conflicts

In [Chapter 10](#) you learned about merge conflicts, which arise when you integrate two branches where different changes have been made to the same parts in the same file(s), or if in one branch a file was deleted that was edited in the other branch.

Git will carry out the entire rebase process independently, unless it encounters merge conflicts. In this case, you must step in and resolve them. The process for resolving merge conflicts while rebasing is similar to the process when doing a three-way merge, with a few small differences.

When resolving merge conflicts in a three-way merge, all the merge conflicts are presented to you at the same time; once you've resolved all the conflicts and added all the updated files to the staging area, you make the final merge commit. By contrast, in the process of rebasing, as Git applies the changes from each commit one by one, it will pause the process if it encounters merge conflicts in any reapplied commit. This means that you may have to resolve merge conflicts several times when rebasing, depending on how many commits contain merge conflicts.

Once you're done resolving the merge conflicts in a specific commit, you need to add the updated files to the staging area and then instruct Git to resume the rebase process by entering the `git rebase` command with the `--continue` option. Git will then continue rebasing the rest of the commits. You don't have to explicitly make any commits, as you do in a three-way merge with conflicts.

As with a merge, if at any point during the process of resolving merge conflicts in a rebase you decide you don't want to continue the rebase process, you can choose to stop or abort the process by using the `git rebase` command with the `--abort` option. This will return all your files to the state they were in before the rebase.

[SAVE THE COMMAND]

`git rebase --continue`

Continue with the rebase process after having resolved merge conflicts

`git rebase --abort`

Stop the rebase process and go back to the state before the rebase

Now that we've covered what to expect when rebasing, it's time to practice with the Rainbow project.

Rebasing a Branch in Practice

Go to [Follow Along 11-8](#), where your friend will rebase the `main` branch in their local repository onto the `origin/main` remote-tracking branch.

[FOLLOW ALONG 11-8]

- 1 In the `friend-rainbow` project directory, make a note of the commit hashes for the black commit and the rainbow commit. You may use the `git log` command for this, which will include the commit hashes in its output. In the example in this book, the commit hashes are:

Black commit: `29bdadd50ddea41c75b476e776b6204a555b3d54`

Rainbow commit: `51dc6ecb327578cca503abba4a56e8c18f3835e1`

- 2 friend-rainbow \$ **git rebase origin/main**
Auto-merging othercolors.txt
CONFLICT (content): Merge conflict in othercolors.txt
error: could not apply 29bdadd... black
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git
rebase --abort".
Could not apply 29bdadd... black

[FOLLOW ALONG 11-8]

3 friend-rainbow \$ **git status**
interactive rebase in progress; onto 6f2cf36
Last command done (1 command done):
 pick 29bdadd black
Next command to do (1 remaining command):
 pick 51dc6ec rainbow
 (use "git rebase --edit-todo" to view and edit)
You are currently rebasing branch 'main' on 'bcb1dc0'.
 (fix conflicts and then run "git rebase --continue")
 (use "git rebase --skip" to skip this patch)
 (use "git rebase --abort" to check out the original branch)
Unmerged paths:
 (use "git restore --staged <file>..." to unstage)
 (use "git add <file>..." to mark resolution)
 both modified: othercolors.txt
no changes added to commit (use "git add" and/or "git commit -a")

What to notice:

- The rebase operation was interrupted because when Git was applying the changes that were included in the black commit, it encountered a merge conflict.
- In step 2, the `git rebase` command hints to you that you must `Resolve all conflicts manually, mark them as resolved with "git add/rm <conflicted_files>"`, then run `"git rebase --continue"`.

- In step 3, the `git status` command also shows you information about the rebase and which files you need to resolve merge conflicts in.

In [Follow Along 11-9](#), your friend is going to apply the steps you learned in [Chapter 10](#) to resolve the merge conflicts. Once they're done, they will execute the `git rebase --continue` command.

[FOLLOW ALONG 11-9]

1 Carry out step 1 of resolving merge conflicts, which is to choose what to keep, edit the content, and remove the conflict markers. You will keep all the changes from both branches. Keep the “Gray is not a color in the rainbow.” sentence above the “Black is not a color in the rainbow.” sentence. Make sure to save the `othercolors.txt` file when you’re done making your edits.

2 friend-rainbow \$ **git add othercolors.txt**

3 friend-rainbow \$ **git status**
interactive rebase in progress; onto 6f2cf36

Last command done (1 command done):

pick 29bdadd black

Next command to do (1 remaining command):

pick 51dc6ec rainbow

(use "git rebase --edit-todo" to view and edit)

You are currently rebasing branch 'main' on 'bcb1dc0'.

(all conflicts fixed: run "git rebase --continue")

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

modified: othercolors.txt

4 friend-rainbow \$ **git rebase --continue**

[FOLLOW ALONG 11-9]

5

Accept the default commit message in the Vim editor, and exit the editor. As you learned in [Chapter 9](#), to do this you must press the Escape key, type :wq, and press Enter. You will see the following output:

```
[detached HEAD e055f2b] black  
 1 file changed, 3 insertions(+), 1 deletion(-)  
 Successfully rebased and updated refs/heads/main.
```

6

```
friend-rainbow $ git log  
commit 7c09136bcbfdd9f638ed13c6653e06451579d21c (HEAD -> main)  
Author: annaskoulikari <gitlearningjourney@gmail.com>  
Date:   Sun Feb 20 10:10:11 2022 +0100  
  
        rainbow  
  
commit e055f2bc66aed1f3627041900a8c825c7a875206  
Author: annaskoulikari <gitlearningjourney@gmail.com>  
Date:   Sun Feb 20 10:07:38 2022 +0100  
  
        black  
  
commit 6f2cf3698e6bf9078e8e0340ec9948f590405091 (origin/main,  
origin/HEAD)  
Author: annaskoulikari <gitlearningjourney@gmail.com>  
Date:   Sun Feb 20 09:27:58 2022 +0100  
  
        gray
```

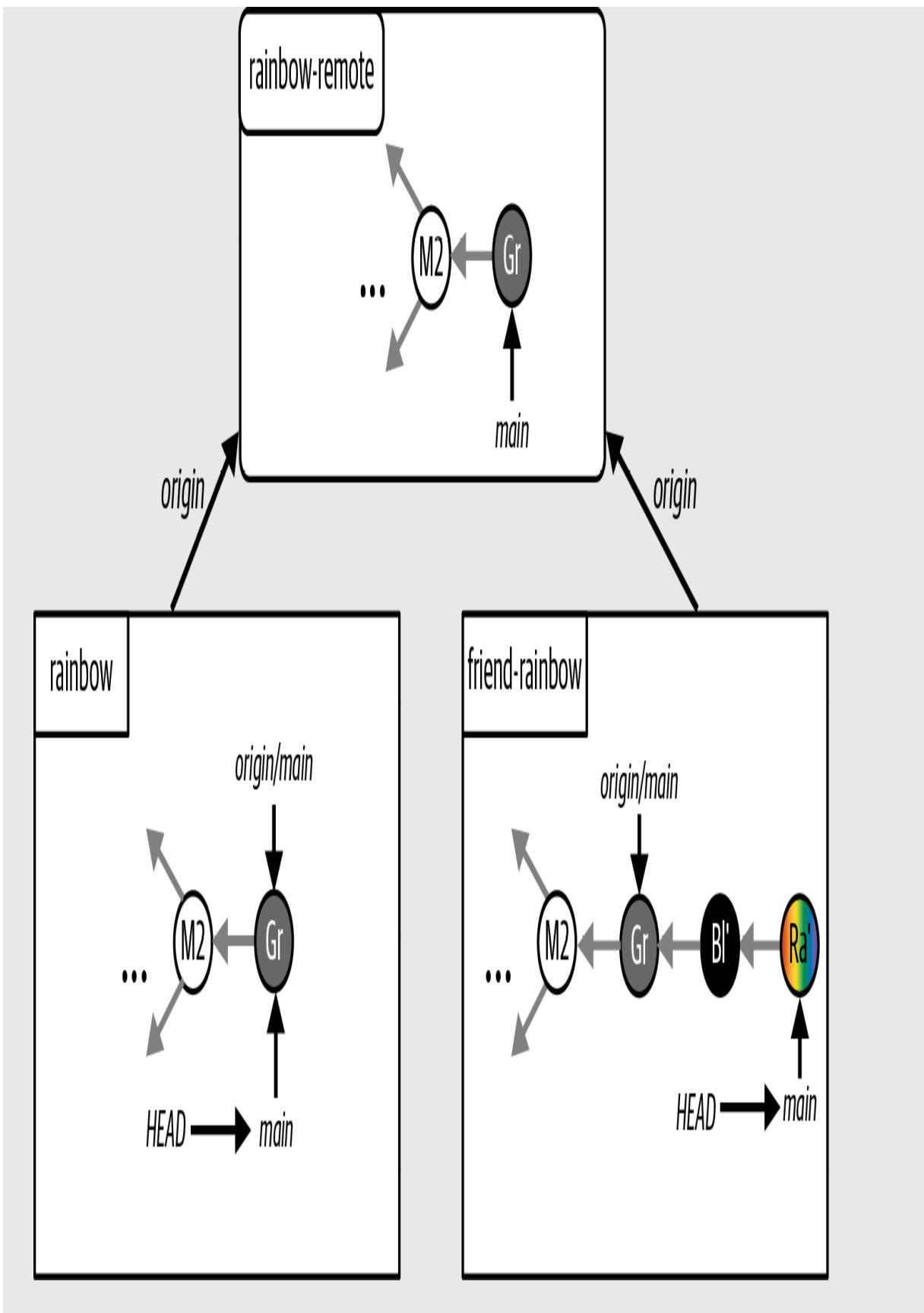
What to notice:

- In step 3, the `git status` output informs you all conflicts fixed: run "git rebase --continue".

- In step 6, the `git log` output indicates that the rebase process created a new rainbow commit and a new black commit. The commit hashes are new, as are the timestamps (the date and time of each commit).

The result of this process is illustrated in [Visualize it 11-16](#).

[VISUALIZE IT 11-16]



The Rainbow project after your friend rebases their local `main` branch onto the `origin/main` remote-tracking branch

What to notice:

- We represent the new black commit and the new rainbow commit as Bl' and Ra' (with an apostrophe).
- In the `friend-rainbow` repository, the gray commit, the new black commit, and the new rainbow commit form a linear project history. In other words, there is no merge commit.

[Figure 11-7](#) compares the commit hashes of the old black and rainbow commits and the new black and rainbow commits in this book. The commit hashes in your repositories will be different from the ones in this book because commit hashes are unique.



FIGURE 11-7

The commit hashes for the old black and rainbow commits are different from the commit hashes for the new black and rainbow commits

From this exercise, you have observed that rebasing rewrites history—and that brings us to the golden rule of rebasing.

The Golden Rule of Rebasing

As you saw in the Rainbow project example, rebasing creates entirely new commits. This means that rebasing changes the commit history. You must always be careful when you change the commit history because it can lead to complications in your project, especially when you’re working with other people.

The golden rule of rebasing states that *you should not rebase a branch that other people may have based work on*. For example, if you have pushed a branch to the remote repository, then it is considered a public branch. This means that other collaborators may also be working on this branch in their local repositories, or they may be pushing work to this branch in the remote repository.

In this case, you should refrain from rebasing the branch. To explore the importance of the golden rule of rebasing, let's take a look at [Example Book Project 11-3](#), in which we revisit the situation discussed in [Example Book Project 11-1](#).

Example Book Project 11-3

In [Example Book Project 11-1](#), I introduced a situation where there were two commits on the `main` branch (commits A and B) when I made the `chapter_five` branch and my coauthor made the `chapter_six` branch. Subsequently, my coauthor merged commit C from their `chapter_six` branch into their `main` branch and pushed the updated `main` branch to the remote expository. At the same time, I added commit D to the `chapter_five` branch in my local repository. This initial scenario is illustrated in [Figure 11-8](#).

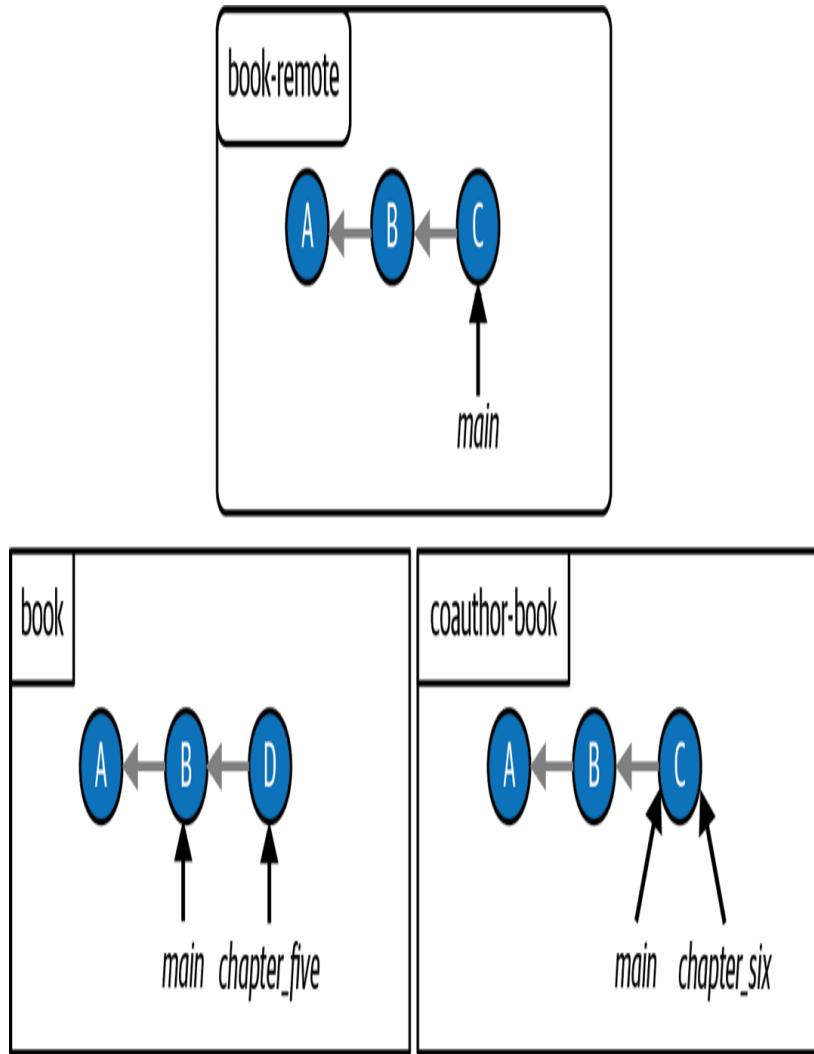


FIGURE 11-8

The Book project when the local `chapter_five` branch and the remote `main` branch have diverged

Now, suppose that I push the `chapter_five` branch to the remote repository, and then, in violation of the golden rule of rebasing, I also pull commit C from the remote repository and update my local `main` branch because I plan to rebase my `chapter_five` branch onto the latest version of the `main` branch. This situation is depicted in [Figure 11-9](#).

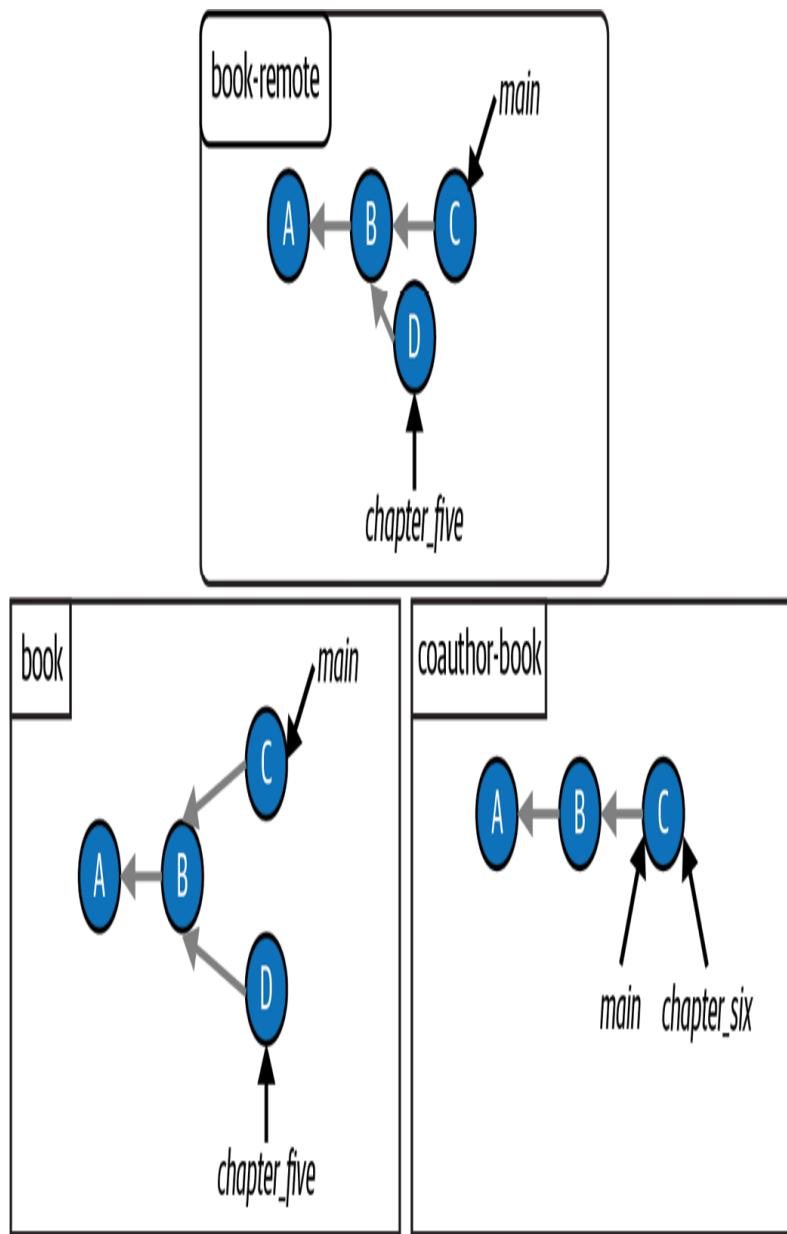


FIGURE 11-9

The Book project after I push the `chapter_five` branch to the remote repository and pull the commits on the `main` branch

I don't realize my mistake, and I go ahead with my plan to rebase the `chapter_five` branch onto the `main` branch. [Figure 11-10](#) shows the state of

the remote and local repositories after the rebase. D' in the `book` repository represents the re-created commit D after the rebase.

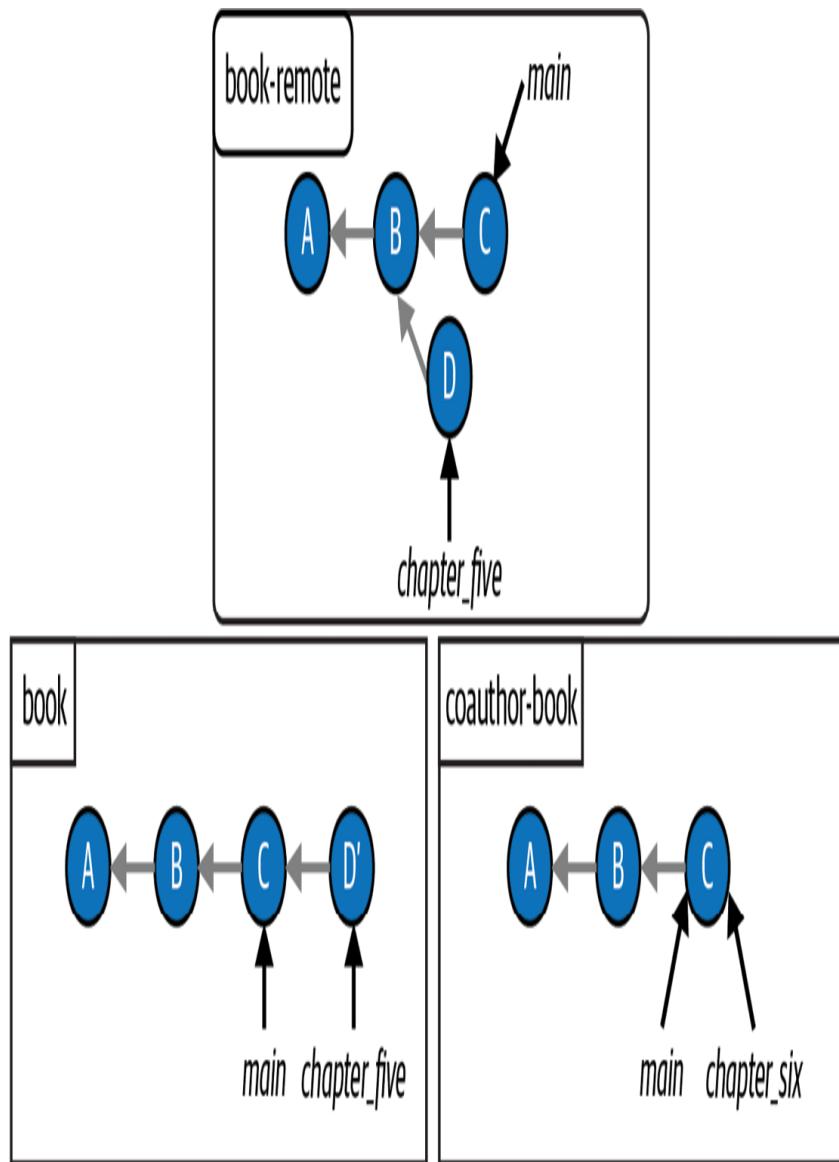


FIGURE 11-10

The Book project after I rebase the `chapter_five` branch onto the `main` branch

I have not yet pushed my updated `chapter_five` branch to the remote repository. Now let's assume my coauthor sees the remote `chapter_five` branch in the remote repository and decides to contribute to it. They pull the `chapter_five` branch into their `coauthor-book` repository, add commits E and F to it, and then push them up to the remote repository. This situation is illustrated in [Figure 11-11](#).

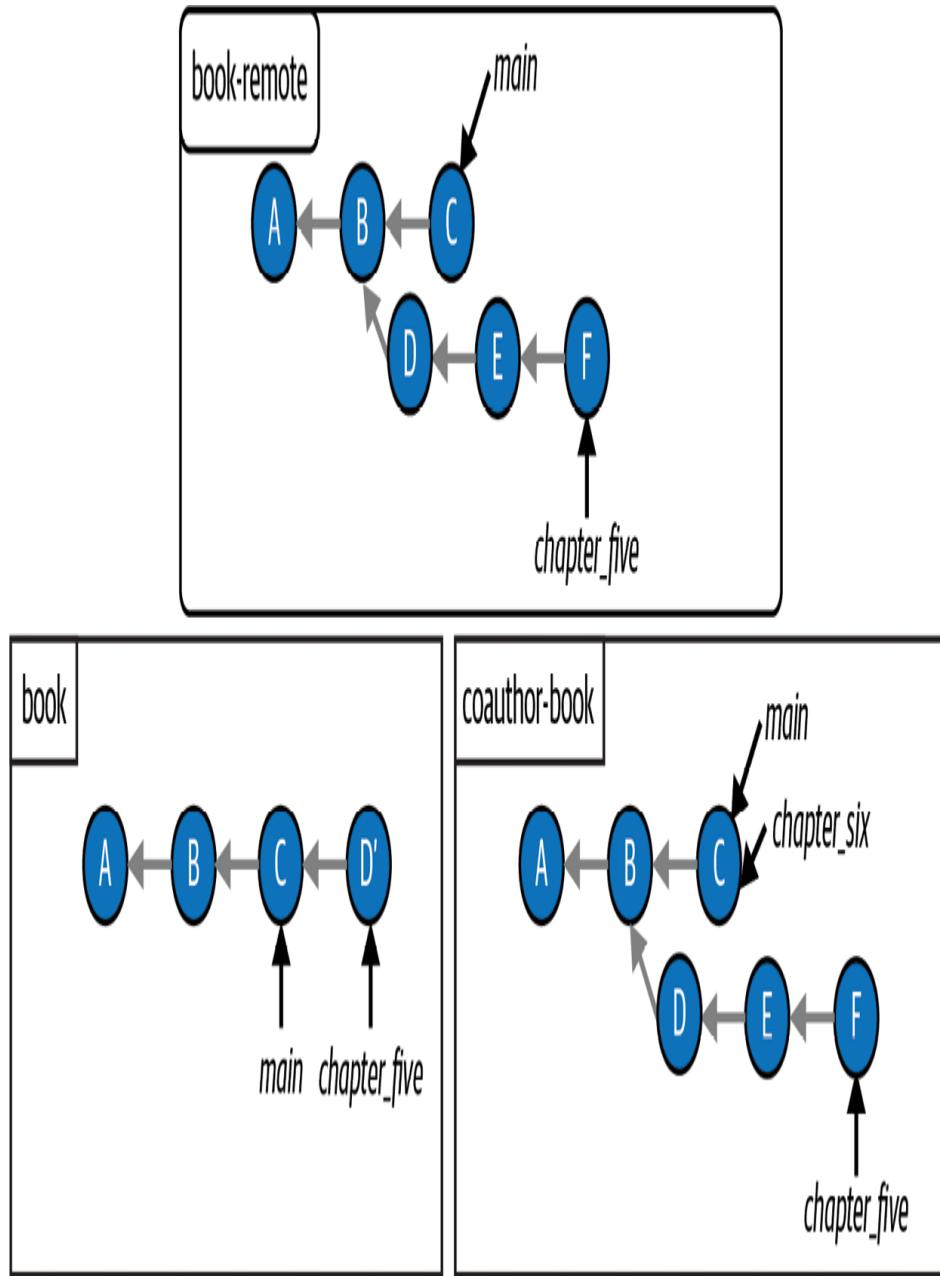


FIGURE 11-11

The Book project after my coauthor contributes to the remote `chapter_five` branch

When I try to push my `chapter_five` branch to the remote repository I will receive the following error:

```
error: failed to push some refs to 'github.com:gitlearningjourney/rainbow-  
remote.git'  
  
hint: Updates were rejected because the tip of your current branch is behind its  
remote counterpart. Integrate the remote changes (e.g. 'git pull ...') before  
pushing again. See the 'Note about fast-forwards' in 'git push --help' for  
details.
```

This error informs me that I am not able to push my changes to the remote repository. As you can see in [Figure 11-11](#), the commit histories of the remote `chapter_five` branch and the local `chapter_five` branch no longer contain the same commits: the remote `chapter_five` branch is made up of commits A, B, D, E, and F, while the local `chapter_five` branch is made up of commits A, B, C, and D'.

This is a sticky situation, because now my coauthor and I have no easy way of resolving this issue. We will have to communicate to troubleshoot and see how we can resolve the disparities between our branches.

[Example Book Project 11-3](#) illustrates why you should follow the golden rule of rebasing when using the rebase operation. To conclude, you may safely rebase a branch if:

- You have a local branch that has never been pushed to the remote repository.
- You have a local branch that you've pushed to a remote repository that you're 100% sure nobody has based work on or contributed to.

That's it. If there is a possibility that someone else has worked on the branch, then it is recommended that you avoid rebasing.

Syncing the Repositories

To make sure the repositories in the Rainbow project are in sync, your friend will have to push their changes to the remote repository and you will have to pull the changes down into the `rainbow` repository. Go to [Follow Along 11-10](#) to do this now.

[FOLLOW ALONG 11-10]

1 friend-rainbow \$ **git push**
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 4 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 652 bytes | 652.00 KiB/s, done.
Total 6 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:gitlearningjourney/rainbow-remote.git
 6f2cf36..7c09136 main -> main

2 friend-rainbow \$ **git log**
commit 7c09136bcbfdd9f638ed13c6653e06451579d21c (**HEAD -> main**,
origin/main, **origin/HEAD**)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date: Sun Feb 20 10:10:11 2022 +0100
 rainbow
commit e055f2bc66aed1f3627041900a8c825c7a875206
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date: Sun Feb 20 10:07:38 2022 +0100
 black

3 Go to the command line window where you're in the **rainbow** repository to execute the command in step 4.

[FOLLOW ALONG 11-10]

4 rainbow \$ **git pull**

```
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 1), reused 6 (delta 1), pack-reused 0
Unpacking objects: 100% (6/6), 632 bytes | 105.00 KiB/s, done.
From github.com:gitlearningjourney/rainbow-remote
   6f2cf36..7c09136  main      -> origin/main
Updating 6f2cf36..7c09136
Fast-forward
  othercolors.txt    | 4 +++-
  rainbowcolors.txt | 2 +-
  2 files changed, 4 insertions(+), 2 deletions(-)
```

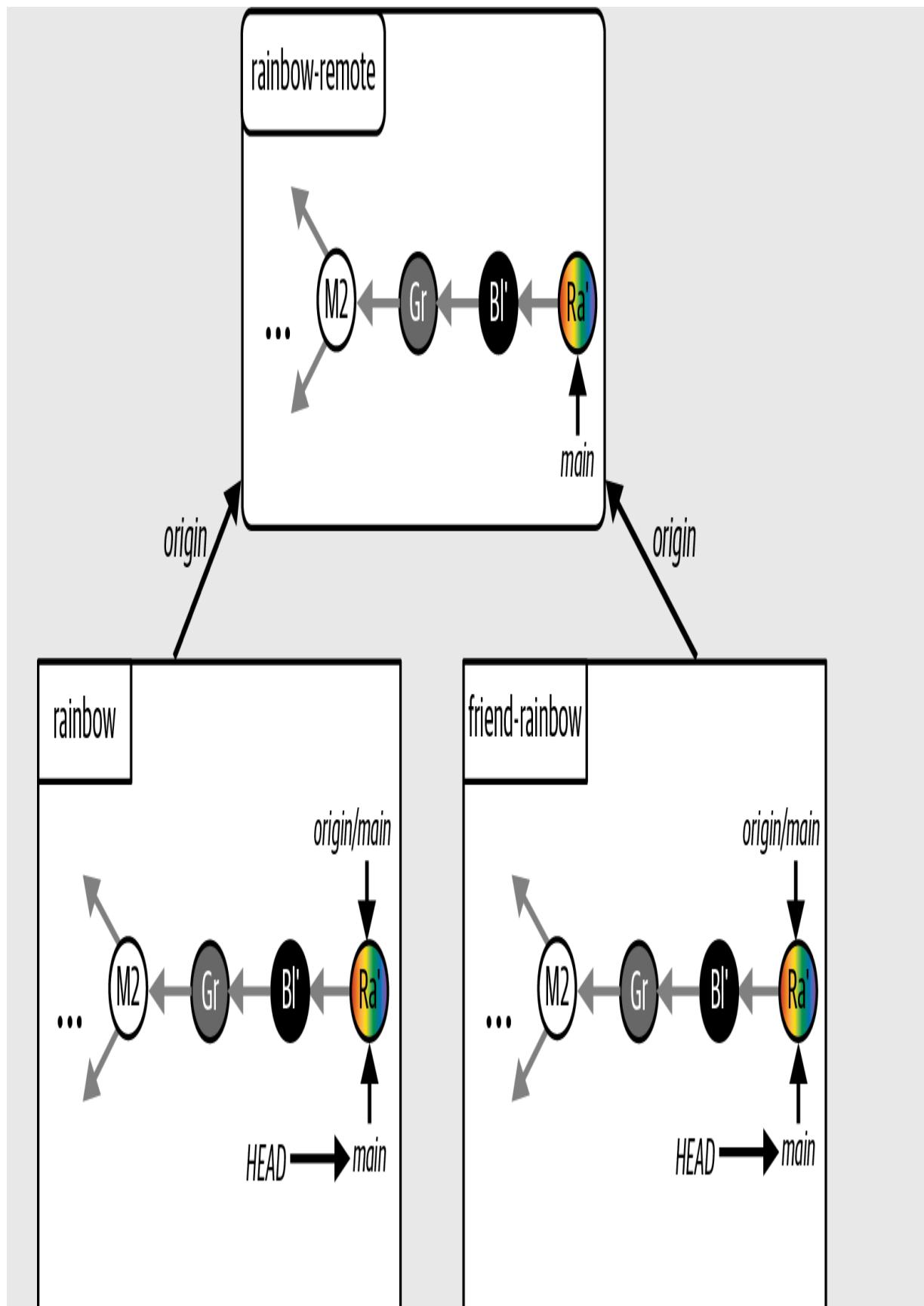
5 rainbow \$ **git log**

```
commit 7c09136bcbfdd9f638ed13c6653e06451579d21c (HEAD -> main,
origin/main)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sun Feb 20 10:10:11 2022 +0100
rainbow

commit e055f2bc66aed1f3627041900a8c825c7a875206
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sun Feb 20 10:07:38 2022 +0100
black
```

All three repositories in the Rainbow project are now in sync, as illustrated in [Visualize it 11-17](#).

[VISUALIZE IT 11-17]

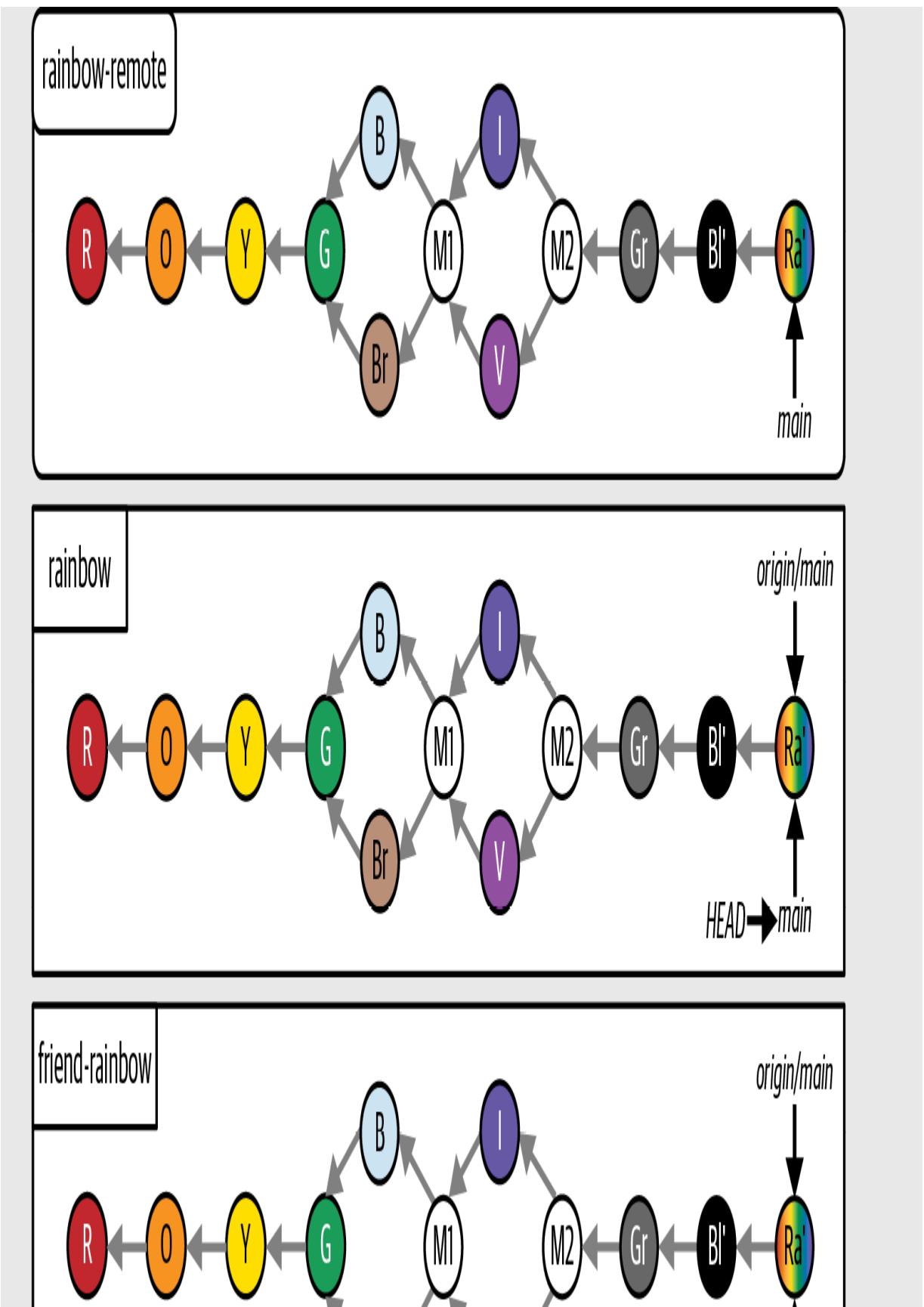


The Rainbow project after your friend pushes their local `main` branch to the remote repository and you pull from the remote repository to update your local `main` branch

State of the Local and Remote Repositories

[Visualize it 11-18](#) shows the state of the local and remote repositories in the Rainbow project with all the commits that were made from [Chapter 1](#) all the way to the end of this chapter.

[VISUALIZE IT 11-18]





The Rainbow project at the end of [Chapter 11](#) with all the commits since [Chapter 1](#)

Summary

In this chapter you learned about rebasing—the second way of integrating changes from one branch to another in Git—and how it can be used to avoid three-way merges and merge commits. You learned about the five stages of the rebase process that Git carries out, then you practiced rebasing with an example in the Rainbow project. During the process, you also practiced resolving merge conflicts.

Because rebasing rewrites the commit history, I introduced the golden rule of rebasing, which states that you should not rebase branches that other collaborators may have based work on. Additionally, you learned how to unstage files, or remove modified files from the staging area, so you can customize exactly what you include in a commit.

To this point, we have covered how you can integrate changes from one branch into another when working with a local repository through either merging or rebasing. Next, in [Chapter 12](#), we’re going to explore a helpful collaboration tool for Git projects that allows you to integrate changes remotely, called a pull request.

Pull Requests (Merge Requests)

Up until now, we have only covered how to integrate changes from one branch into another when working with a local repository.

In this chapter you are going to learn about pull requests, a helpful tool for collaboration that allows you to integrate changes from one branch into another in a remote repository. Along the way, you will also learn a handy trick for how to more easily define upstream branches for new local branches.

Additional resources to assist you as you work through this chapter are available in the Learning Git repository (<https://github.com/gitlearningjourney/learning-git>).

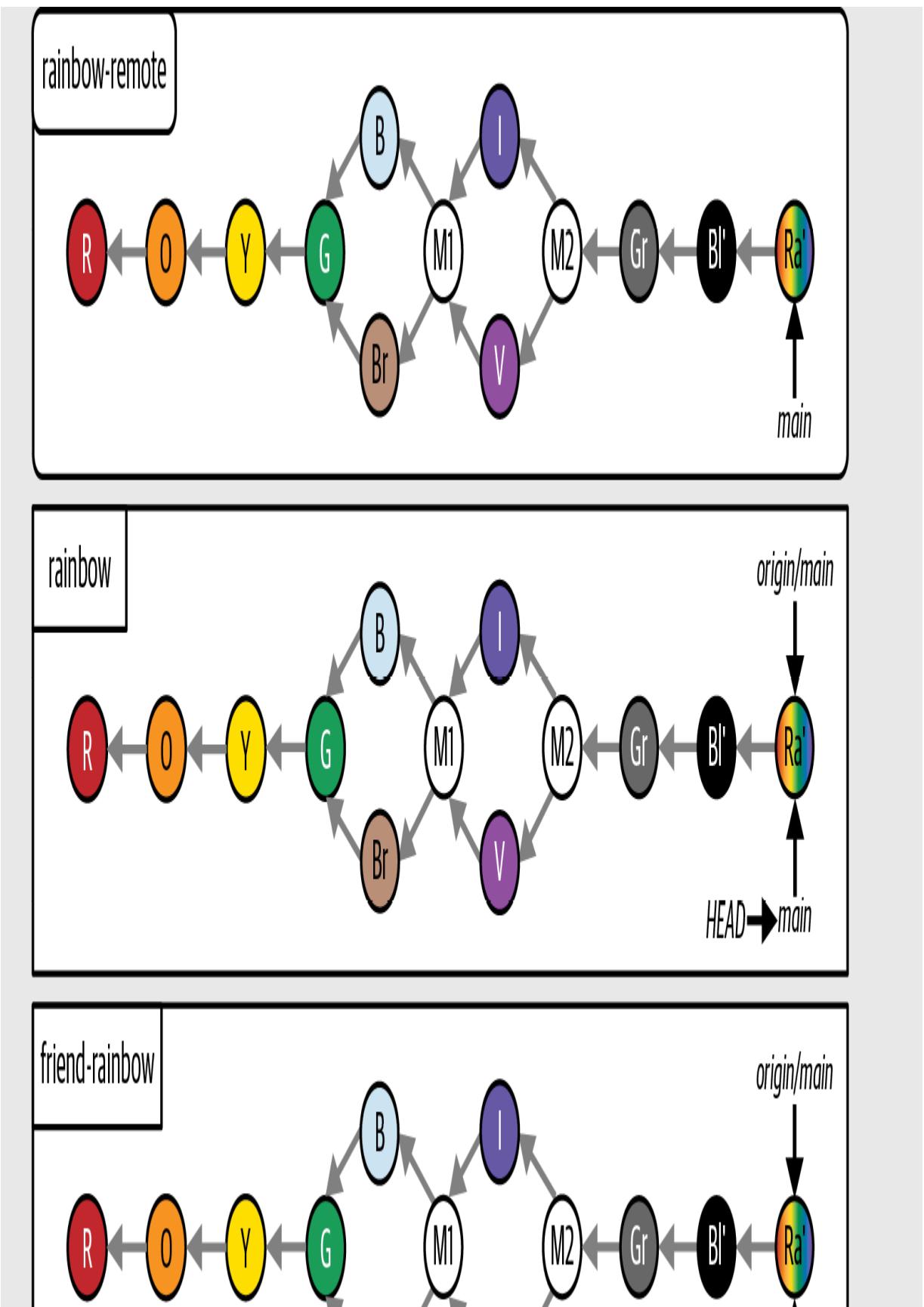
State of the Local and Remote Repositories

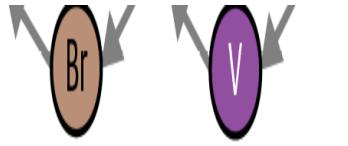
At the start of this chapter, you should have two local repositories called `rainbow` and `friend-rainbow` and one remote repository called `rainbow-remote`. All three of these repositories should be in sync, with the same commits and branches.

[Visualize it 12-1](#) shows the state of the local and remote repositories in the Rainbow project with all the commits that were made from [Chapter 1](#) to

[Chapter 11.](#)

[VISUALIZE IT 12-1]

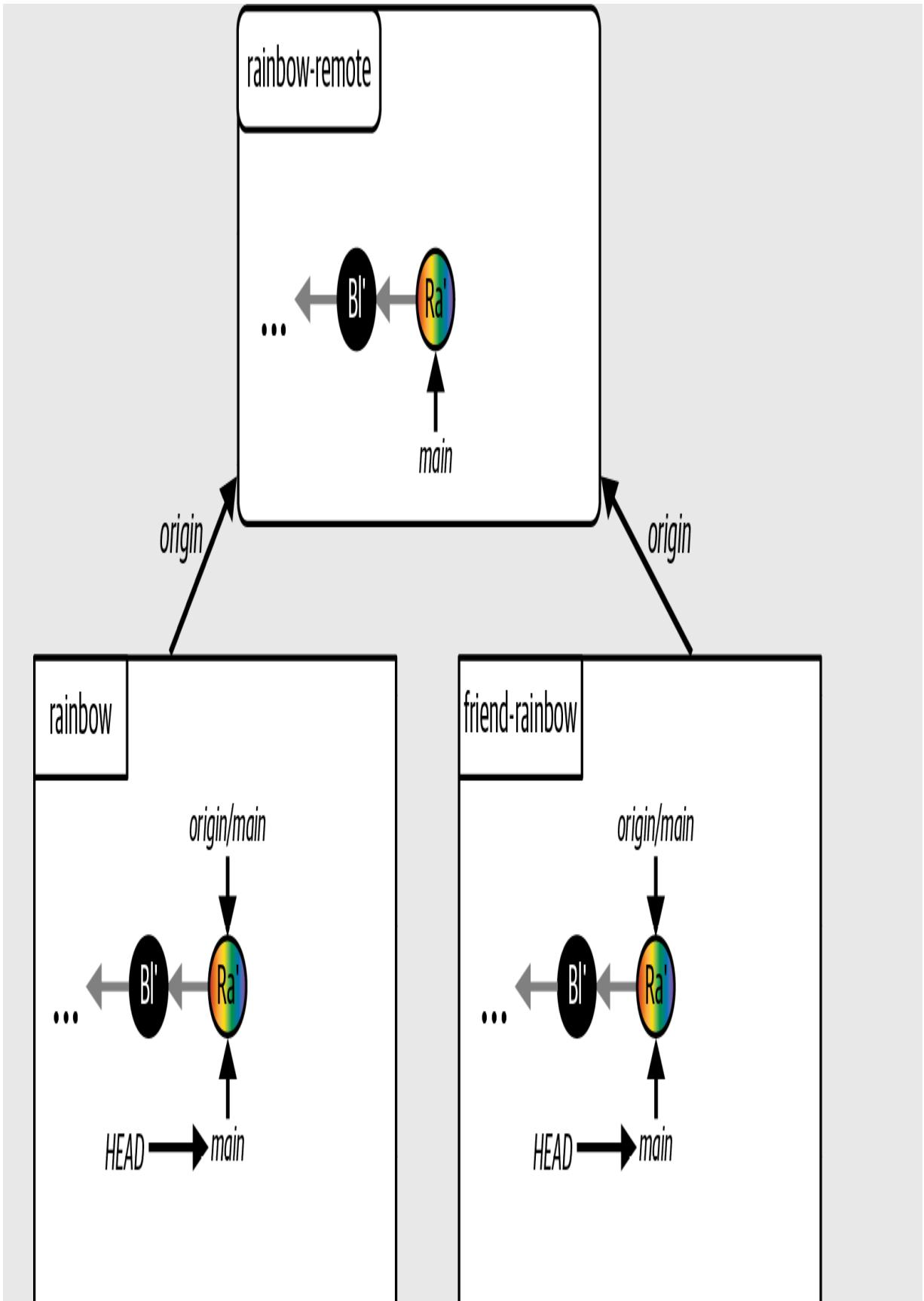




The Rainbow project at the start of [Chapter 12](#) with all the commits since [Chapter 1](#)

To focus on the commits you are going to make in this chapter, from here on I will simplify the Visualize It diagrams and show only the last two commits that are a part of the `main` branch in all the repositories, which are the black commit and the rainbow commit. This representation is shown in [Visualize it 12-2](#).

[VISUALIZE IT 12-2]



A simplified representation of the Rainbow project at the start of [Chapter 12](#), showing just the last two commits on the `main` branch in all the repositories

Introducing Pull Requests

A *pull request* (also referred to as a merge request) is a feature offered by a hosting service that allows you to share work you have done on a branch with your collaborators, potentially gather feedback on that work, and finally integrate that work into the project remotely on the hosting service. Although pull requests are not a feature of Git but of the hosting services that host projects using Git, they are so useful in day-to-day work with Git that I wanted to introduce them.

Pull requests can be integrated by merging or rebasing, but the default (and most common) option is merging, so we will stick to that for the example in this chapter. Throughout the rest of the chapter I will refer to the process of integrating a pull request as “merging a pull request.”

When you create a pull request, you may say that you “open” a pull request. Once the pull request has been reviewed, approved, and merged, you “close” it. You may also close a pull request if you decide not to merge it and you want to remove it from the list of open pull requests (for example, if the pull request does not get approved).

The pull request process can be split into nine steps:

1. Create a branch in the local repository.
2. Add work by making commits on the branch.
3. Push the branch to the remote repository.

4. Create (or open) a pull request in the hosting service.
5. Get the pull request reviewed and potentially incorporate any feedback from other people into the pull request.
6. Get the pull request approved.
7. Merge the pull request.
8. If it is a topic branch (feature branch), delete the remote branch.
9. Pull the changes to sync your local repository with the remote repository, and clean up by deleting the local branch and the remote-tracking branch.

[NOTE]

If there are merge conflicts between the branches in the pull request, you will not be able to merge it. You must first resolve the merge conflicts. Some hosting services provide support for resolving merge conflicts on their website; however, it is most common to resolve merge conflicts in your local repository, using the process outlined in [Chapter 10](#).

Before we get into an example, let's briefly touch upon some things you'll need to keep in mind when working with a hosting service.

Hosting Service Specifics

The specific steps for creating and managing pull requests with GitHub, GitLab, and Bitbucket are different, and you may need to consult your hosting service's documentation to complete some of the Follow Along exercises in this chapter. For additional resources, go to the Learning Git repository (<https://github.com/gitlearningjourney/learning-git>).

Terminology may also differ between hosting services: notably, while GitHub and Bitbucket use the term “pull request,” GitLab refers to the same feature as a “merge request.” Keep these differences in mind when carrying out the Follow Along exercises in this chapter.

[NOTE]

Even though “pull” is in the name, pull requests are not actually related to the `git pull` command.

Finally, in [Chapter 6](#), when we walked through choosing a hosting service and setting up HTTPS or SSH access, I recommended that you use a personal hosting service account rather than a company account. This is because it is possible to configure additional settings in a hosting service for the pull request creation and approval process. For example, you (or the company you work for) may define certain requirements for creating a pull request, or you may set restrictions on who is able to approve pull requests or how many individuals in a team need to approve a pull request before it can be merged.

The examples in this book assume that no extra settings have been configured. However, keep this in mind if you’re using a company account, as there may be additional requirements and restrictions in place that affect your ability to create and manage pull requests.

Next, let’s discuss why you might want to use pull requests in the first place.

Why Use Pull Requests?

Pull requests facilitate communication and collaboration on Git projects by providing an easy mechanism to review work. They have a useful commenting feature that allows you and your collaborators to add comments to specific lines in the files of a project, respond to these comments, and start discussion threads. This helps organize the review process.

Since pull requests are managed entirely in the hosting service UI, they also allow non-Git users to provide feedback on Git projects—you don't need to know how to use Git to make comments on a pull request! Let's look at [Example Book Project 12-1](#) to see an example of how using pull requests can be useful.

Example Book Project 12-1

Suppose I make a branch off the `main` branch called `chapter_nine` to work on chapter 9 of my book. I make two commits on that branch, commit W and commit X, then I push my local `chapter_nine` branch to the remote repository, creating a remote `chapter_nine` branch. The state of the local and remote repositories is illustrated in [Figure 12-1](#).

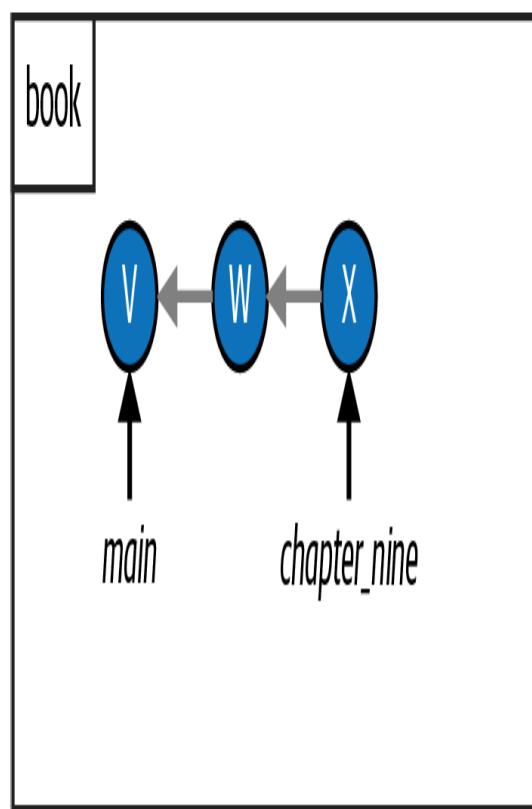
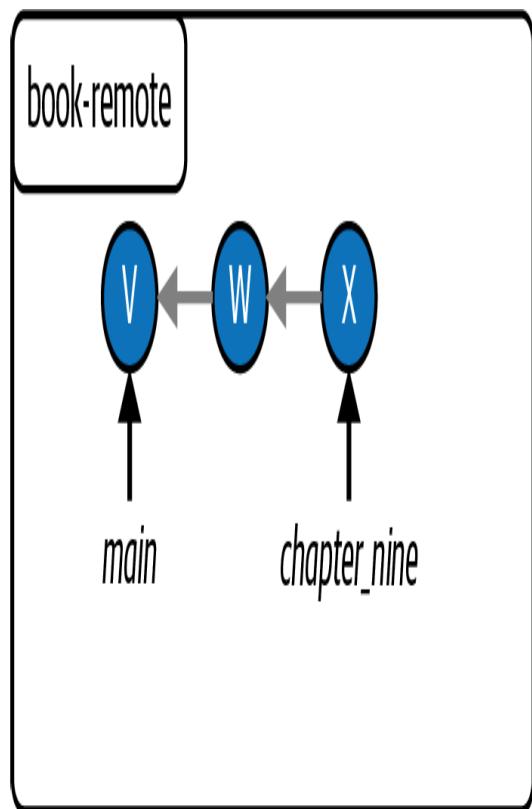


FIGURE 12-1

The Book project after I push the `chapter_nine` branch to the remote repository

Recall that I've agreed with my editor that they must review my work before I merge it into the `main` branch. This means they need to review the new `chapter_nine` branch. One option is for them to clone the remote repository on their local computer and check out the branch there, in order to view my updated file. However, this doesn't give them an easy way of providing me with feedback and comments. Also, let's assume that my editor hasn't yet learned how to use Git and therefore would struggle with cloning and working in a local repository.

Instead, I'll make sure my editor has access to the remote repository, then I'll make a pull request in the remote repository to merge the remote `chapter_nine` branch into the remote `main` branch. I can either send my editor the URL to the pull request or simply tell them to go to the remote repository and find the pull request titled "Chapter 9 updates."

My editor can then use the commenting feature to ask questions and provide feedback. Suppose they notice an inconsistency in the chapter that I need to fix, so they leave a comment in the pull request and let me know that I should review their feedback. After fixing the issue in the chapter in my local repository, I make another commit on the local `chapter_nine` branch (commit Y) and push it to the remote repository. This will automatically update the remote `chapter_nine` branch, and it will therefore also automatically update the pull request. [Figure 12-2](#) shows the updated state of the local and remote repositories.

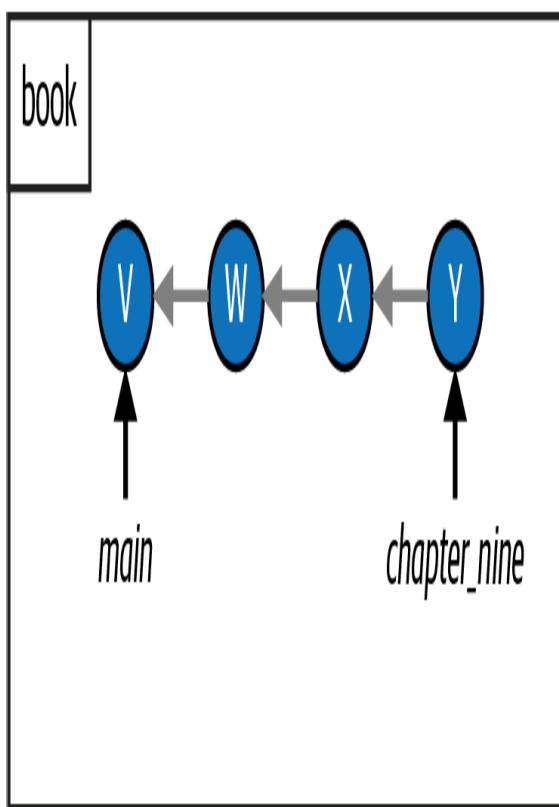
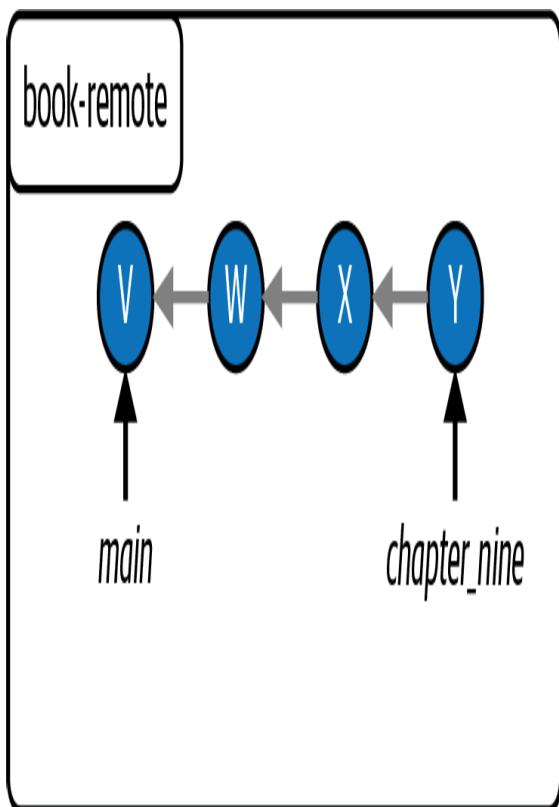


FIGURE 12-2

The Book project after I make commit Y on the `chapter_nine` branch and push it to the remote repository

The fact I made the pull request makes it easy for my editor to provide me with feedback on my Book project, and for me to share my updates after I incorporate their feedback.

[Example Book Project 12-1](#) illustrates why pull requests are so useful.

Next, let's take a look at how pull requests actually integrate work.

Understanding How Pull Requests Are Merged

As you know, there are two types of merges in Git: a *fast-forward merge* happens when the developmental histories of the source branch and the target branch have not diverged, whereas if they have, a *three-way merge* happens and a merge commit is made. You walked through examples of performing both kinds of merges in Chapters [5](#) and [9](#), respectively.

By default, merging remotely is different from merging locally. The default setting for most hosting services is that a remote merge with a pull request happens with a merge option called *non-fast-forward*. With this option, even if the development histories of the source branch and the target branch have *not* diverged, a merge commit will *still* be made.

Merges made with the non-fast-forward option are sometimes referred to as *explicit merges*. This is because they always explicitly show where a merge happened with a merge commit.

It is possible to change the setting on a hosting service to merge with a different option or approach. However, since the non-fast-forward option is the most common, that's what we will cover in this book. Let's revisit the scenario from [Example Book Project 12-1](#) in [Example Book Project 12-2](#) to see how pull requests usually merge work.

Example Book Project 12-2

In [Example Book Project 12-1](#), I described how I made a pull request titled “Chapter 9 updates” so my editor could review the work I had done on the `chapter_nine` branch that I wanted to merge into the `main` branch. [Figure 12-3](#) shows the state of the Book project at this point.

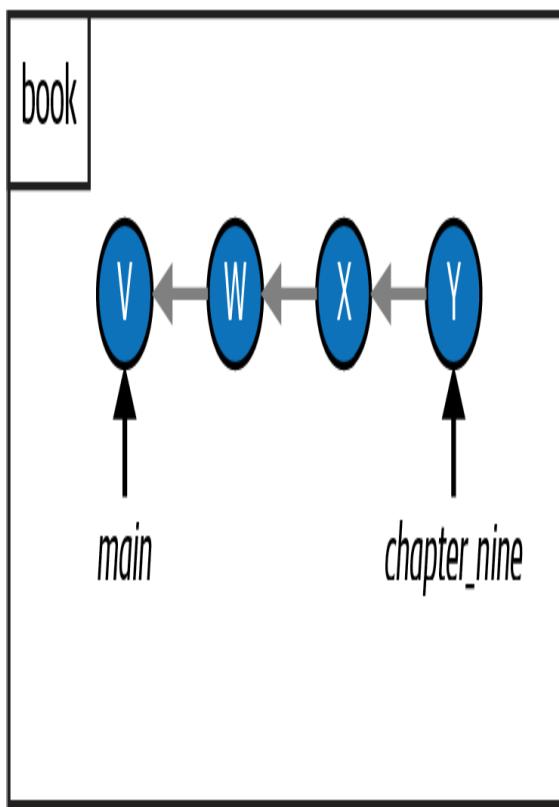
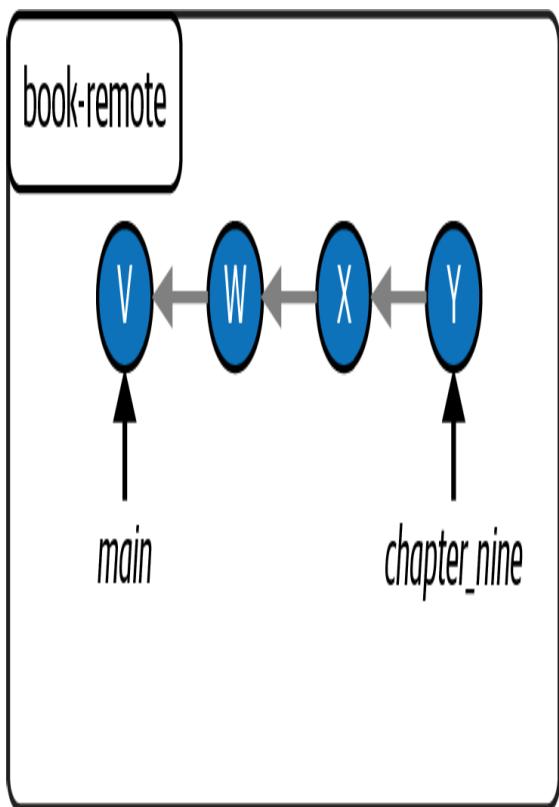


FIGURE 12-3

The state of the local and remote repositories in the Book project

After my editor reviews the latest work that I pushed to the remote `chapter_nine` branch, which is represented by commit Y, they no longer have any more feedback for me. So, they select the button on the hosting service's website to approve the pull request.

This means that I can merge the remote `chapter_nine` branch into the remote `main` branch by selecting the button on the website to merge the pull request. Since I have the default settings in place, this will be a non-fast-forward merge. Even though the development histories of the remote `chapter_nine` branch and the remote `main` branch have not diverged, a merge commit will still be made. This is represented as commit M in [Figure 12-4](#).

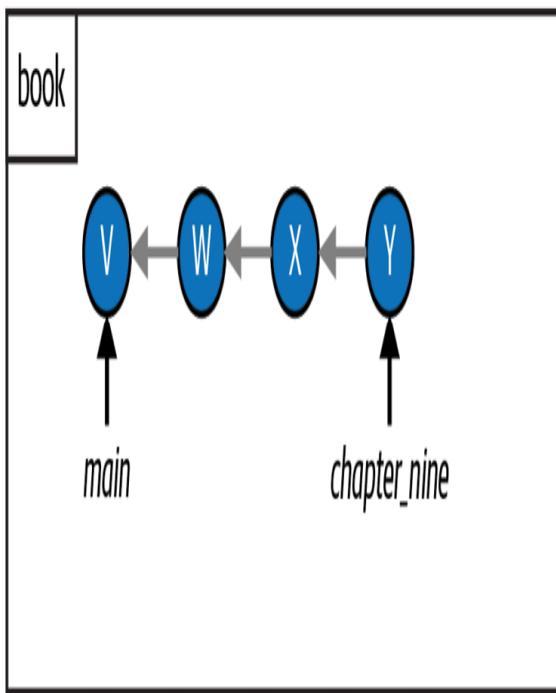
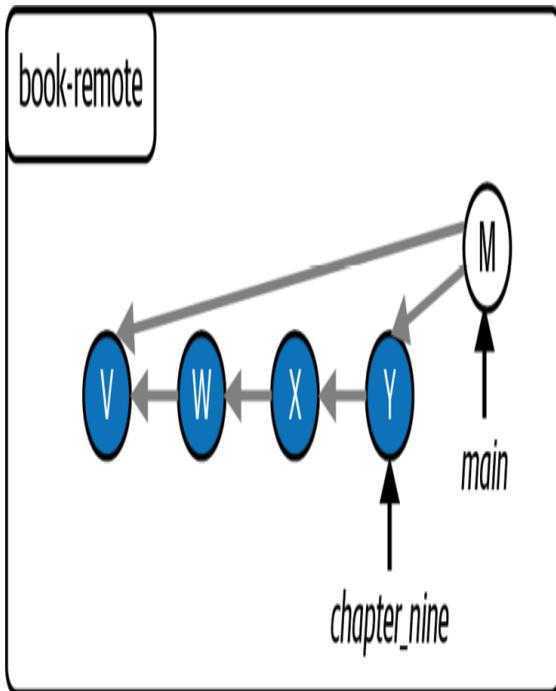


FIGURE 12-4

The Book project after I merge the pull request to merge the `chapter_nine` branch into the `main` branch

The parent commits of the merge commit M are commit V, where the `main` branch was pointing before the merge, and commit Y, the latest commit on the `chapter_nine` branch.

The final actions I need to take are to delete the `chapter_nine` branch and pull the most up-to-date version of the `main` branch from the remote repository to my local repository, so that my local `main` branch is in sync with the remote repository. This is illustrated in [Figure 12-5](#).

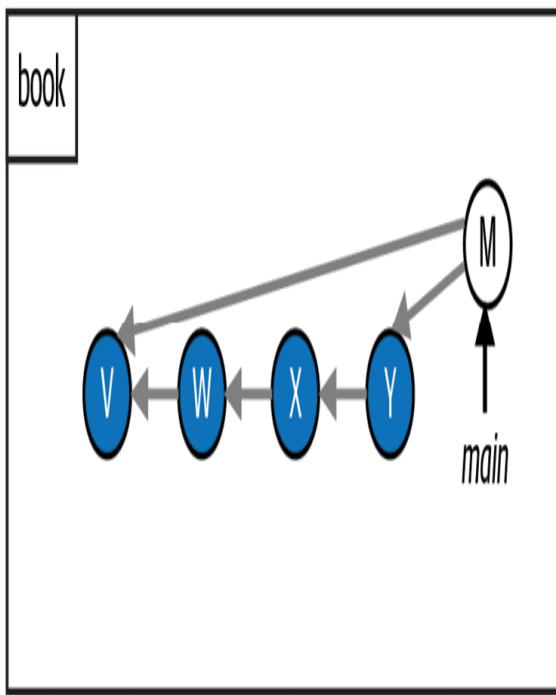
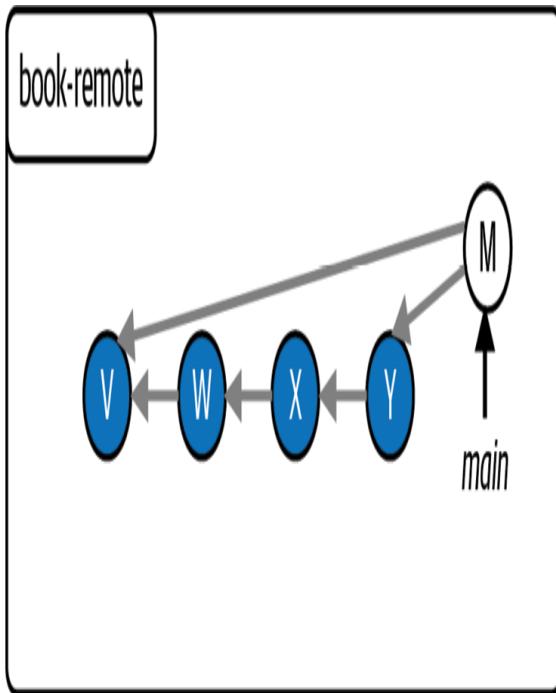


FIGURE 12-5

The Book project after I delete the `chapter_nine` branch and pull the changes from the remote `main` branch to my local `main` branch

Now that we have covered an example of a pull request in [Example Book Project 12-2](#), let's turn to the Rainbow project so you can practice going through the pull request process.

Preparing to Make a Pull Request

To go over an example of a pull request, you will start by completing steps 1 and 2 of the pull request process (described in [“Introducing Pull Requests” on page 233](#)) in the `rainbow` repository.

In [Follow Along 12-1](#), you will use the `git switch` command with the `-c` option, introduced in [“Creating a Branch and Switching onto It in One Go” on page 86](#), to create and immediately switch onto a new branch called `topic`. Then you will add work to it.

[NOTE]

The reason you will use the generic name `topic` for the new branch is because, as mentioned in [“Why Do We Use Branches?” on page 44](#), it is common to refer to branches that are created to work on a specific part of a project in Git as *topic branches* (or *feature branches*). In a real Git project, the branch name would usually contain a brief description of the feature or topic you're working on.

[FOLLOW ALONG 12-1]

1 rainbow \$ **git switch -c topic**

Switched to a new branch 'topic'

2 Open the `rainbow` project directory in your text editor. In the `othercolors.txt` file, on line 4, add "Pink is not a color in the rainbow." and save the file.

3 rainbow \$ **git add othercolors.txt**

4 rainbow \$ **git commit -m "pink"**

[topic 4c35a5c] pink

1 file changed, 1 insertion(+), 1 deletion(-)

5 rainbow \$ **git log**

commit 4c35a5c02c3dc03f044cbdfdbb0ae55161af6a86 (**HEAD -> topic**)

Author: annaskoulikari <gitlearningjourney@gmail.com>

Date: Sun Jul 3 14:16:01 2022 +0200

pink

commit 7c09136bcbfdd9f638ed13c6653e06451579d21c (**origin/main, main**)

Author: annaskoulikari <gitlearningjourney@gmail.com>

Date: Sun Feb 20 10:10:11 2022 +0100

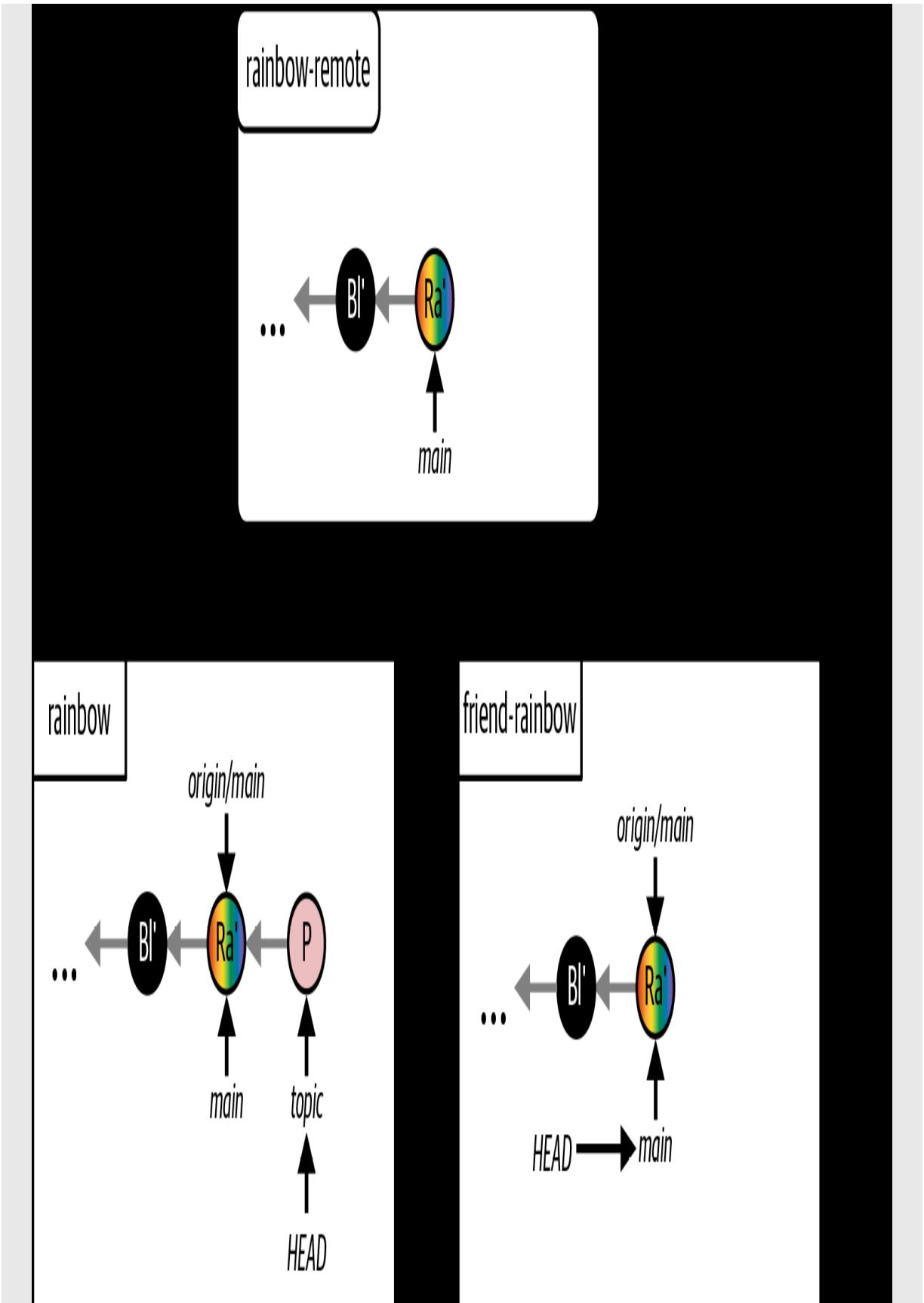
rainbow

What to notice:

- You created the pink commit on the `topic` branch in the `rainbow` repository.

This is illustrated in [Visualize it 12-3](#).

[VISUALIZE IT 12-3]



The Rainbow project after you make the pink commit on the `topic` branch in the `rainbow` repository

The `topic` branch is a new local branch. It does not have an upstream branch defined for it. (Recall that an *upstream branch* is the remote branch that a particular local branch tracks; you learned about this concept in [Chapter 7](#), and you learned about defining an upstream branch in [Chapter 9](#).)

Next, you will carry out step 3 of the pull request process, which is to push your work to the remote repository. In the process, I'll introduce an easier way to define upstream branches.

An Easier Way to Define Upstream Branches

In [Chapter 9](#), you learned how to use the `git branch -u <shortname>/<branch_name>` command to define an upstream branch. But now, as we near the end of this book, I'll let you in on a little secret. It's really common for Git users to forget to define upstream branches. Or to skip setting them up out of sheer laziness.

However, there is a helpful trick that Git users often use to define upstream branches when they first push a new branch to a remote repository. If you use the `git push` command without any arguments on a branch that does *not* have an upstream branch defined, then Git will issue a warning in the output of the command. The warning will provide you with the command to use to set one (`git push --set-upstream <shortname> <branch_name>`), and it will suggest a shortname and branch name that you might want to use (Git assumes that you'll want the remote branch to have the same name as the

local branch). If you copy and paste this command into the command line and execute it, then you can accomplish two tasks in one go: you push the local branch to the remote repository and define an upstream branch for it.

Go to [Follow Along 12-2](#) to carry out a hands-on example of this.

[FOLLOW ALONG 12-2]

1 rainbow \$ **git branch -vv**
main 7c09136 [origin/main] rainbow
* **topic** 4c35a5c pink

2 rainbow \$ **git push**
fatal: The current branch topic has no upstream branch.
To push the current branch and set the remote as upstream, use
git push --set-upstream origin topic

3 Copy the `git push --set-upstream origin topic` command included in the output of the `git push` command in step 2, and paste it in the command line.

4 rainbow \$ **git push --set-upstream origin topic**
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 314 bytes | 314.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'topic' on GitHub by visiting:
remote: https://github.com/gitlearningjourney/rainbow-
remote/pull/new/topic
remote:
To github.com:gitlearningjourney/rainbow-remote.git
* [new branch] topic -> topic
branch 'topic' set up to track 'origin/topic'.

[FOLLOW ALONG 12-2]

```
5 rainbow $ git branch -vv
  main 7c09136 [origin/main] rainbow
* topic 4c35a5c [origin/topic] pink

6 rainbow $ git log
commit 4c35a5c02c3dc03f044cbdfdbb0ae55161af6a86 (HEAD -> topic,
origin/topic)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sun Jul 3 14:16:01 2022 +0200

pink

commit 7c09136bcbfdd9f638ed13c6653e06451579d21c (origin/main, main)
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sun Feb 20 10:10:11 2022 +0100

rainbow
```

7 Go to the `rainbow-remote` repository on your hosting service and refresh the page. You should see the new `topic` branch you created and if you select to view the commits on the `topic` branch, you will also see the pink commit.

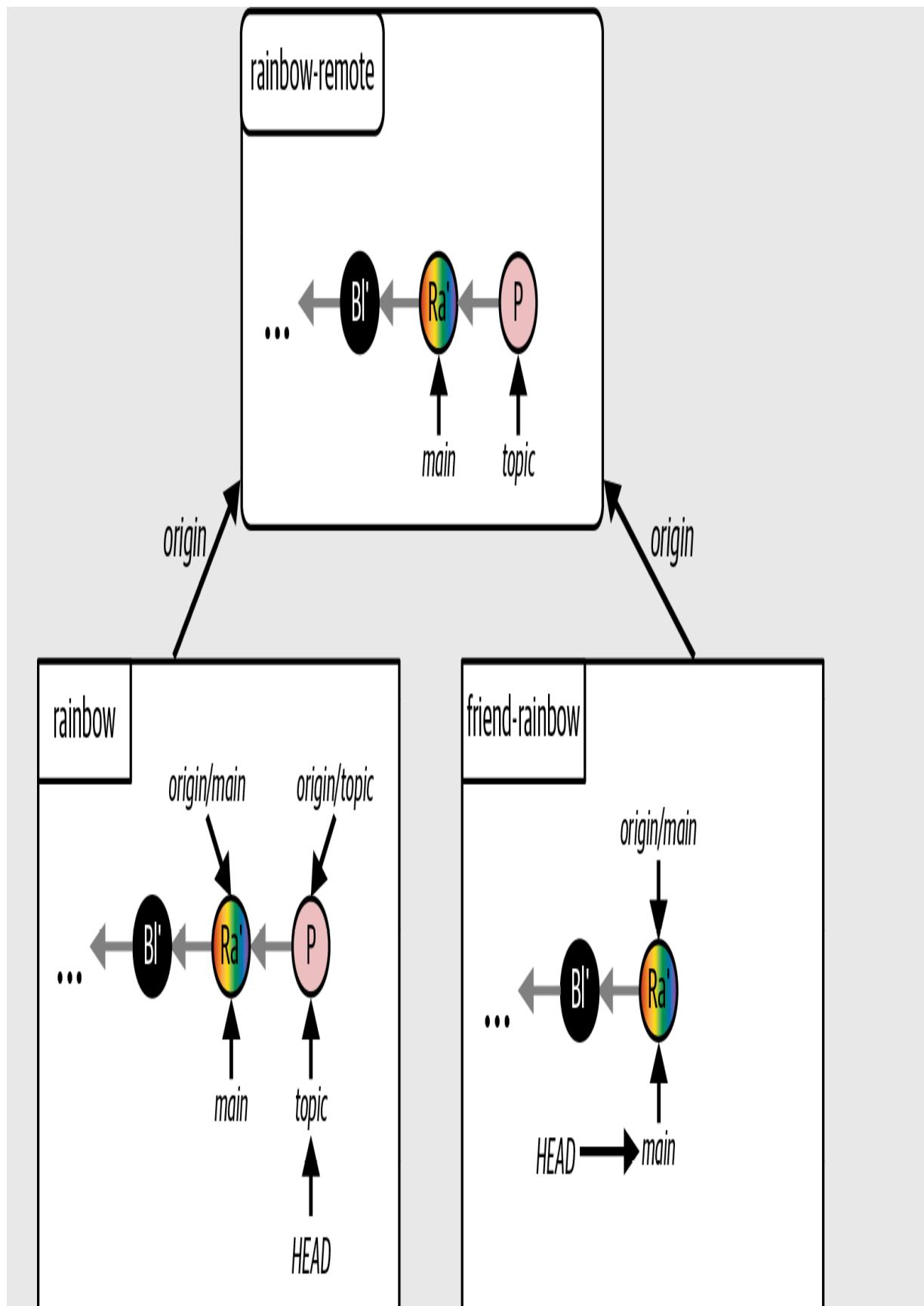
What to notice:

- In step 2, the output of the `git push` command provides a warning: `The current branch topic has no upstream branch.` It also provides the following instructions: `To push the current branch and set the remote as upstream, use git push --set-upstream origin topic.`
- In step 4, the `git push` output recommends that you make a pull request for the new branch: `Create a pull request for 'topic' on GitHub by visiting:`

<https://github.com/gitlearningjourney/rainbow-remote/pull/new/topic>.

The current state of the repositories after [Follow Along 12-2](#) is illustrated in [Visualize it 12-4](#).

[VISUALIZE IT 12-4]



The Rainbow project after you push the `topic` branch to the `rainbow-remote` repository and define the upstream branch for the local `topic` branch in one go

You just saw how you can use the `git push` command to easily define an upstream branch while pushing a branch to the remote repository. Next, you will carry out step 4 of the pull request process.

[NOTE]

This trick assumes that you want the upstream branch of your local branch to be a remote branch of the same name in the remote repository. If this is not the case, you may have to edit the `git push` command that Git provides you to indicate the alternative remote branch that you want to set as the upstream branch.

Creating a Pull Request on a Hosting Service

When you create a pull request, you have to define the source branch and the target branch. In the Rainbow project example, `topic` is the source branch and `main` is the target branch. In other words, you are merging `topic` into `main`. To create the pull request, you will have to use your hosting service's UI. For additional resources, go to the Learning Git repository (<https://github.com/gitlearningjourney/learning-git>).

On the web page to create the pull request, you will have the opportunity to enter information about the pull request. The only required field for most hosting services is a title. As with commit messages and the names of branches, if you're working with a team on a Git project you should check

with them about whether they have conventions established for pull request titles. For the Rainbow project example in this book you will use the “Adding the color pink” and leave all the other fields empty.

In GitLab and Bitbucket, when making a pull request you may also select an option that will automatically delete the branch that you’re merging after the pull request is merged. It’s up to you whether you want to select this option (if it is available on your hosting service). In any case, after you merge the pull request you will make sure the remote branch is deleted (if it is still there), as this is step 8 in the pull request process.

Now, go to [Follow Along 12-3](#) to create your pull request.

[FOLLOW ALONG 12-3]

- 1 Go to your remote repository on your hosting service. Follow the steps to create a pull request to merge `topic` (the source branch) into `main` (the target branch). The hosting service may prefill the pull request title field with the commit message of your last commit, which in your case is “pink”. You should change the title to “Adding the color pink”. For additional resources, go to the Learning Git repository (<https://github.com/gitlearningjourney/learning-git>).

Now that you have created your first pull request, it’s time to move on to steps 5 and 6 of the pull request process and get it reviewed and approved.

Reviewing and Approving a Pull Request

Pull requests provide an opportunity for collaborators on a project to review your work, contribute to it, or simply approve it. The hosting service UI

usually provides a way to view the lines that have been changed in each file that has been modified and how they have been changed, using colors and symbols to display this information. [Figure 12-6](#) shows how a hosting service might indicate that you added a sentence on line 4 of the `othercolors.txt` file about the color pink.

othercolors.txt	
1. Brown is not a color in the rainbow. 2. Gray is not a color in the rainbow. 3. - Black is not a color in the rainbow.	1. Brown is not a color in the rainbow. 2. Gray is not a color in the rainbow. 3. + Black is not a color in the rainbow. 4. + Pink is not a color in the rainbow.

FIGURE 12-6

An example of how a hosting service displays the files changed in a pull request

When collaborators review your work, they can choose to leave comments on the pull request. If your pull request has comments, you can review them, potentially make changes, and push one or more additional commits to the branch that you are merging. This will automatically update the pull request with the new commits.

A collaborator can also decide to pull the branch to their local repository and make additional changes on the branch by adding commits and pushing them to the remote repository. Again, the pull request will automatically update with their additional commits.

In the Rainbow project example, suppose you've told your friend that you made the pull request and asked them to review it. Your friend will go to the pull request and take a look at it, decide that they're happy with the changes you made as they are, and approve the pull request.

Normally if you had a collaborator reviewing your pull request, they would log into their own account on the hosting service, then review and approve it by selecting the “approve” button in the hosting service’s UI. However, since you’re simulating that you are two people, this “approve” button might not appear in the UI. In this case, you will just have to pretend that your friend approves the pull request.

Go to [Follow Along 12-4](#) now to simulate your friend reviewing and approving the pull request.

[FOLLOW ALONG 12-4]

1 Pretend you are your friend, and go to the pull request on your hosting service.

2 In the pull request, view the files that were changed. If necessary, consult the hosting service’s documentation for details on how to do this.

3 “Approve” the pull request.

Once the pull request has been approved (figuratively, if not literally), you may proceed to step 7 of the pull request process.

Merging a Pull Request

Acting as yourself again, you can now go back to your hosting service and carry out the steps to merge the pull request. This usually just consists of going to the pull request in the UI and selecting the relevant button. Go to [Follow Along 12-5](#) to merge your pull request now.

[FOLLOW ALONG 12-5]

1 Follow the steps for your hosting service to merge the pull request.

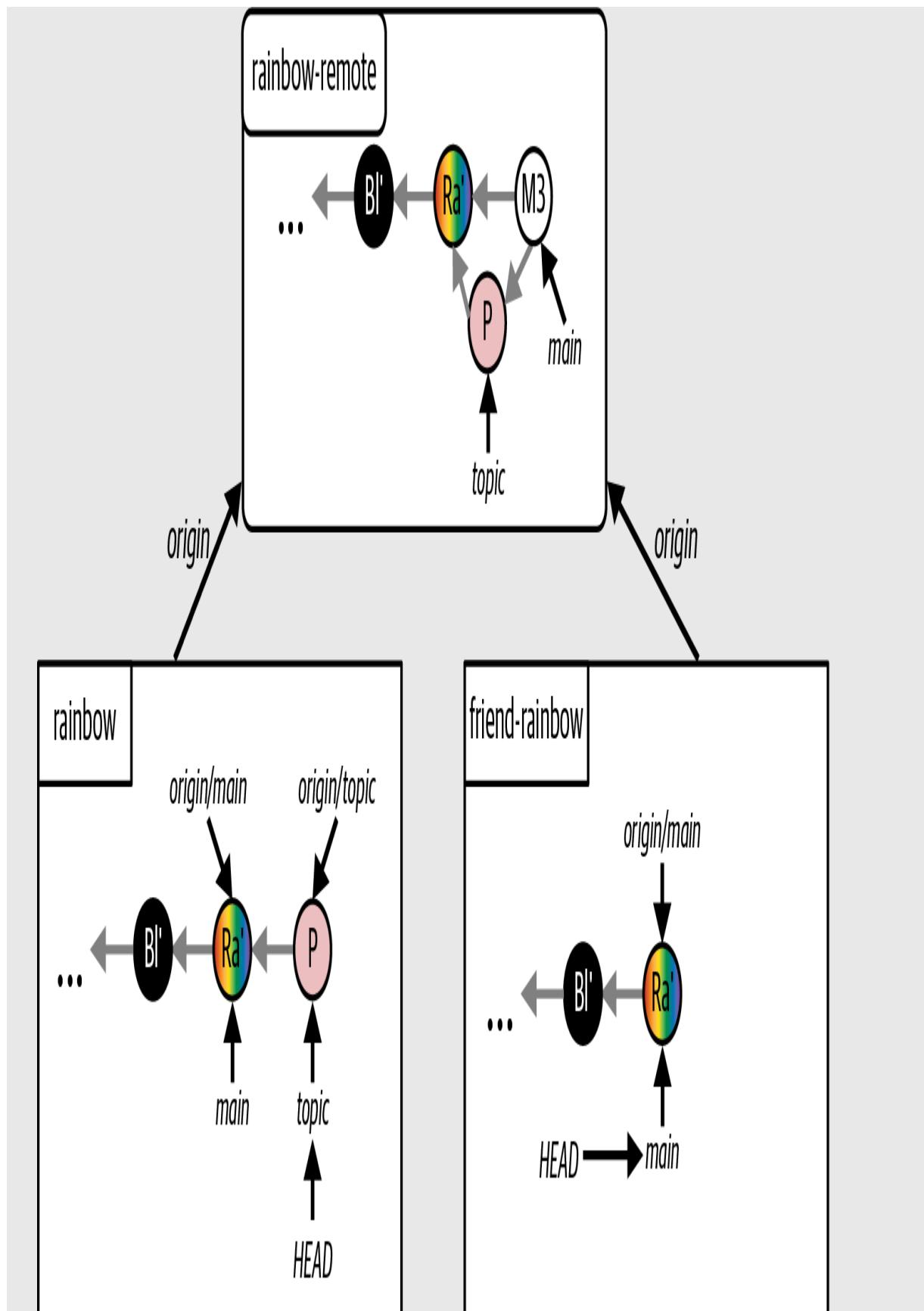
2 Review the list of commits in the remote repository. Find the latest merge commit and select it. Make a note of its commit hash as well as the commit hashes of its two parent commits.

What to notice:

- You merged the remote `topic` branch into the remote `main` branch and created a merge commit.

This is represented in [Visualize it 12-5](#); the new merge commit is labeled M3.

[VISUALIZE IT 12-5]



The Rainbow project after you merge the pull request in the remote repository

As mentioned earlier in this chapter, this merge is performed with the default non-fast-forward option. This means that even though the development histories of the remote `topic` branch and the remote `main` branch had not diverged, the merge still created a merge commit.

In [Visualize it 12-5](#), you can see that the remote `topic` branch is still in the `rainbow-remote` repository and the local `topic` branch and `origin/topic` remote-tracking branch are still in the `rainbow` repository. Moving on to the eighth step of the pull request process, let's go ahead and delete the remote `topic` branch.

Deleting Remote Branches

If the branch you have merged in a pull request is a topic branch, it is common to delete it after the pull request is merged because you can assume that the work on that branch has been completed. For new pieces of work, you will make new branches and go through the pull request process again. This keeps the remote repository organized and uncluttered by old branches.

When you created your pull request, if your hosting service offered you the option of automatically deleting the branch once the pull request was merged, then you may no longer have a remote `topic` branch. If this is not the case, or you chose not to select that option, go on to [Follow Along 12-6](#) to delete the remote `topic` branch.

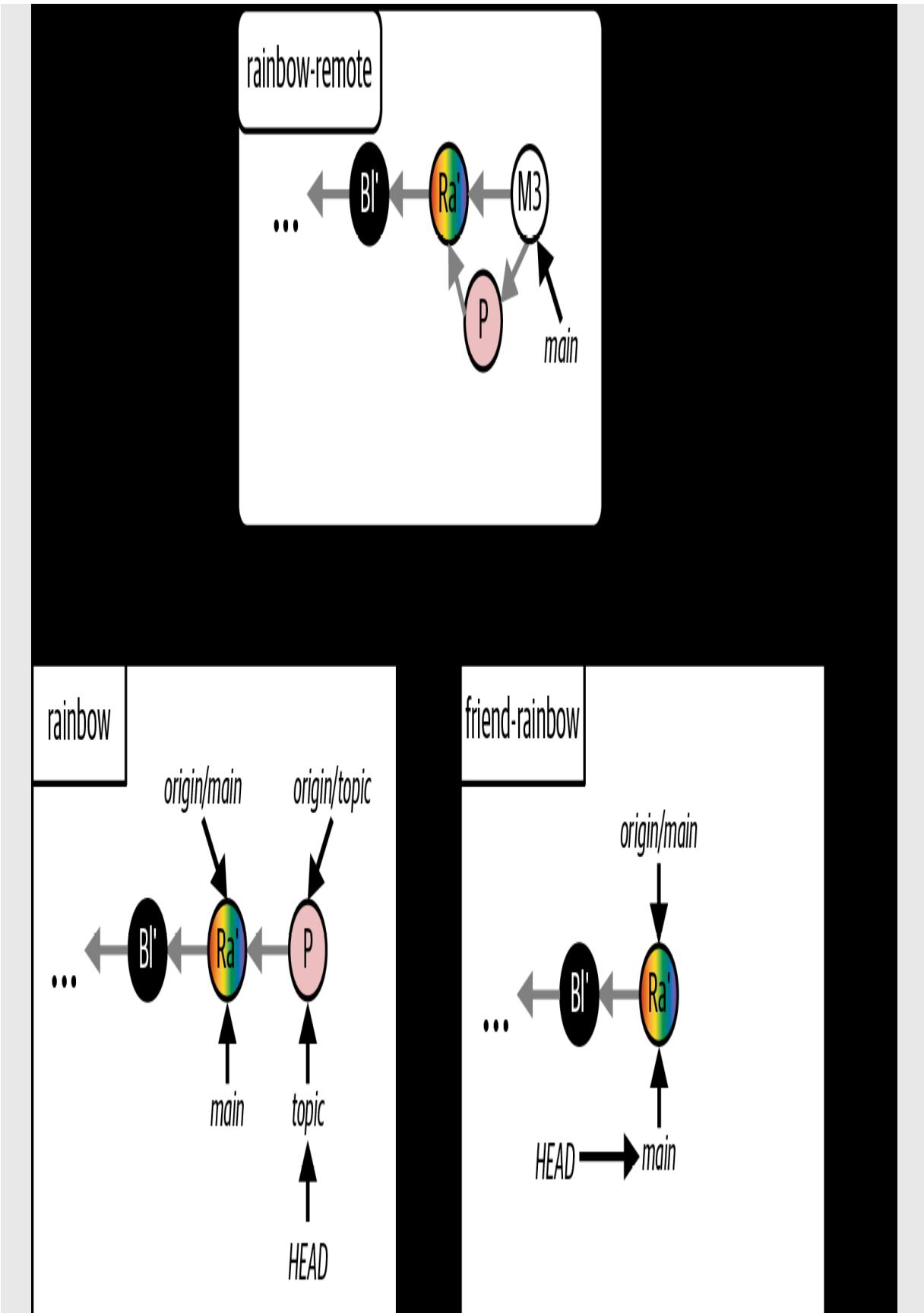
[FOLLOW ALONG 12-6]

1

Go to your remote repository on your hosting service and delete the remote `topic` branch. If necessary, consult the hosting service's documentation for details on how to do this.

[Visualize it 12-6](#) illustrates the state of the repositories after the deletion of the remote `topic` branch.

[VISUALIZE IT 12-6]



The Rainbow project after you delete the remote `topic` branch

As you can see, the local `main` branches in both of the local repositories are now out of sync with the `main` branch in the remote repository. Also, the local `topic` branch and `origin/topic` remote-tracking branch are still in the `rainbow` repository. Next, you'll make sure everyone pulls the latest changes from the remote repository so that the primary line of development in the Rainbow project, the `main` branch, is up to date in both local repositories. You'll also clean up the `rainbow` repository by deleting the local `topic` branch and the `origin/topic` remote-tracking branch.

Syncing the Local Repositories and Cleaning Up

The final step of the pull request process is for you and your friend to sync your local repositories with the remote repository. First you will sync the `rainbow` repository, and since the `topic` branch has been merged you will also delete the local `topic` branch and the `origin/topic` remote-tracking branch.

Recall that in [Chapter 8](#), you used the `git fetch` command with the `-p` (which stands for “prune”) option to delete any remote-tracking branches that correspond to remote branches that have been deleted in the remote repository. This time around you will use the `-p` option with the `git pull` command, and it will have the same effect. Go to [Follow Along 12-7](#) to sync the `rainbow` repository.

[FOLLOW ALONG 12-7]

1 rainbow \$ **git switch main**

Switched to branch 'main'

Your branch is up to date with 'origin/main'.

2 rainbow \$ **git pull -p**

From github.com:gitlearningjourney/rainbow-remote

- [deleted] (none) -> origin/topic

remote: Enumerating objects: 1, done.

remote: Counting objects: 100% (1/1), done.

Unpacking objects: 100% (1/1), 626 bytes | 626.00 KiB/s, done.

remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0

7c09136..2f833d6 main -> origin/main

Updating 7c09136..2f833d6

Fast-forward

othercolors.txt | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)

3 rainbow \$ **git branch -d topic**

Deleted branch topic (was 4c35a5c).

[FOLLOW ALONG 12-7]

4

```
rainbow $ git log
commit 2f833d6fa783882c5f832da9e1eafe6d405d3468 (HEAD -> main,
origin/main)

Merge: 7c09136 4c35a5c

Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Mon Jul 4 05:50:08 2022 +0200

    Merge pull request #1 from gitlearningjourney/topic

        Adding the color pink

commit 4c35a5c02c3dc03f044cbdfdbb0ae55161af6a86

Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sun Jul 3 14:16:01 2022 +0200

    pink
```

What to notice:

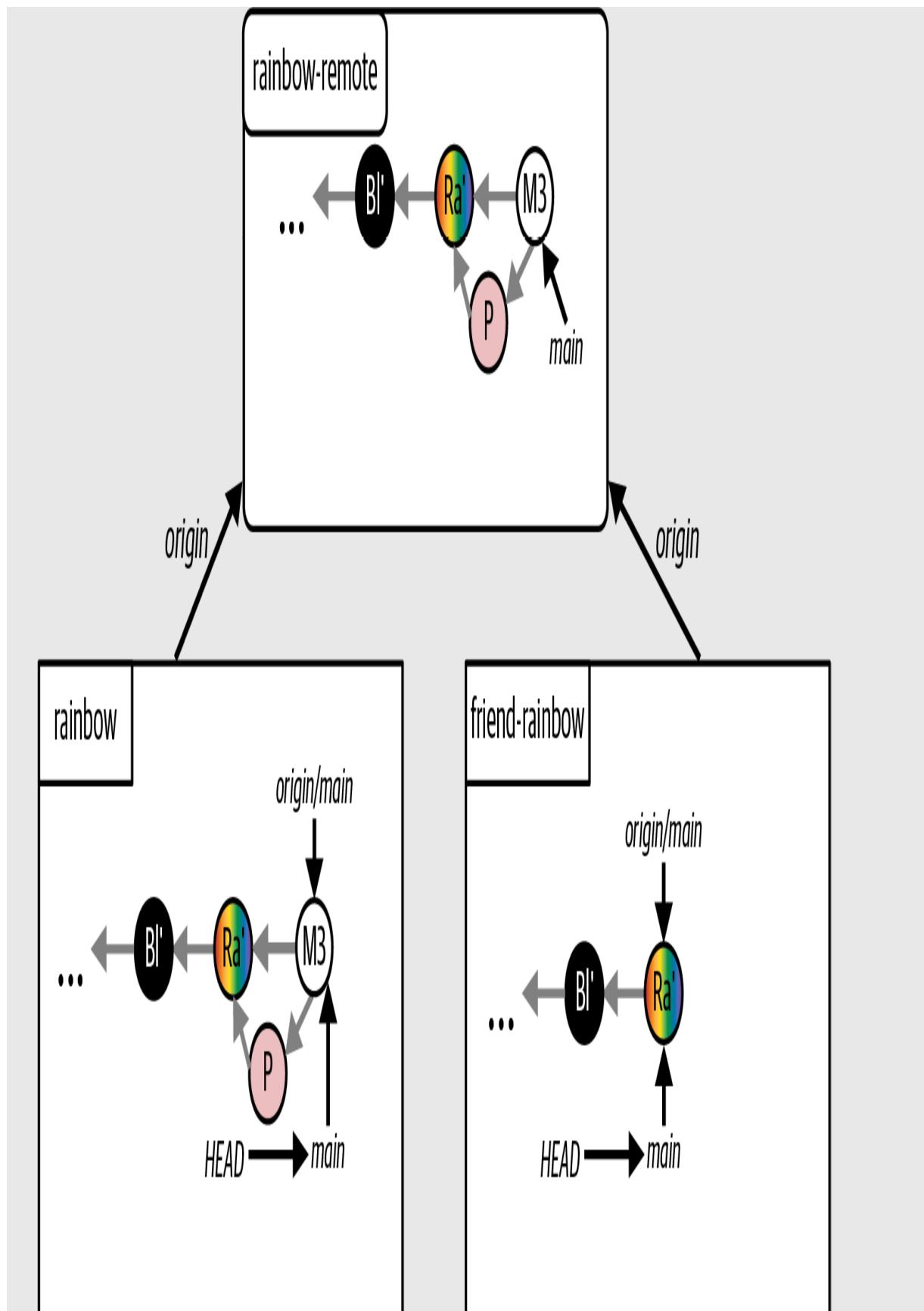
- You deleted the local `topic` branch and the `origin/topic` remote-tracking branch.
- You updated your local `main` branch.
- The merge commit (M3) has a commit message and a description that were automatically generated by the hosting service. In the example in this book the commit message is `Merge pull request #1 from gitlearningjourney/topic` and the commit description is `Adding the color pink`, which was the title of the pull request you created. Each hosting

service may have a slightly different template on which they base this default commit message.

- The parents of the M3 merge commit are `7c09136` (the rainbow commit) and `4c35a5c` (the pink commit). As usual, the commit hashes in your repository will be different from the ones in this book because commit hashes are unique.

The state of the Rainbow project after [Follow Along 12-7](#) is illustrated in [Visualize it 12-7](#).

[VISUALIZE IT 12-7]



The Rainbow project after you pull the changes from the remote `main` branch to the local `main` branch in the `rainbow` repository and delete the local `topic` branch and the `origin/topic` remote-tracking branch

What to notice:

- In the `rainbow` repository, the `main` branch points to the latest M3 merge commit.

Next, in [Follow Along 12-8](#), your friend will sync their local `main` branch with the changes from the remote repository.

[FOLLOW ALONG 12-8]

1 friend-rainbow \$ **git pull**

```
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 1), reused 2 (delta 0), pack-reused 0
Unpacking objects: 100% (4/4), 831 bytes | 166.00 KiB/s, done.
From github.com:gitlearningjourney/rainbow-remote
   7c09136..2f833d6  main      -> origin/main
Updating 7c09136..2f833d6
Fast-forward
  othercolors.txt | 3 +-+
  1 file changed, 2 insertions(+), 1 deletion(-)
```

2 friend-rainbow \$ **git log**

```
commit 2f833d6fa783882c5f832da9e1eafe6d405d3468 (HEAD -> main,
origin/main, origin/HEAD)
Merge: 7c09136 4c35a5c
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Mon Jul 4 05:50:08 2022 +0200

Merge pull request #1 from gitlearningjourney/topic
```

```
Adding the color pink

commit 4c35a5c02c3dc03f044cbdfdbb0ae55161af6a86
Author: annaskoulikari <gitlearningjourney@gmail.com>
Date:   Sun Jul 3 14:16:01 2022 +0200

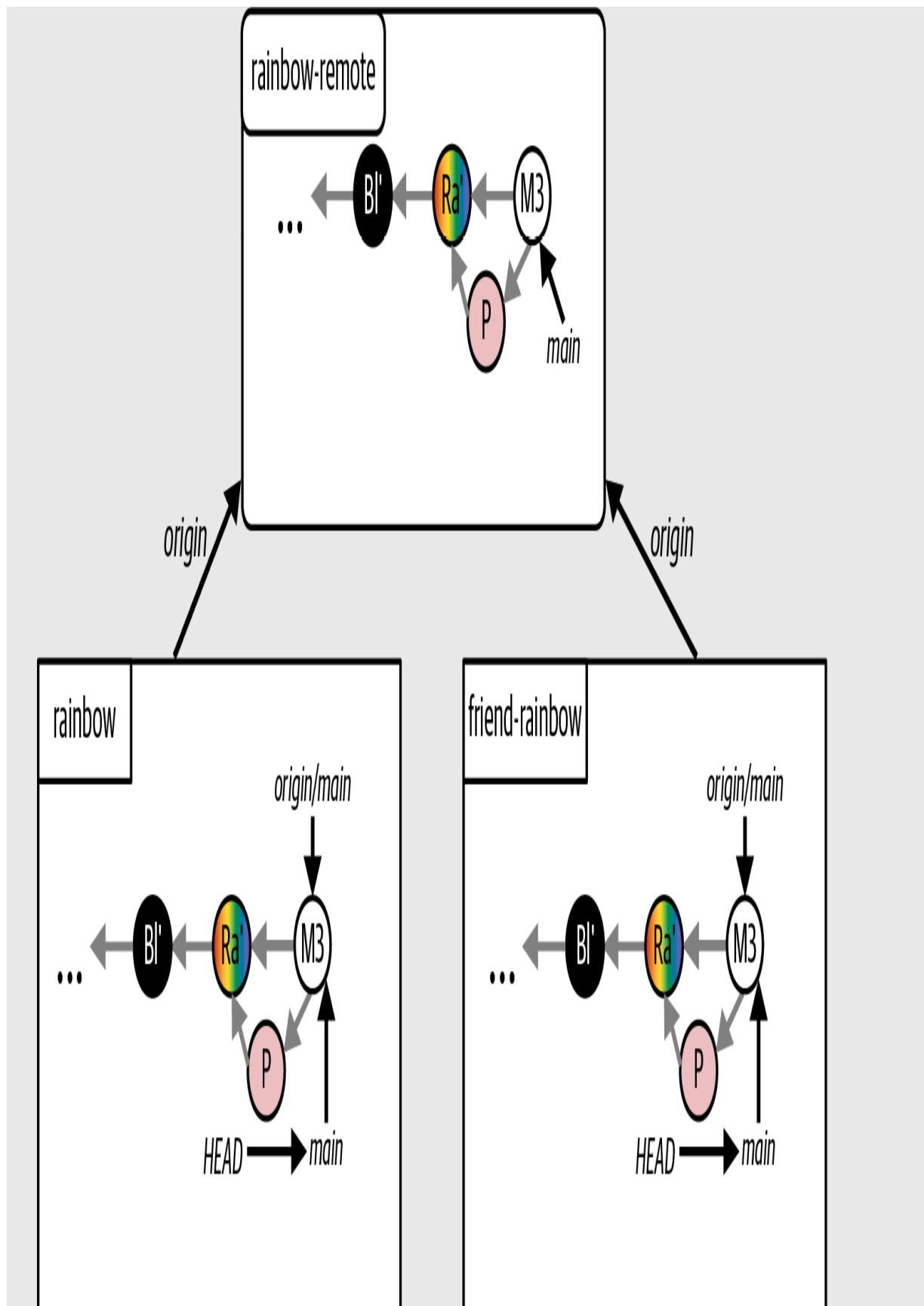
pink
```

What to notice:

- Your friend updated their local `main` branch.

This is illustrated in [Visualize it 12-8](#).

[VISUALIZE IT 12-8]

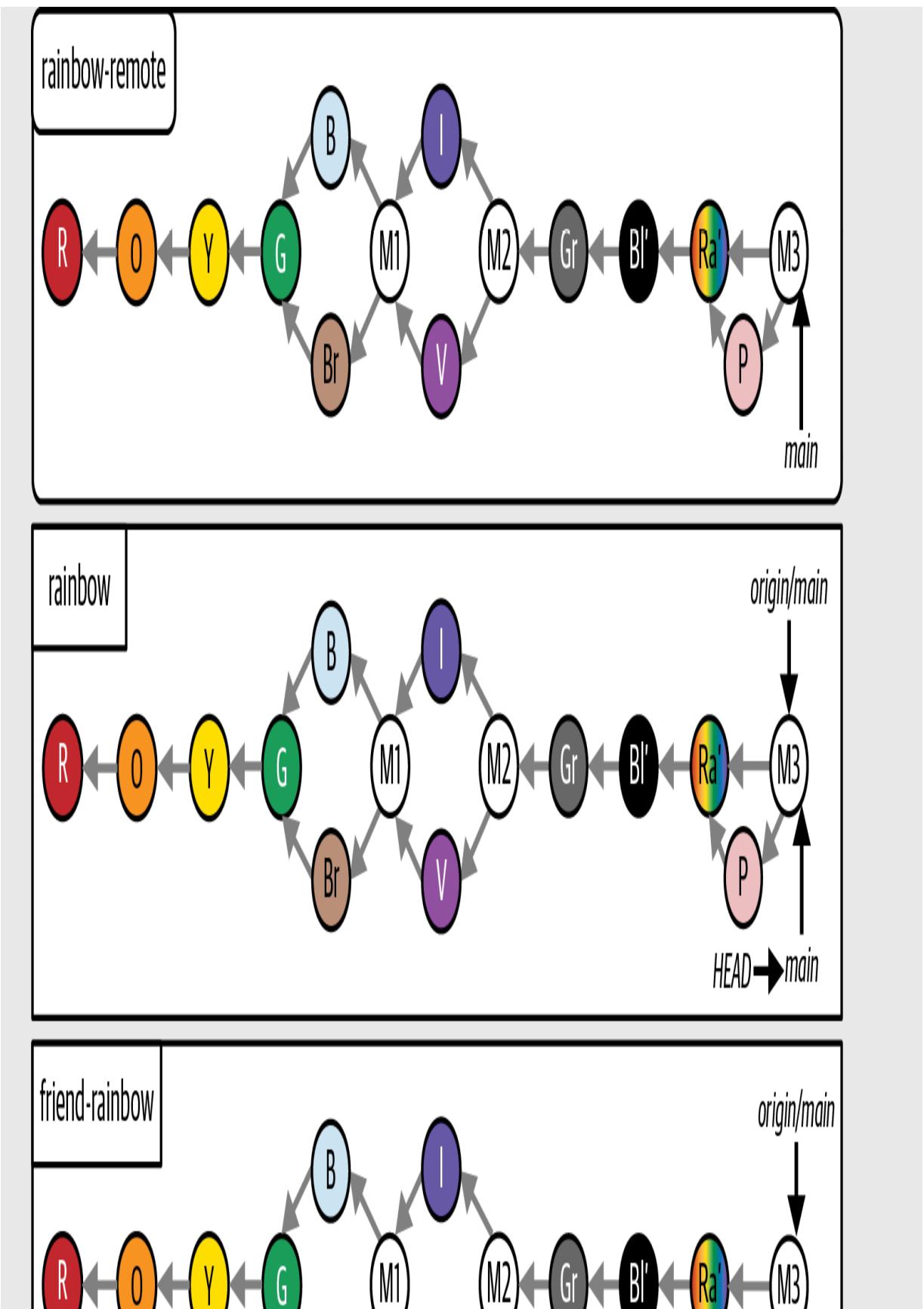


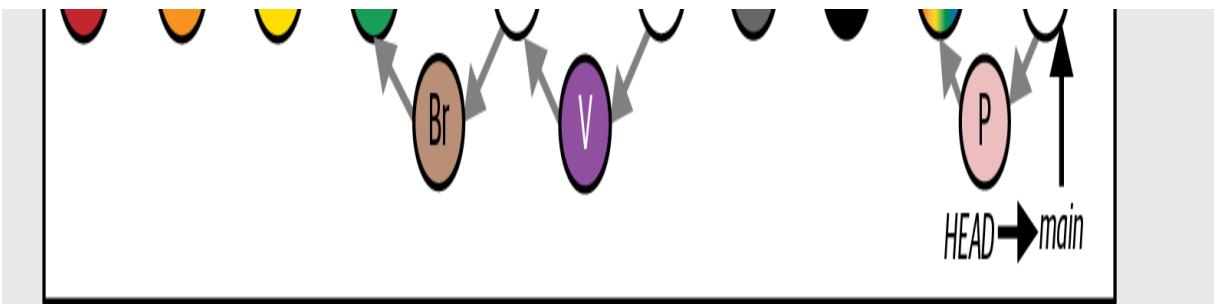
The Rainbow project after your friend pulls the changes from the remote `main` branch to the local `main` branch in the `friend-rainbow` repository

State of the Local and Remote Repositories

You have come to the end of the *Learning Git* experience. [Visualize it 12-9](#) shows the state of the local and remote repositories in the Rainbow project with all the commits that were made from [Chapter 1](#) all the way to the end of [Chapter 12](#).

[VISUALIZE IT 12-9]





The Rainbow project at the end of [Chapter 12](#) with all the commits since [Chapter 1](#)

Summary

In this chapter, you learned about pull requests. We discussed why they're a helpful tool for collaborating on Git projects with multiple people, because they facilitate communication and the review process. You carried out all nine steps of the pull request process in a hands-on example in the Rainbow project. While working through this example, you witnessed how by default pull requests are merged with the non-fast-forward option, which produces a merge commit even in the case of a fast-forward merge when the branches involved in the merge have not diverged. You also learned about a simple trick to get Git to generate a command you can use to push a branch to a remote repository and define an upstream branch for it in one go.

Check out the Epilogue for some closing thoughts, and be sure to take a look at the appendixes for some useful reference material.

[*Epilogue*]

Congratulations—you have reached the end of Learning Git! I hope this book has helped you in your Git learning journey and that you now feel you have a solid mental model of the basics of how Git works.

Although we’re at the end of the book, this is just the beginning of your Git adventure. The next step for you is to start using Git to version control your projects. In the process, you will learn to use many more of the features that exist in the world of Git. You’ll also need to figure out what Git workflow will work for you or your team, or, if you’re already part of a team, learn how their Git workflow works.

Now that you have gone through the entire learning journey working on the Rainbow project from Chapters [1](#) through [12](#), you may want to revisit the content in a specific chapter or try some of the hands-on exercises again. In that case, feel free to consult [Appendix A](#) to learn how to create the minimum setup for the Rainbow project that you’ll need to start fresh from any of the chapters.

Finally, if this book helped you, feel free to share it with someone else so it can help them on their Git learning journey.

[*Appendix A*]

Chapter Prerequisites

Learning Git is a hands-on learning experience designed to be read from [Chapter 1](#) through [Chapter 12](#) in a linear way. Throughout the book, you will work on the Rainbow project to learn how Git works.

However, there may be some situations in which you want or need to start from a specific chapter. For example:

- You have gone through the exercises in the entire book once and you want to review from a specific chapter onward.
- Something went wrong in the Rainbow project in a previous chapter that you were not able to troubleshoot, and you want to continue from a new chapter afresh.

In this case, you can use the instructions in the relevant section of this appendix to re-create an approximation of what the Rainbow project should look like at the start of the chapter you want to begin from. This will provide the minimum setup you need to be able to continue working on the Rainbow project from your chapter of choice. For more information and in-depth explanations about the steps in each Follow Along, refer to the relevant chapter's contents.

[NOTE]

For Chapters [9](#) to [11](#), you will not re-create all the commits that were made in the previous chapters. You will only re-create a commit similar to the last commit that was made before the chapter you want to start from. This means that your Rainbow project repositories will have fewer commits than the Rainbow project repositories illustrated in the Visualize It diagrams in the chapter contents. Keep this in mind while working through those chapters.

All of the instructions assume you have carried out the actions in [Chapter 1](#) on the computer you are using for the Rainbow project exercises at least once before. This consists of installing Git, choosing a text editor, and setting the `user.name` and `user.email` variables. If this is not the case, you'll need to start with the Follow Along in [“Prerequisite Setup for All Chapters” on page 260](#).

The instructions use the same names for the repositories that were used in the rest of the book: `rainbow`, `friend-rainbow`, and `rainbow-remote`. If you still have those repositories on your local computer and your hosting service, then you will have to use slightly different names; for example, `rainbow1`, `friend-rainbow1`, and `rainbow-remote1`. I recommend you stick with similar names, because these names are referred to throughout each chapter's contents.

Prerequisite Setup for All Chapters

If you don't have Git installed or a text editor ready to be used, complete the steps in [Follow Along A-1](#). Otherwise, you may skip it.

[FOLLOW ALONG A-1]

- 1** Go to the Learning Git repository (<https://github.com/gitlearningjourney/learning-git>) and follow the steps to download Git for your operating system.
- 2** Choose your preferred text editor. If you don't already have a text editor on your computer, then download one. For more information, see "[Preparing a Text Editor](#)" on page 19.

Next, you must set the `user.name` and `user.email` configuration variables to the appropriate values. If you have not done this yet, go to [Follow Along A-2](#). For more information about Git configuration variables, see "[Setting Git Configurations](#)" on page 17. Make sure to provide your name and email address in place of the placeholders in the Follow Along.

[FOLLOW ALONG A-2]

- 1** `$ git config --global user.name "<name>"`
- 2** `$ git config --global user.email "<email>"`

Chapter 2 Prerequisite Setup

In [Follow Along A-3](#), you will prepare the basic setup to start working from [Chapter 2](#).

[FOLLOW ALONG A-3]

1 Use your command line application to open a command line window.

2 \$ cd desktop

3 desktop \$ mkdir rainbow

4 desktop \$ cd rainbow

5 Open the `rainbow` project directory in a text editor window.

Chapter 3 Prerequisite Setup

In [Follow Along A-4](#), you will prepare the basic setup for the `rainbow` repository to start working from [Chapter 3](#).

[FOLLOW ALONG A-4]

1 Use your command line application to open a command line window.

2 \$ **cd desktop**

3 desktop \$ **mkdir rainbow**

4 desktop \$ **cd rainbow**

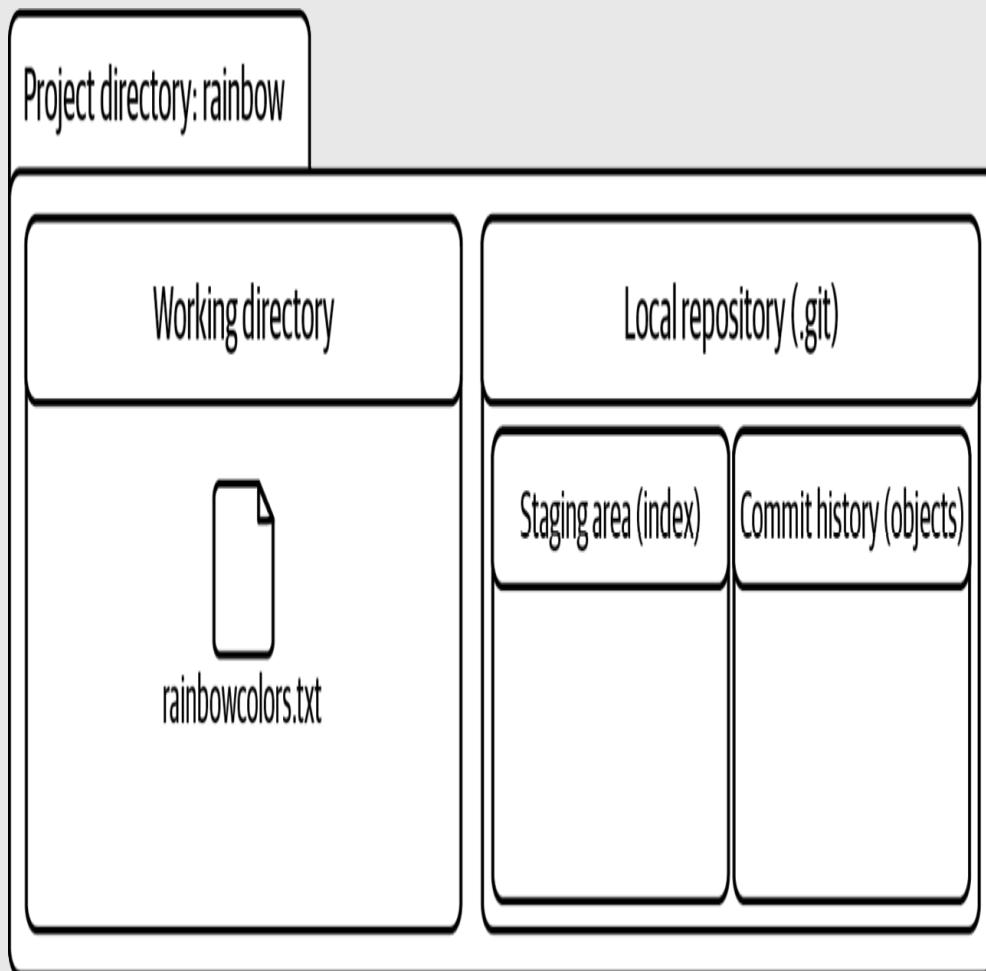
5 rainbow \$ **git init -b main**
Initialized empty Git repository in
`/Users/annaskoulikari/desktop/rainbow/.git/`

6 Open the `rainbow` project directory in a text editor window and create a file called `rainbowcolors.txt` inside this directory.

7 In the `rainbowcolors.txt` file in your text editor, add “Red is the first color of the rainbow.” on line 1 and save the file.

At the end of this process the `rainbow` repository you have created will contain the `rainbowcolors.txt` file in the working directory, as shown in [Visualize it A-1.](#)

[**VISUALIZE IT A-1**]



The re-created `rainbow` repository to start working from [Chapter 3](#), without having gone through [Chapter 2](#)

Chapter 4 Prerequisite Setup

In [Follow Along A-5](#), you will prepare the basic setup for the `rainbow` repository to start working from [Chapter 4](#).

[FOLLOW ALONG A-5]

1 Use your command line application to open a command line window.

2 \$ **cd desktop**

3 desktop \$ **mkdir rainbow**

4 desktop \$ **cd rainbow**

5 rainbow \$ **git init -b main**

Initialized empty Git repository in
/Users/annaskoulikari/desktop/rainbow/.git/

6 Open the **rainbow** project directory in a text editor window and create a file called **rainbowcolors.txt** inside this directory.

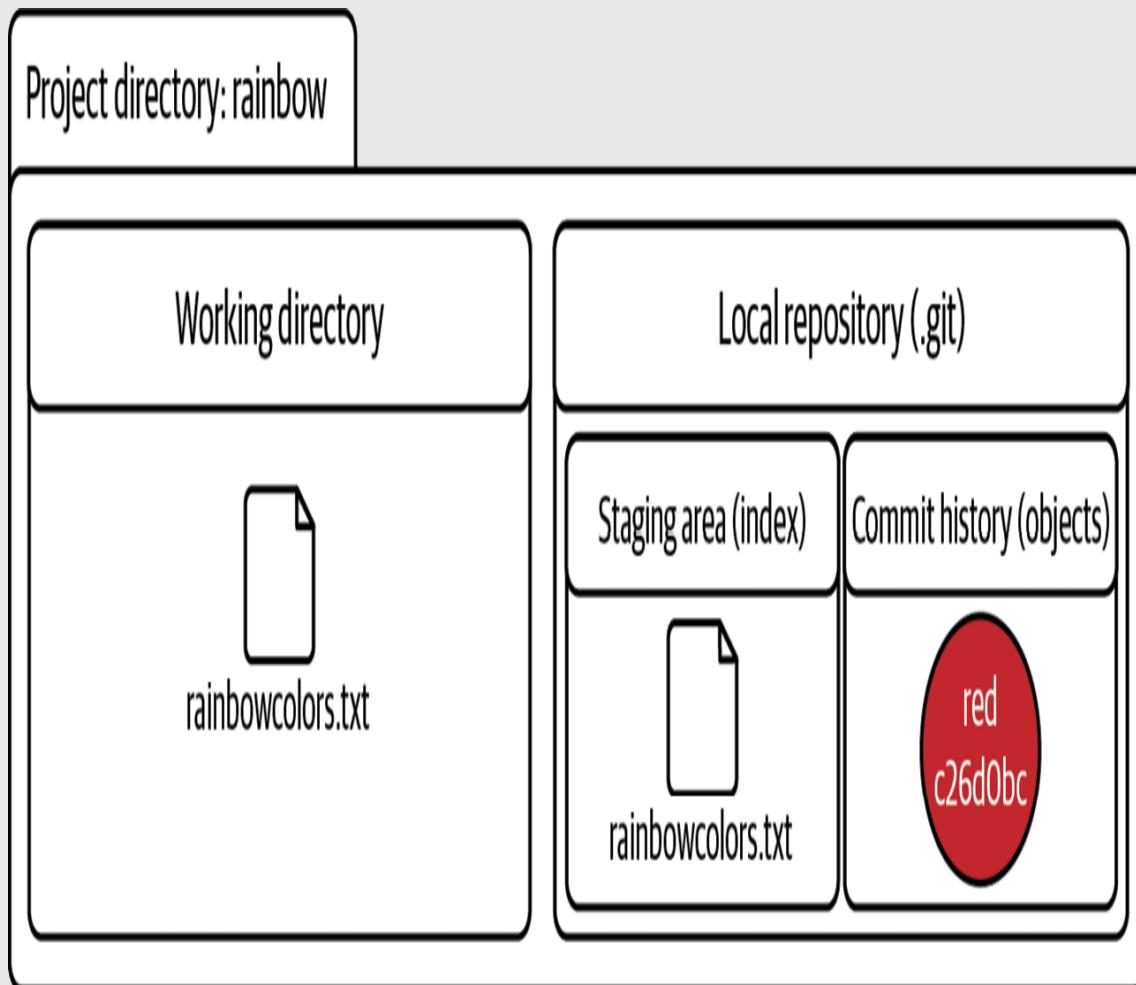
7 In the **rainbowcolors.txt** file in your text editor, add “Red is the first color of the rainbow.” on line 1 and save the file.

8 rainbow \$ **git add rainbowcolors.txt**

9 rainbow \$ **git commit -m "red"**
[main (root-commit) c26d0bc] red
1 file changed, 1 insertion(+)
create mode 100644 rainbowcolors.txt

At the end of this process the `rainbow` repository you have created will contain the red commit, as shown in [Visualize it A-2](#).

[VISUALIZE IT A-2]



The re-created `rainbow` repository to start working from [Chapter 4](#), without having gone through [Chapter 3](#)

Chapter 5 Prerequisite Setup

In [Follow Along A-6](#), you will prepare the basic setup for the `rainbow` repository to start working from [Chapter 5](#).

[FOLLOW ALONG A-6]

1 Use your command line application to open a command line window.

2 \$ **cd desktop**

3 desktop \$ **mkdir rainbow**

4 desktop \$ **cd rainbow**

5 rainbow \$ **git init -b main**

Initialized empty Git repository in
/Users/annaskoulikari/desktop/rainbow/.git/

6 Open the **rainbow** project directory in a text editor window and create a file called **rainbowcolors.txt** inside this directory.

7 In the **rainbowcolors.txt** file in your text editor, add “Red is the first color of the rainbow.” on line 1 and save the file.

8 rainbow \$ **git add rainbowcolors.txt**

9 rainbow \$ **git commit -m "red"**
[main (root-commit) c26d0bc] red
1 file changed, 1 insertion(+)
create mode 100644 rainbowcolors.txt

[FOLLOW ALONG A-6]

10 In the `rainbowcolors.txt` file in your text editor, add “Orange is the second color of the rainbow.” on line 2 and save the file.

11 `rainbow $ git add rainbowcolors.txt`

12 `rainbow $ git commit -m "orange"`

`[main 7acb333] orange`
`1 file changed, 2 insertions(+), 1 deletion(-)`

13 `rainbow $ git branch feature`

14 `rainbow $ git switch feature`
`Switched to branch 'feature'`

15 In `rainbowcolors.txt` file in your text editor, add “Yellow is the third color of the rainbow.” on line 3 and save the file.

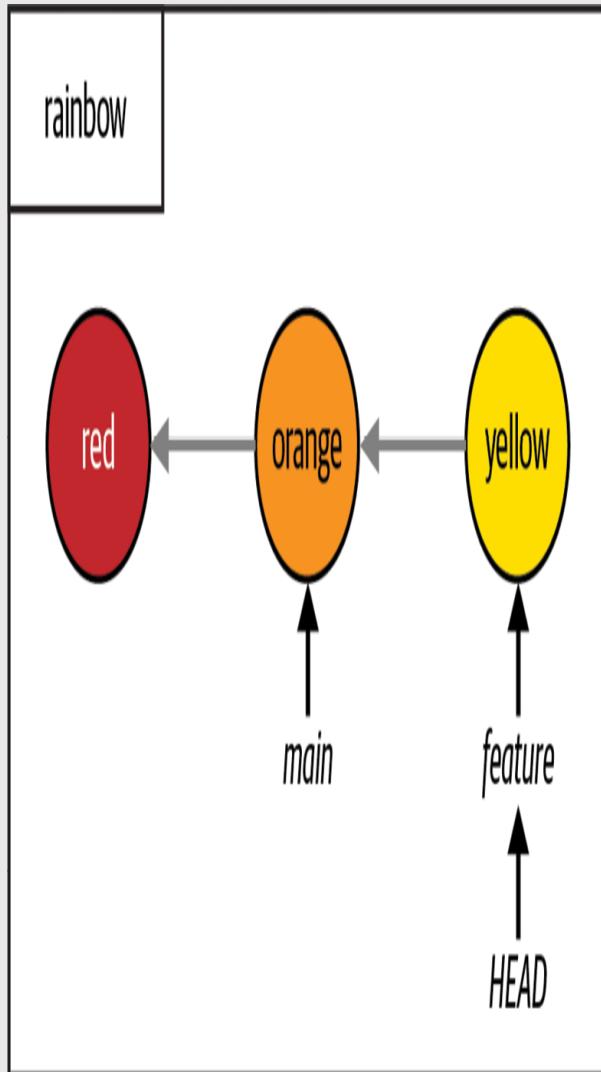
16 `rainbow $ git add rainbowcolors.txt`

17 `rainbow $ git commit -m "yellow"`

`[feature fc8139c] yellow`
`1 file changed, 2 insertions(+), 1 deletion(-)`

At the end of this process the `rainbow` repository you have created will contain the red, orange, and yellow commits, as shown in [Visualize it A-3](#).

[VISUALIZE IT A-3]



The re-created `rainbow` repository to start working from [Chapter 5](#), without having gone through [Chapter 4](#)

Chapter 6 and 7 Prerequisite Setup

In [Follow Along A-7](#), you will prepare the basic setup for the `rainbow` repository to start working from Chapters [6](#) or [7](#).

[FOLLOW ALONG A-7]

1 Use your command line application to open a command line window.

2 \$ **cd desktop**

3 desktop \$ **mkdir rainbow**

4 desktop \$ **cd rainbow**

5 rainbow \$ **git init -b main**

Initialized empty Git repository in
/Users/annaskoulikari/desktop/rainbow/.git/

6 Open the **rainbow** project directory in a text editor window and create a file called **rainbowcolors.txt** inside this directory.

7 In the **rainbowcolors.txt** file in your text editor, add “Red is the first color of the rainbow.” on line 1 and save the file.

8 rainbow \$ **git add rainbowcolors.txt**

9 rainbow \$ **git commit -m "red"**
[main (root-commit) c26d0bc] red
1 file changed, 1 insertion(+)
create mode 100644 rainbowcolors.txt

[FOLLOW ALONG A-7]

- 10** In the `rainbowcolors.txt` file in your text editor, add “Orange is the second color of the rainbow.” on line 2 and save the file.

11 `rainbow $ git add rainbowcolors.txt`

12 `rainbow $ git commit -m "orange"`

`[main 7acb333] orange`
1 file changed, 2 insertions(+), 1 deletion(-)

13 `rainbow $ git branch feature`

14 `rainbow $ git switch feature`
Switched to branch 'feature'

- 15** In the `rainbowcolors.txt` file in your text editor, add “Yellow is the third color of the rainbow.” on line 3 and save the file.

16 `rainbow $ git add rainbowcolors.txt`

17 `rainbow $ git commit -m "yellow"`

`[feature fc8139c] yellow`
1 file changed, 2 insertions(+), 1 deletion(-)

18 `rainbow $ git switch main`

`Switched to branch 'main'`

[FOLLOW ALONG A-7]

19

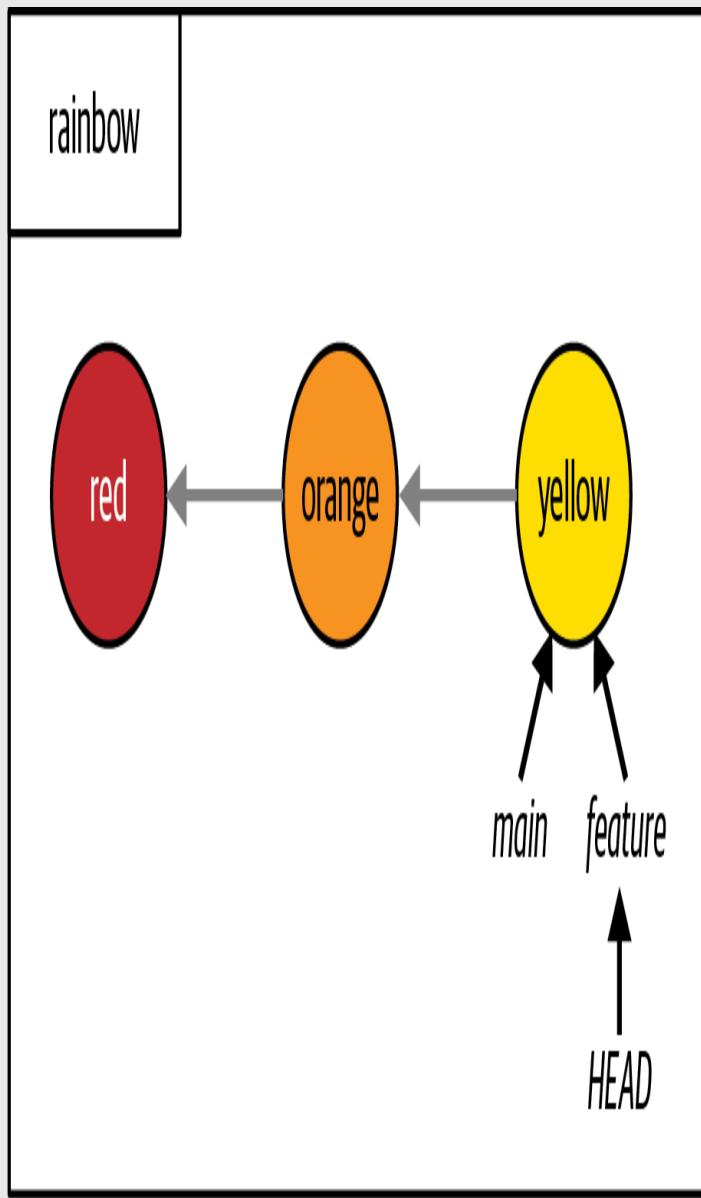
```
rainbow $ git merge feature
Updating 7acb333..fc8139c
Fast-forward
  rainbowcolors.txt | 3 +-+
  1 file changed, 2 insertions(+), 1 deletion(-)
```

[NOTE]

If you are using this section to start at [Chapter 7](#) and you don't yet have a hosting service account and/or authentication details set up for HTTPS or SSH access, then you must go to [Chapter 6](#) and carry out the exercises in that chapter before continuing. Following the instructions in this section will not be sufficient to complete the exercises in [Chapter 7](#).

At the end of this process the `rainbow` repository you have created will contain the red, orange, and yellow commits, as shown in [Visualize it A-4](#).

[VISUALIZE IT A-4]



The re-created `rainbow` repository to start working from Chapters [6](#) or [7](#), without having gone through [Chapter 5](#)

Chapter 8 Prerequisite Setup

In [Follow Along A-8](#), you will prepare the basic setup for the Rainbow project to start working from [Chapter 8](#). In step 24, be sure to enter the entire command on a single line.

[FOLLOW ALONG A-8]

1 Use your command line application to open a command line window.

2 \$ **cd desktop**

3 desktop \$ **mkdir rainbow**

4 desktop \$ **cd rainbow**

5 rainbow \$ **git init -b main**

Initialized empty Git repository in
/Users/annaskoulikari/desktop/rainbow/.git/

6 Open the **rainbow** project directory in a text editor window and create a file called **rainbowcolors.txt** inside this directory.

7 In the **rainbowcolors.txt** file in your text editor, add “Red is the first color of the rainbow.” on line 1 and save the file.

8 rainbow \$ **git add rainbowcolors.txt**

9 rainbow \$ **git commit -m "red"**
[main (root-commit) c26d0bc] red
1 file changed, 1 insertion(+)
create mode 100644 rainbowcolors.txt

[FOLLOW ALONG A-8]

- 10** In the `rainbowcolors.txt` file in your text editor, add “Orange is the second color of the rainbow.” on line 2 and save the file.

11 `rainbow $ git add rainbowcolors.txt`

12 `rainbow $ git commit -m "orange"`
[main 7acb333] orange
1 file changed, 2 insertions(+), 1 deletion(-)

13 `rainbow $ git branch feature`

14 `rainbow $ git switch feature`
Switched to branch 'feature'

- 15** In the `rainbowcolors.txt` file in your text editor, add “Yellow is the third color of the rainbow.” on line 3 and save the file.

16 `rainbow $ git add rainbowcolors.txt`

17 `rainbow $ git commit -m "yellow"`
[feature fc8139c] yellow
1 file changed, 2 insertions(+), 1 deletion(-)

18 `rainbow $ git switch main`
Switched to branch 'main'

[FOLLOW ALONG A-8]

19 rainbow \$ **git merge feature**

Updating 7acb333..fc8139c

Fast-forward

 rainbowcolors.txt | 3 +-+

 1 file changed, 2 insertions(+), 1 deletion(-)

20 If you don't yet have a hosting service account and/or authentication details set up for HTTPS or SSH access, then you must go to [Chapter 6](#) to complete this part of the process now. Return to this Follow Along once that is done.

21 Log in to your hosting service account.

22 Create a remote repository. For more information on how to do this, go to the Learning Git repository (<https://github.com/gitlearningjourney/learning-git>) or go directly to your hosting service's documentation.

When creating the repository for this exercise:

- For the repository name, use `rainbow-remote`.
- You may choose to make the repository public or private. I recommend making it private.
- Do not include any files. For example, do not include a `README` file or a `.gitignore` file.
- If you are asked to provide a default branch name, you may leave the field blank or set it to `main`.

[FOLLOW ALONG A-8]

23

Once you've finished the steps to create the remote repository, locate the remote repository URL. If you're unsure of where to find this, consult your hosting service's documentation.

There will be two versions of the URL, one for HTTPS access and one for SSH access. In the examples in this book, the two remote repository URLs are:

- HTTPS: `https://github.com/gitlearningjourney/rainbow-remote.git`
- SSH: `git@github.com:gitlearningjourney/rainbow-remote.git`

Copy the URL for the protocol you have chosen to use. In the following steps, wherever you see my URL you must use your URL instead.

24

```
rainbow $ git remote add origin https://github.com/gitlearnin  
gjourney/rainbow-remote.git
```

25

```
rainbow $ git push origin main  
  
Enumerating objects: 9, done.  
  
Counting objects: 100% (9/9), done.  
  
Delta compression using up to 4 threads  
  
Compressing objects: 100% (5/5), done.  
  
Writing objects: 100% (9/9), 747 bytes | 373.00 KiB/s, done.  
  
Total 9 (delta 1), reused 0 (delta 0), pack-reused 0  
  
remote: Resolving deltas: 100% (1/1), done.  
  
To github.com:gitlearningjourney/rainbow-remote.git  
 * [new branch]      main -> main
```

26

```
rainbow $ git switch feature  
  
Switched to branch 'feature'
```

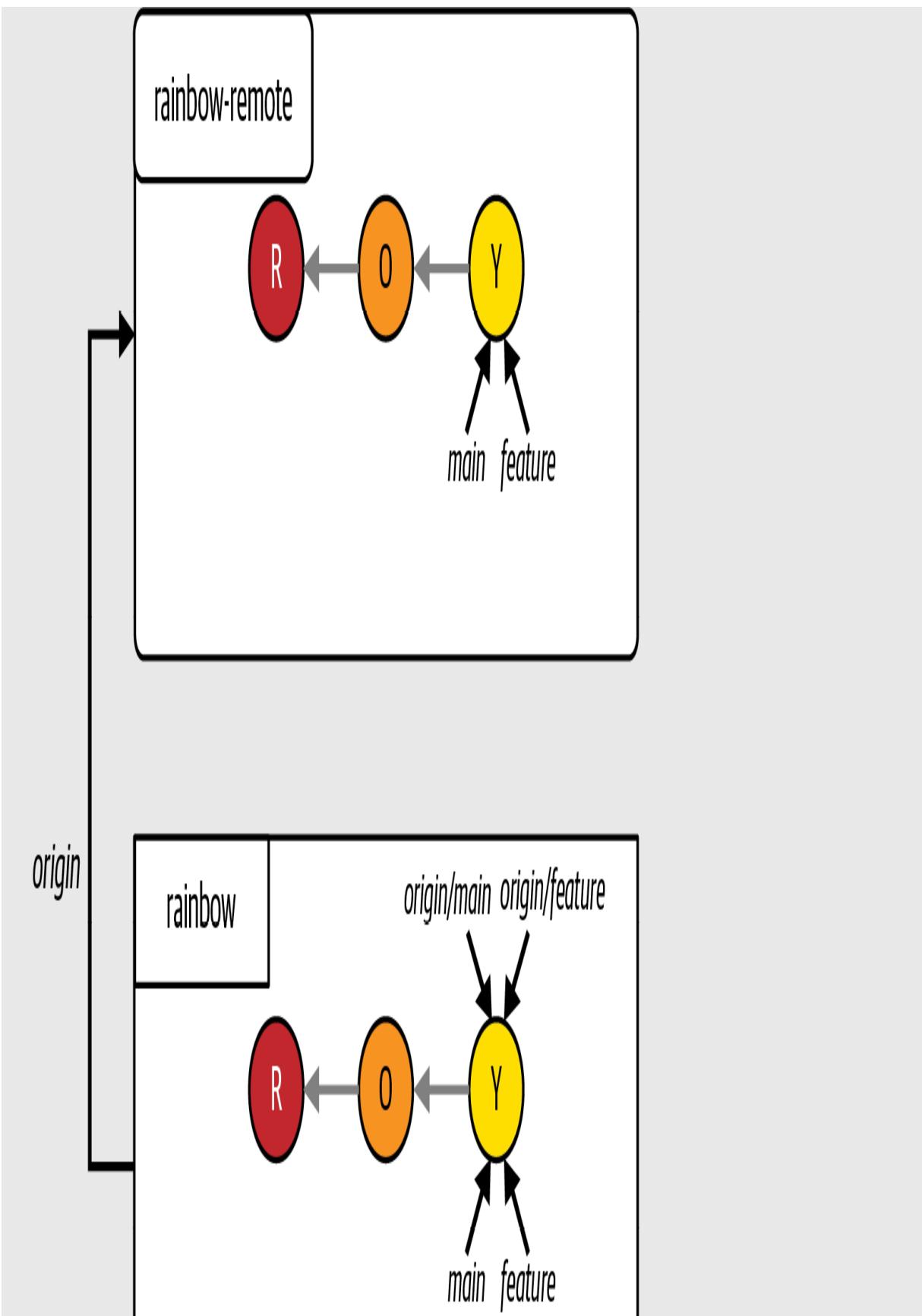
[FOLLOW ALONG A-8]

27

```
rainbow $ git push origin feature
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'feature' on GitHub by visiting:
remote:     https://github.com/gitlearningjourney/rainbow-
remote/pull/new/feature
remote:
To github.com:gitlearningjourney/rainbow-remote.git
 * [new branch]      feature -> feature
```

At the end of this process the Rainbow project you have created will contain the red, orange, and yellow commits in all the repositories, as shown in [Visualize it A-5](#).

[VISUALIZE IT A-5]





The re-created Rainbow project to start working from [Chapter 8](#), without having gone through [Chapter 7](#)

Chapter 9 Prerequisite Setup

In [Follow Along A-9](#), you will prepare the basic setup for the Rainbow project to start working from [Chapter 9](#). In steps 14 and 17, be sure to enter the entire command on a single line.

[FOLLOW ALONG A-9]

1 Use your command line application to open a command line window.

2 \$ **cd desktop**

3 desktop \$ **mkdir rainbow**

4 desktop \$ **cd rainbow**

5 rainbow \$ **git init -b main**

Initialized empty Git repository in
/Users/annaskoulikari/desktop/rainbow/.git/

6 Open the **rainbow** project directory in a text editor window and create a file called **rainbowcolors.txt** inside this directory.

7 Add the following text to the **rainbowcolors.txt** file in your text editor, and save the file:

Red is the first color of the rainbow.
Orange is the second color of the rainbow.
Yellow is the third color of the rainbow.
Green is the fourth color of the rainbow.

8 rainbow \$ **git add rainbowcolors.txt**

[FOLLOW ALONG A-9]

9 rainbow \$ **git commit -m "green"**
[main (root-commit) 4e59074] "green"
1 file changed, 4 insertions(+)
create mode 100644 rainbowcolors.txt

10 If you don't yet have a hosting service account and/or authentication details set up for HTTPS or SSH access, then you must go to [Chapter 6](#) to complete this part of the process now. Return to this Follow Along once that is done.

11 Log in to your hosting service account.

12 Create a remote repository. For more information on how to do this, go to the Learning Git repository (<https://github.com/gitlearningjourney/learning-git>) or go directly to your hosting service's documentation.

When creating the repository for this exercise:

- For the repository name, use `rainbow-remote`.
- You may choose to make the repository public or private. I recommend making it private.
- Do not include any files. For example, do not include a `README` file or a `.gitignore` file.
- If you are asked to provide a default branch name, you may leave the field blank or set it to `main`.

[FOLLOW ALONG A-9]

13

Once you've finished the steps to create the remote repository, locate the remote repository URL. If you're unsure of where to find this, consult your hosting service's documentation.

There will be two versions of the URL, one for HTTPS access and one for SSH access. In the examples in this book, the two remote repository URLs are:

- HTTPS: `https://github.com/gitlearningjourney/rainbow-remote.git`
- SSH: `git@github.com:gitlearningjourney/rainbow-remote.git`

Copy the URL for the protocol you have chosen to use. In the following steps, wherever you see my URL you must use your URL instead.

14

```
rainbow $ git remote add origin https://github.com/gitlearnin  
gjourney/rainbow-remote.git
```

15

```
rainbow $ git push origin main  
  
Enumerating objects: 3, done.  
  
Counting objects: 100% (3/3), done.  
  
Delta compression using up to 4 threads  
  
Compressing objects: 100% (2/2), done.  
  
Writing objects: 100% (3/3), 311 bytes | 311.00 KiB/s, done.  
  
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0  
  
To github.com:gitlearningjourney/rainbow-remote.git  
 * [new branch]      main -> main
```

16

Open a new command line window to navigate to the `desktop` directory to simulate that you are your friend now.

[FOLLOW ALONG A-9]

17

```
desktop $ git clone https://github.com/gitlearningjourney/rainbo  
w-remote.git friend-rainbow  
  
Cloning into 'friend-rainbow'...  
  
remote: Enumerating objects: 3, done.  
  
remote: Counting objects: 100% (3/3), done.  
  
remote: Compressing objects: 100% (2/2), done.  
  
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0  
  
Receiving objects: 100% (3/3), done.
```

18

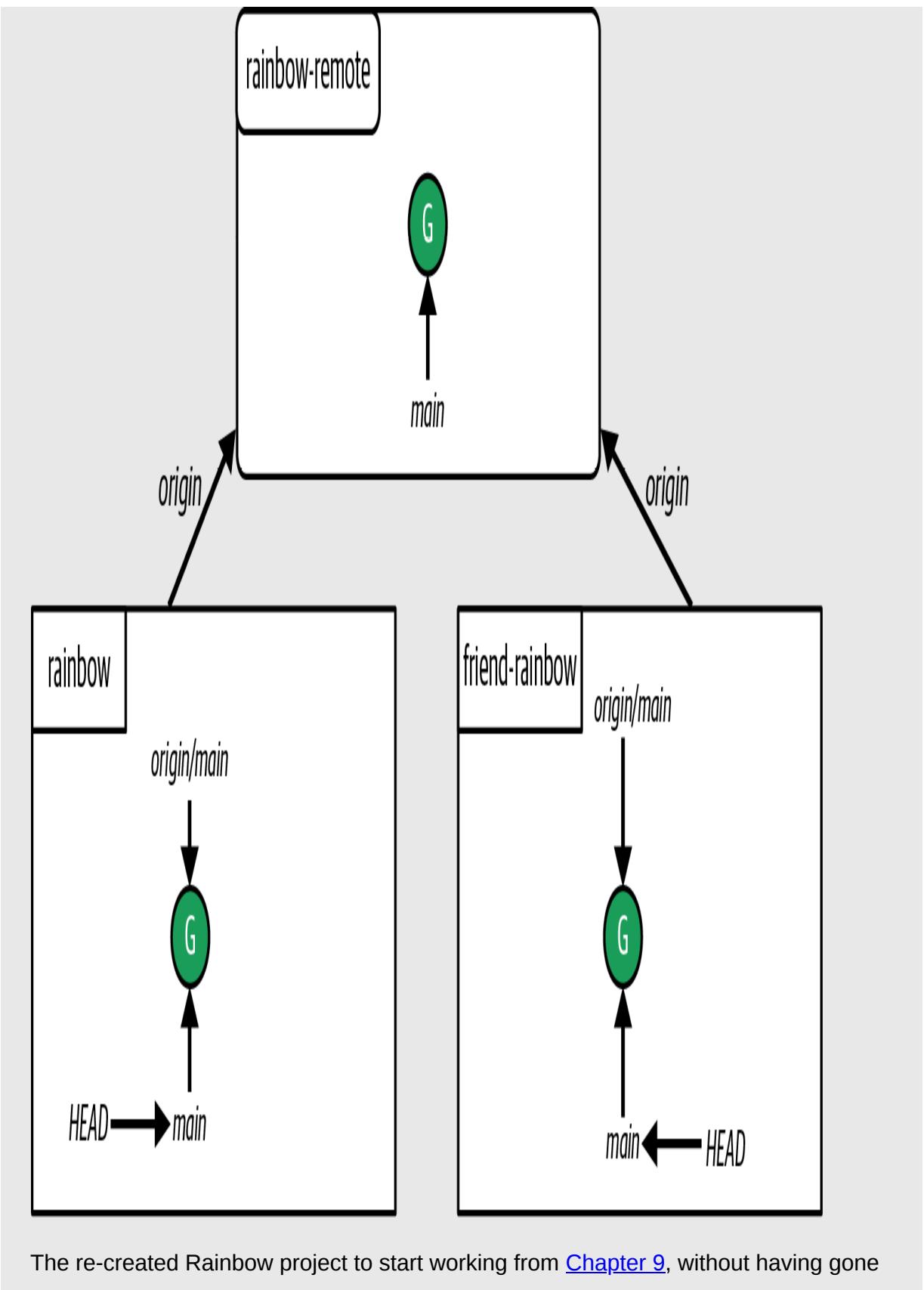
```
desktop $ cd friend-rainbow
```

19

Open the `friend-rainbow` project directory in a new text editor window.

At the end of this process the minimum setup for the Rainbow project you have created will contain just one commit, the green commit, in all the repositories, as shown in [Visualize it A-6](#).

[VISUALIZE IT A-6]



through [Chapter 8](#)

Chapter 10 Prerequisite Setup

In [Follow Along A-10](#), you will prepare the basic setup for the Rainbow project to start working from [Chapter 10](#). In steps 15 and 19, be sure to enter the entire command on a single line.

[FOLLOW ALONG A-10]

1 Use your command line application to open a command line window.

2 \$ **cd desktop**

3 desktop \$ **mkdir rainbow**

4 desktop \$ **cd rainbow**

5 rainbow \$ **git init -b main**

Initialized empty Git repository in
/Users/annaskoulikari/desktop/rainbow/.git/

6 Open the **rainbow** project directory in a text editor window and create a file called **rainbowcolors.txt** and a file called **othercolors.txt** inside this directory.

7 Add the following text to the **rainbowcolors.txt** file in your text editor and save the file:

Red is the first color of the rainbow.
Orange is the second color of the rainbow.
Yellow is the third color of the rainbow.
Green is the fourth color of the rainbow.
Blue is the fifth color of the rainbow.

8 Add the following text to the **othercolors.txt** file in your text editor and save the file:

Brown is not a color in the rainbow.

[FOLLOW ALONG A-10]

9 rainbow \$ **git add rainbowcolors.txt othercolors.txt**

10 rainbow \$ **git commit -m "fake merge commit 1"**

2 files changed, 6 insertions(+)
create mode 100644 othercolors.txt
create mode 100644 rainbowcolors.txt

11 If you don't yet have a hosting service account and/or authentication details set up for HTTPS or SSH access, then you must go to [Chapter 6](#) to complete this part of the process now. Return to this Follow Along once that is done.

12 Log in to your hosting service account.

13 Create a remote repository. For more information on how to do this, go to the Learning Git repository (<https://github.com/gitlearningjourney/learning-git>) or go directly to your hosting service's documentation.

When creating the repository for this exercise:

- For the repository name, use `rainbow-remote`.
- You may choose to make the repository public or private. I recommend making it private.
- Do not include any files. For example, do not include a `README` file or a `.gitignore` file.
- If you are asked to provide a default branch name, you may leave the field blank or set it to `main`.

[FOLLOW ALONG A-10]

14

Once you've finished the steps to create the remote repository, locate the remote repository URL. If you're unsure of where to find this, consult your hosting service's documentation.

There will be two versions of the URL, one for HTTPS access and one for SSH access. In the examples in this book, the two remote repository URLs are:

- HTTPS: <https://github.com/gitlearningjourney/rainbow-remote.git>
- SSH: <git@github.com:gitlearningjourney/rainbow-remote.git>

Copy the URL for the protocol you have chosen to use. In the following steps, wherever you see my URL you must use your URL instead.

15

```
rainbow $ git remote add origin https://github.com/gitlearnin  
gjourney/rainbow-remote.git
```

16

```
rainbow $ git push origin main  
  
Enumerating objects: 4, done.  
  
Counting objects: 100% (4/4), done.  
  
Delta compression using up to 4 threads  
  
Compressing objects: 100% (3/3), done.  
  
Writing objects: 100% (4/4), 394 bytes | 394.00 KiB/s, done.  
  
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0  
  
To github.com:gitlearningjourney/rainbow-remote.git  
 * [new branch]      main -> main
```

17

```
rainbow $ git branch -u origin/main  
  
branch 'main' set up to track 'origin/main'.
```

18

Open a new command line window to navigate to the [desktop](#) directory to simulate that you are your friend now.

[FOLLOW ALONG A-10]

19

```
desktop $ git clone https://github.com/gitlearningjourney/rainbow-remote.git friend-rainbow
Cloning into 'friend-rainbow'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 4 (delta 0), pack-reused 0
Receiving objects: 100% (4/4), done.
```

20

```
desktop $ cd friend-rainbow
```

21

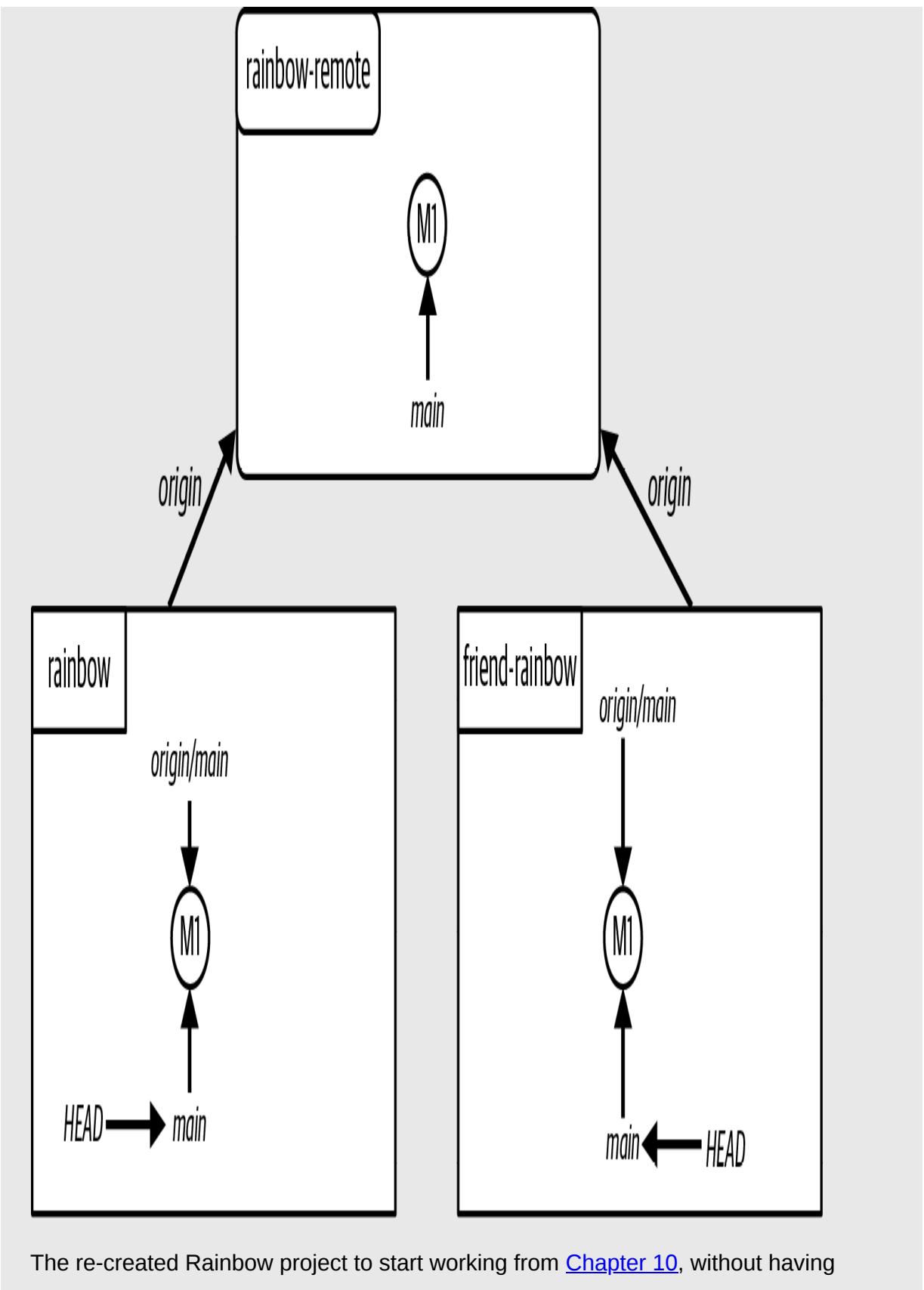
Open the `friend-rainbow` project directory in a new text editor window.

At the end of this process the minimum setup for the Rainbow project you have created will contain just one commit, the fake merge commit 1 (M1), in all the repositories, as shown in [Visualize it A-7](#).

[NOTE]

The last commit created in [Chapter 9](#) was merge commit 1 (M1), which has two parents. Your commit will be the first and only one in the repositories, so it won't have any parent commits. Therefore, it is a fake merge commit.

[VISUALIZE IT A-7]



gone through [Chapter 9](#)

Chapter 11 Prerequisite Setup

In [Follow Along A-11](#), you will prepare the basic setup for the Rainbow project to start working from [Chapter 11](#). In steps 15 and 19, be sure to enter the entire command on a single line.

[FOLLOW ALONG A-11]

1 Use your command line application to open a command line window.

2 \$ **cd desktop**

3 desktop \$ **mkdir rainbow**

4 desktop \$ **cd rainbow**

5 rainbow \$ **git init -b main**

Initialized empty Git repository in
/Users/annaskoulikari/desktop/rainbow/.git/

6 Open the **rainbow** project directory in a text editor window and create a file called **rainbowcolors.txt** and a file called **othercolors.txt** inside this directory.

7 Add the following text to the **rainbowcolors.txt** file in your text editor and save the file:

Red is the first color of the rainbow.
Orange is the second color of the rainbow.
Yellow is the third color of the rainbow.
Green is the fourth color of the rainbow.
Blue is the fifth color of the rainbow.
Indigo is the sixth color of the rainbow.
Violet is the seventh color of the rainbow.

8 Add the following text to the **othercolors.txt** file in your text editor and save the file:

Brown is not a color in the rainbow.

[FOLLOW ALONG A-11]

9 rainbow \$ **git add rainbowcolors.txt othercolors.txt**

10 rainbow \$ **git commit -m "fake merge commit 2"**

```
[main (root-commit) 32fa0b7] fake merge commit 2
 2 files changed, 8 insertions(+)
  create mode 100644 othercolors.txt
  create mode 100644 rainbowcolors.txt
```

11 If you don't yet have a hosting service account and/or authentication details set up for HTTPS or SSH access, then you must go to [Chapter 6](#) to complete this part of the process now. Return to this Follow Along once that is done.

12 Log in to your hosting service account.

13 Create a remote repository. For more information on how to do this, go to the Learning Git repository (<https://github.com/gitlearningjourney/learning-git>) or go directly to your hosting service's documentation.

When creating the repository for this exercise:

- For the repository name, use `rainbow-remote`.
- You may choose to make the repository public or private. I recommend making it private.
- Do not include any files. For example, do not include a `README` file or a `.gitignore` file.
- If you are asked to provide a default branch name, you may leave the field blank or set it to `main`.

[FOLLOW ALONG A-11]

14

Once you've finished the steps to create the remote repository, locate the remote repository URL. If you're unsure of where to find this, consult your hosting service's documentation.

There will be two versions of the URL, one for HTTPS access and one for SSH access. In the examples in this book, the two remote repository URLs are:

- HTTPS: <https://github.com/gitlearningjourney/rainbow-remote.git>
- SSH: <git@github.com:gitlearningjourney/rainbow-remote.git>

Copy the URL for the protocol you have chosen to use. In the following steps, wherever you see my URL you must use your URL instead.

15

```
rainbow $ git remote add origin https://github.com/gitlearningjourney/rainbow-remote.git
```

16

```
rainbow $ git push origin main
Enumerating objects: 4, done.

Counting objects: 100% (4/4), done.

Delta compression using up to 4 threads

Compressing objects: 100% (3/3), done.

Writing objects: 100% (4/4), 413 bytes | 413.00 KiB/s, done.

Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:gitlearningjourney/rainbow-remote.git
 * [new branch]      main -> main
```

17

```
rainbow $ git branch -u origin/main
branch 'main' set up to track 'origin/main'.
```

18

Open a new command line window to navigate to the [desktop](#) directory to simulate that you are your friend now.

[FOLLOW ALONG A-11]

19

```
desktop $ git clone https://github.com/gitlearningjourney/rainbo
w-remote.git friend-rainbow
Cloning into 'friend-rainbow'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 4 (delta 0), pack-reused 0
Receiving objects: 100% (4/4), done.
```

20

```
desktop $ cd friend-rainbow
```

21

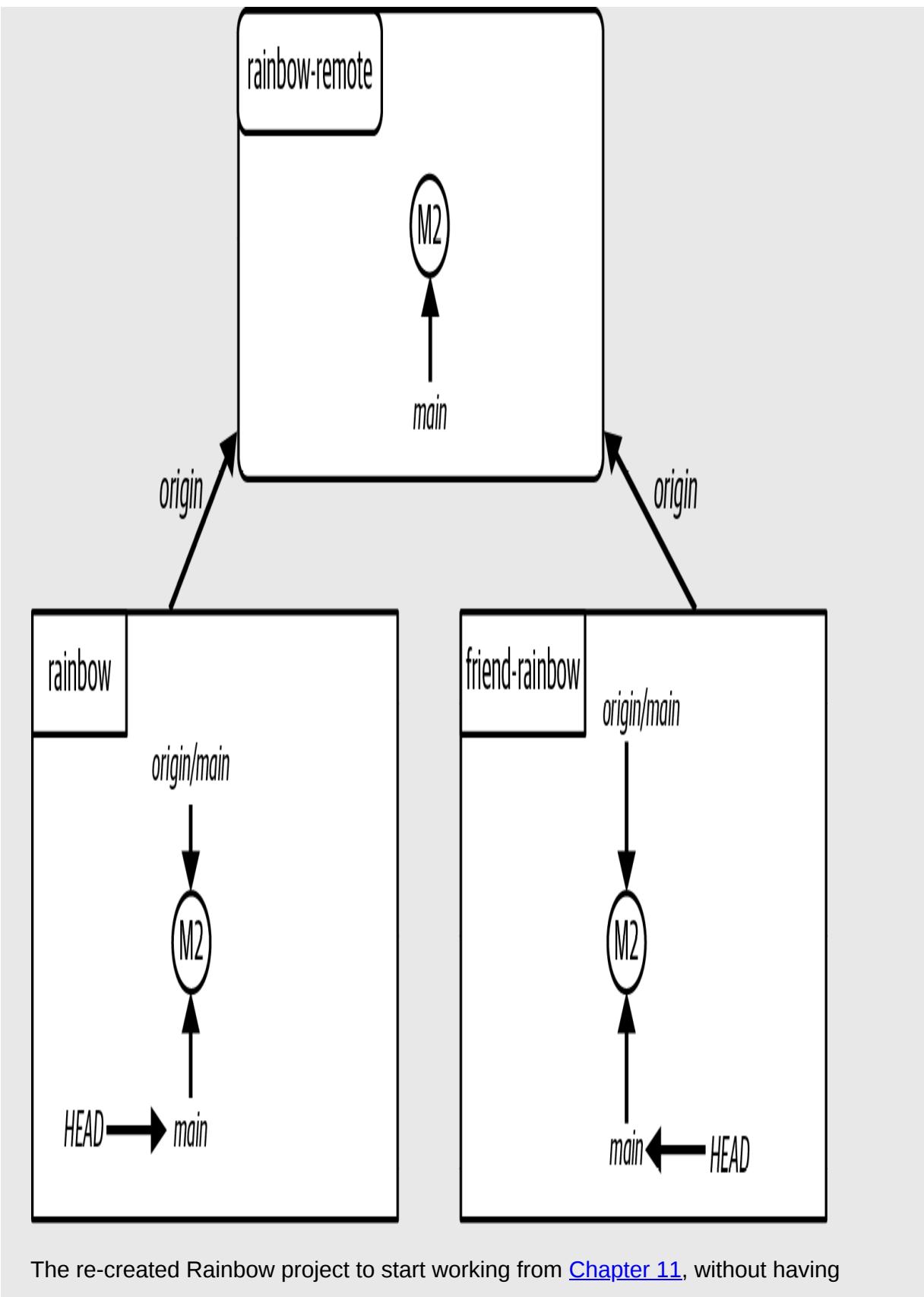
Open the `friend-rainbow` project directory in a new text editor window.

At the end of this process the minimum setup for the Rainbow project you have created will contain just one commit, the fake merge commit 2 (M2), in all the repositories, as shown in [Visualize it A-8](#).

[NOTE]

The last commit created in [Chapter 10](#) was merge commit 2 (M2), which has two parents. Your commit will be the first and only one in the repositories, so it won't have any parent commits. Therefore, it is a fake merge commit.

[VISUALIZE IT A-8]



gone through [Chapter 10](#)

Chapter 12 Prerequisite Setup

In [Follow Along A-12](#), you will prepare the basic setup for the Rainbow project to start working from [Chapter 12](#). In steps 15 and 19, be sure to enter the entire command on a single line.

[FOLLOW ALONG A-12]

1 Use your command line application to open a command line window.

2 \$ **cd desktop**

3 desktop \$ **mkdir rainbow**

4 desktop \$ **cd rainbow**

5 rainbow \$ **git init -b main**

Initialized empty Git repository in
/Users/annaskoulikari/desktop/rainbow/.git/

6 Open the **rainbow** project directory in a text editor window and create a file called **rainbowcolors.txt** and a file called **othercolors.txt** inside this directory.

7 Add the following text to the **rainbowcolors.txt** file in your text editor and save the file:

Red is the first color of the rainbow.

Orange is the second color of the rainbow.

Yellow is the third color of the rainbow.

Green is the fourth color of the rainbow.

Blue is the fifth color of the rainbow.

Indigo is the sixth color of the rainbow.

Violet is the seventh color of the rainbow.

These are the colors of the rainbow.

[FOLLOW ALONG A-12]

- 8** Add the following text to the `othercolors.txt` file in your text editor and save the file:

Brown is not a color in the rainbow.

Gray is not a color in the rainbow.

Black is not a color in the rainbow.

- 9** `rainbow $ git add rainbowcolors.txt othercolors.txt`

- 10** `rainbow $ git commit -m "rainbow"`
[main (root-commit) 56b92dc] "rainbow"
2 files changed, 11 insertions(+)
create mode 100644 othercolors.txt
create mode 100644 rainbowcolors.txt

- 11** If you don't yet have a hosting service account and/or authentication details set up for HTTPS or SSH access, then you must go to [Chapter 6](#) to complete this part of the process now. Return to this Follow Along once that is done.

- 12** Log in to your hosting service account.

[FOLLOW ALONG A-12]

13

Create a remote repository. For more information on how to do this, go to the Learning Git repository (<https://github.com/gitlearningjourney/learning-git>) or go directly to your hosting service's documentation.

When creating the repository for this exercise:

- For the repository name, use `rainbow-remote`.
- You may choose to make the repository public or private. I recommend making it private.
- Do not include any files. For example, do not include a `README` file or a `.gitignore` file.
- If you are asked to provide a default branch name, you may leave the field blank or set it to `main`.

14

Once you've finished the steps to create the remote repository, locate the remote repository URL. If you're unsure of where to find this, consult your hosting service's documentation.

There will be two versions of the URL, one for HTTPS access and one for SSH access. In the examples in this book, the two remote repository URLs are:

- HTTPS: <https://github.com/gitlearningjourney/rainbow-remote.git>
- SSH: `git@github.com:gitlearningjourney/rainbow-remote.git`

Copy the URL for the protocol you have chosen to use. In the following steps, wherever you see my URL you must use your URL instead.

15

```
rainbow $ git remote add origin https://github.com/gitlearnin  
gjourney/rainbow-remote.git
```

[FOLLOW ALONG A-12]

16

```
rainbow $ git push origin main
Enumerating objects: 4, done.

Counting objects: 100% (4/4), done.

Delta compression using up to 4 threads

Compressing objects: 100% (4/4), done.

Writing objects: 100% (4/4), 445 bytes | 445.00 KiB/s, done.

Total 4 (delta 0), reused 0 (delta 0), pack-reused 0

To github.com:gitlearningjourney/rainbow-remote.git
 * [new branch]      main -> main
```

17

```
rainbow $ git branch -u origin/main
branch 'main' set up to track 'origin/main'.
```

18

Open a new command line window to navigate to the `desktop` directory to simulate that you are your friend now.

19

```
desktop $ git clone https://github.com/gitlearningjourney/rai
nbow-remote.git friend-rainbow
Cloning into 'friend-rainbow'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 4 (delta 0), pack-reused 0
Receiving objects: 100% (4/4), done.
```

20

```
desktop $ cd friend-rainbow
```

21

Open the `friend-rainbow` project directory in a new text editor window.

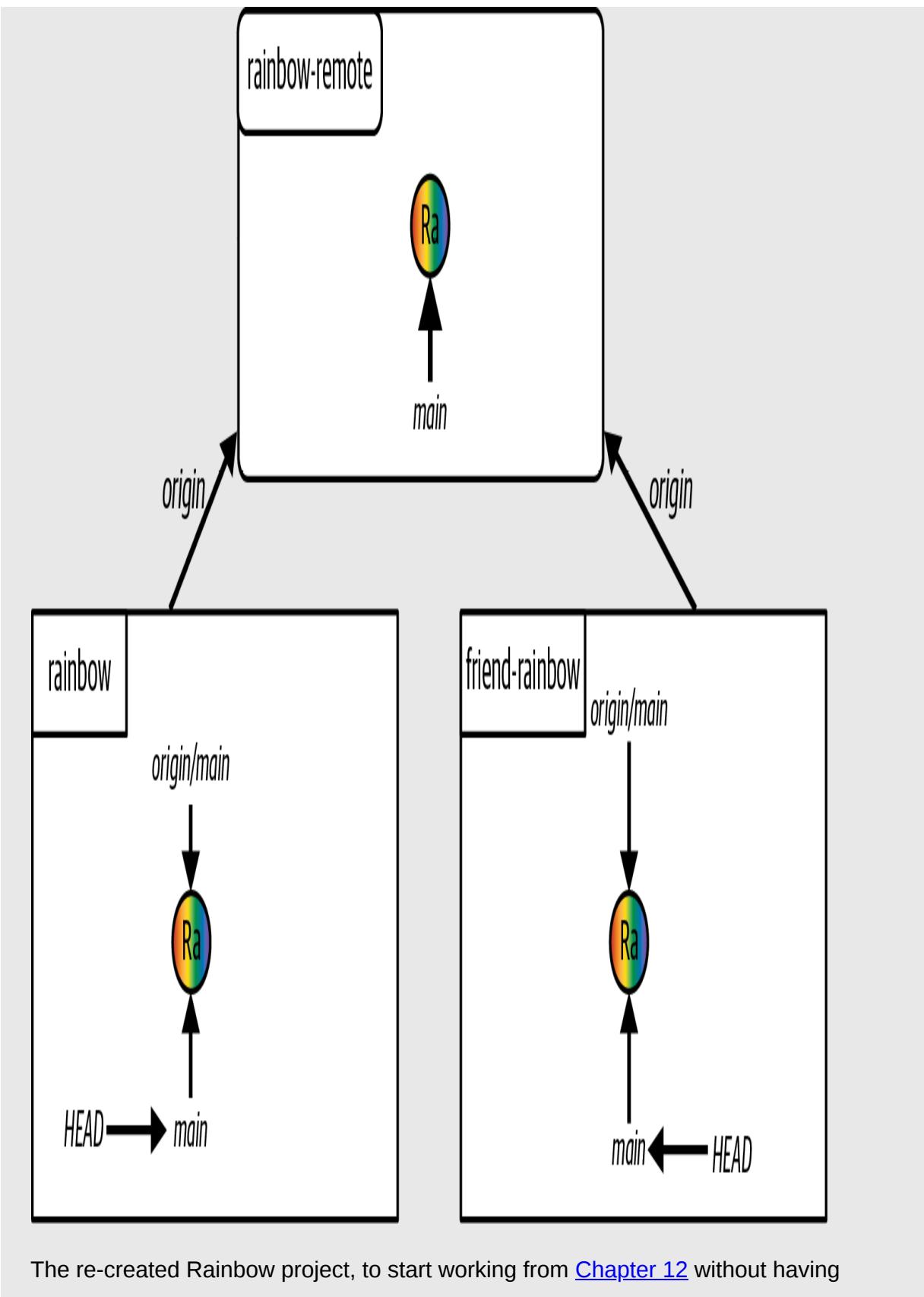
At the end of this process the minimum setup for the Rainbow project you have created will contain just one commit, the rainbow commit, in all the repositories, as shown in [Visualize it A-9](#).

[NOTE]

The rainbow commit at the start of [Chapter 12](#) is the rebased rainbow commit, which is why it is labeled Ra' in the Visualize It diagrams in that chapter.

Your rainbow commit at the end of this process is a regular commit. Keep this in mind as you work through the chapter. See [Appendix C](#) for a summary of the visual language of the book, if you would like a refresher.

[VISUALIZE IT A-9]



gone through [Chapter 11](#)

[Appendix B]

Command Quick Reference

CHAPTER 1	
clear	Clear the command line screen
pwd	Show the path of the current working directory
ls	List visible files and directories
ls -a	List hidden and visible files and directories
cd <path_to_directory>	Change directory
mkdir <directory_name>	Make a directory
git config --global --list	List the variables in the global Git configuration file and their values
git config --global user.name "<name>"	Set your name in the global Git configuration file
git config --global user.email "<email>"	Set your email address in the global Git configuration file

CHAPTER 2	
git init	Initialize a Git repository
git init -b <branch_name>	Initialize a Git repository and set the name for the initial branch to be <branch_name>

CHAPTER 3	
git status	Show the state of the working directory and the staging area
git add <filename>	Add one file to the staging area

CHAPTER 3

git add <filename> <filename> ...	Add multiple files to the staging area
git add -A	Add all the files in the working directory that have been edited or changed to the staging area
git commit -m " <message>"	Create a new commit with a commit message
git log	Show a list of commits in reverse chronological order

CHAPTER 4

git branch	List local branches
git branch <new_branch_name>	Create a branch
git switch <branch_name>	Switch branches
git checkout <branch_name>	Switch branches

CHAPTER 5

git merge <branch_name>	Integrate changes from one branch into another branch
git log --all	Show a list of commits in reverse chronological order for all branches in a local repository
git checkout <commit_hash>	Check out a commit
git switch -c <new_branch_name>	Create a new branch and switch onto it
git checkout -b <new_branch_name>	Create a new branch and switch onto it

CHAPTER 7

git push	Upload data to a remote repository
----------	------------------------------------

CHAPTER 7

git remote add <shortname> <URL>	Add a connection to a remote repository named <shortname> at <URL>
git remote	List the remote repository connections stored in the local repository by shortname
git remote -v	List the remote repository connections in the local repository with shortnames and URLs
git push <shortname> <branch_name>	Upload content from <branch_name> to the <shortname> remote repository
git branch --all	List local branches and remote-tracking branches

CHAPTER 8

git clone <URL> <directory_name>	Clone a remote repository
git push <shortname> -d <branch_name>	Delete a remote branch and the associated remote-tracking branch
git branch -d <branch_name>	Delete a local branch
git branch -vv	List the local branches and their upstream branches, if they have any
git fetch <shortname>	Download data from the <shortname> remote repository
git fetch	Download data from the remote repository with shortname origin
git fetch -p	Remove remote-tracking branches that correspond to deleted remote branches and download data from the remote repository

CHAPTER 9

git branch -u <shortname>/<branch_name>	Define an upstream branch for the current local branch
--	--

CHAPTER 9

git pull <shortname> <branch_name>	Fetch and integrate changes from the <shortname> remote repository for the specified <branch_name>
git pull	If an upstream branch is defined for the current branch, fetch and integrate changes from the defined upstream branch

CHAPTER 10

git merge --abort	Stop the merge process and go back to the state before the merge
-------------------	--

CHAPTER 11

git rebase <branch_name>	Reapply commits on top of another branch
git restore --staged <filename>	Restore a file to another version of the file in the staging area
git rebase --continue	Continue with the rebase process after having resolved merge conflicts
git rebase --abort	Stop the rebase process and go back to the state before the rebase

[Appendix C]

Visual Language Reference

Commits

In the diagrams in this book, commits are represented as circles. Regular commits in the Rainbow project are one solid color and contain the full name or an abbreviation of the name (the only exception is the rainbow commit, which contains all the colors of the rainbow). See [Figure C-1](#) for an example.

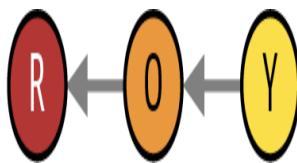


FIGURE C-1

Example of regular Rainbow project commits using color and commit name abbreviations to distinguish them from one another

If a regular commit is re-created because of a rebase operation (discussed in [Chapter 11](#)), then we add an apostrophe to the name of the re-created commit to distinguish it from the original commit. See [Figure C-2](#) for an example.

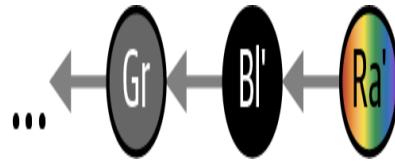


FIGURE C-2

Example of rebased commits in the Rainbow project, with apostrophes in the abbreviations to distinguish them from the original commits

Throughout the book, merge commits are represented by a white circle with a thick black border and the letter M (sometimes they may also include a number). An example of this can be seen in [Figure C-3](#).



FIGURE C-3

Example of a merge commit

Regular commits in the Book project are represented by blue circles, with letters of the alphabet used to distinguish one from another. See [Figure C-4](#) for an example.

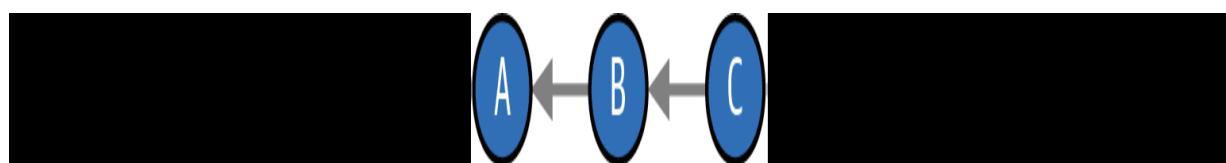


FIGURE C-4

Example of regular Book project commits using letters of the alphabet to distinguish them

from one another

Git Diagram

The Git Diagram, introduced in [Chapter 2](#), represents one Git project directory. It consists of the four areas of Git: the working directory, the staging area, the commit history, and the local repository. The working directory and staging area may contain files, and the commit history may contain commits and branches. Branches and the `HEAD` pointer (reference) are represented by black arrows, and parent links between commits are represented by gray arrows. An example of a Git Diagram is shown in [Figure C-5](#).

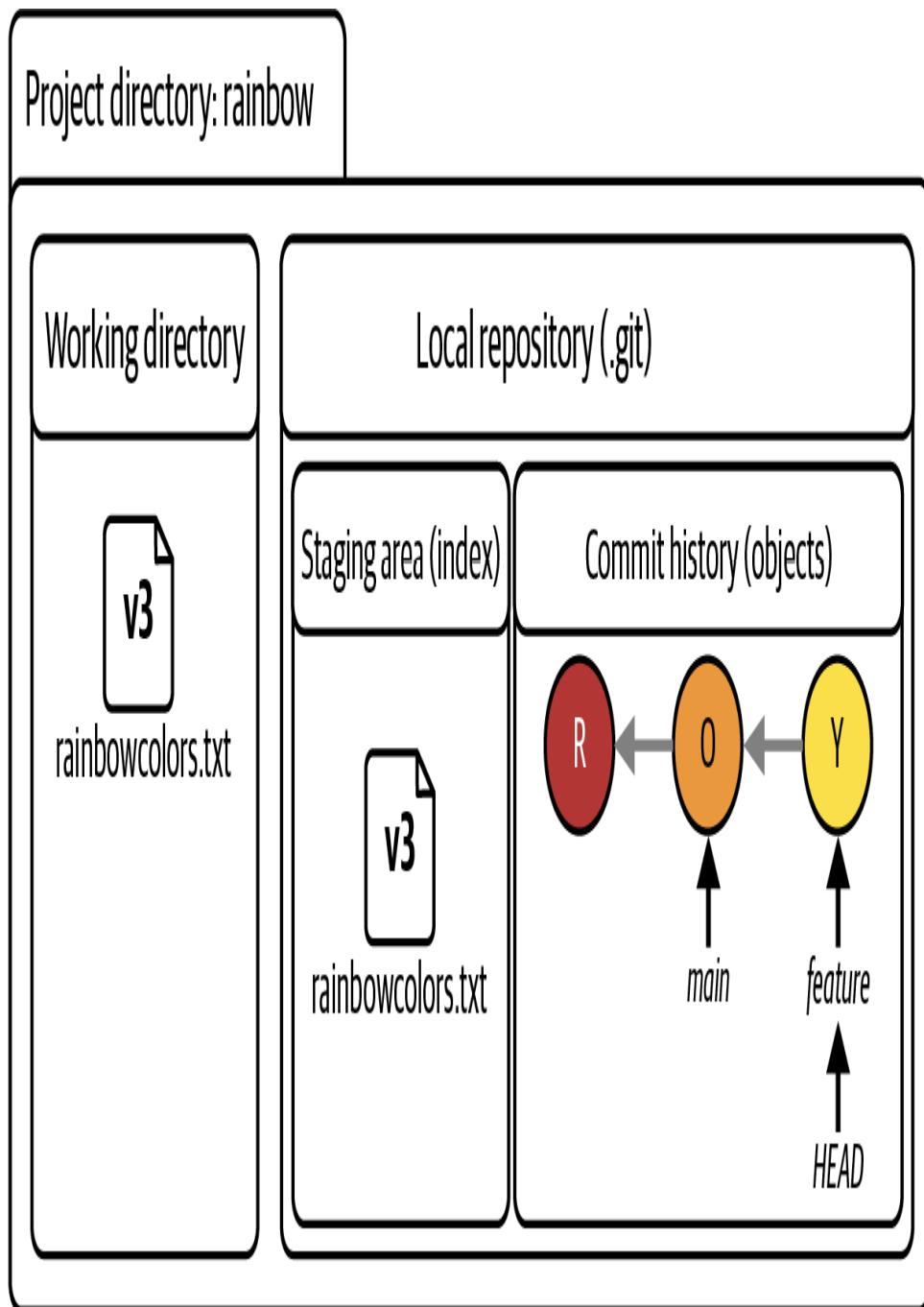


FIGURE C-5

An example of a Git Diagram with representations of the working directory, staging area, commit history, and local repository, showing the state of the Rainbow project at the start of

[Chapter 5](#)

Repository Diagram

The Repository Diagram represents either one repository or multiple repositories. Local repositories are always represented by normal rectangles. Remote repositories are represented by rectangles with rounded corners. The repositories may contain commits and branches. The Repository Diagram is built up from [Chapter 4](#) onward.

In [Figure C-6](#), you can see an example of a Repository Diagram with one local repository.

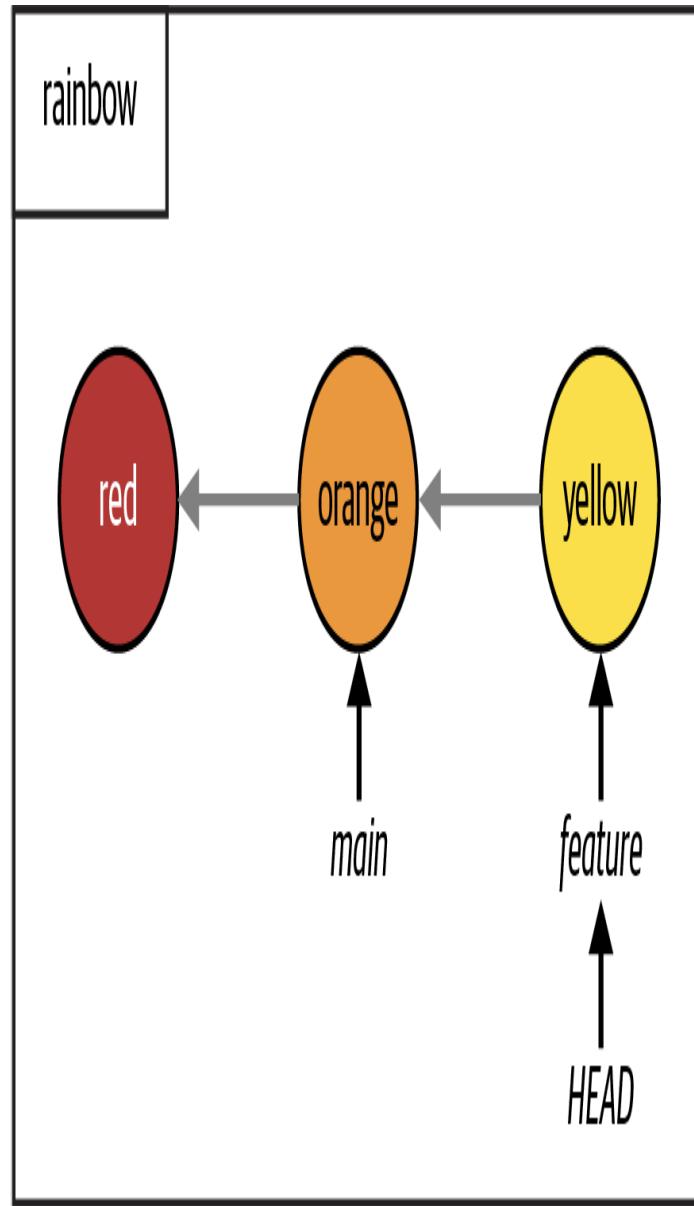


FIGURE C-6

An example of a Repository Diagram with one local repository, representing the state of the Rainbow project at the end of [Chapter 4](#)

In [Figure C-7](#), you can see an example of a Repository Diagram with two local repositories and one remote repository.

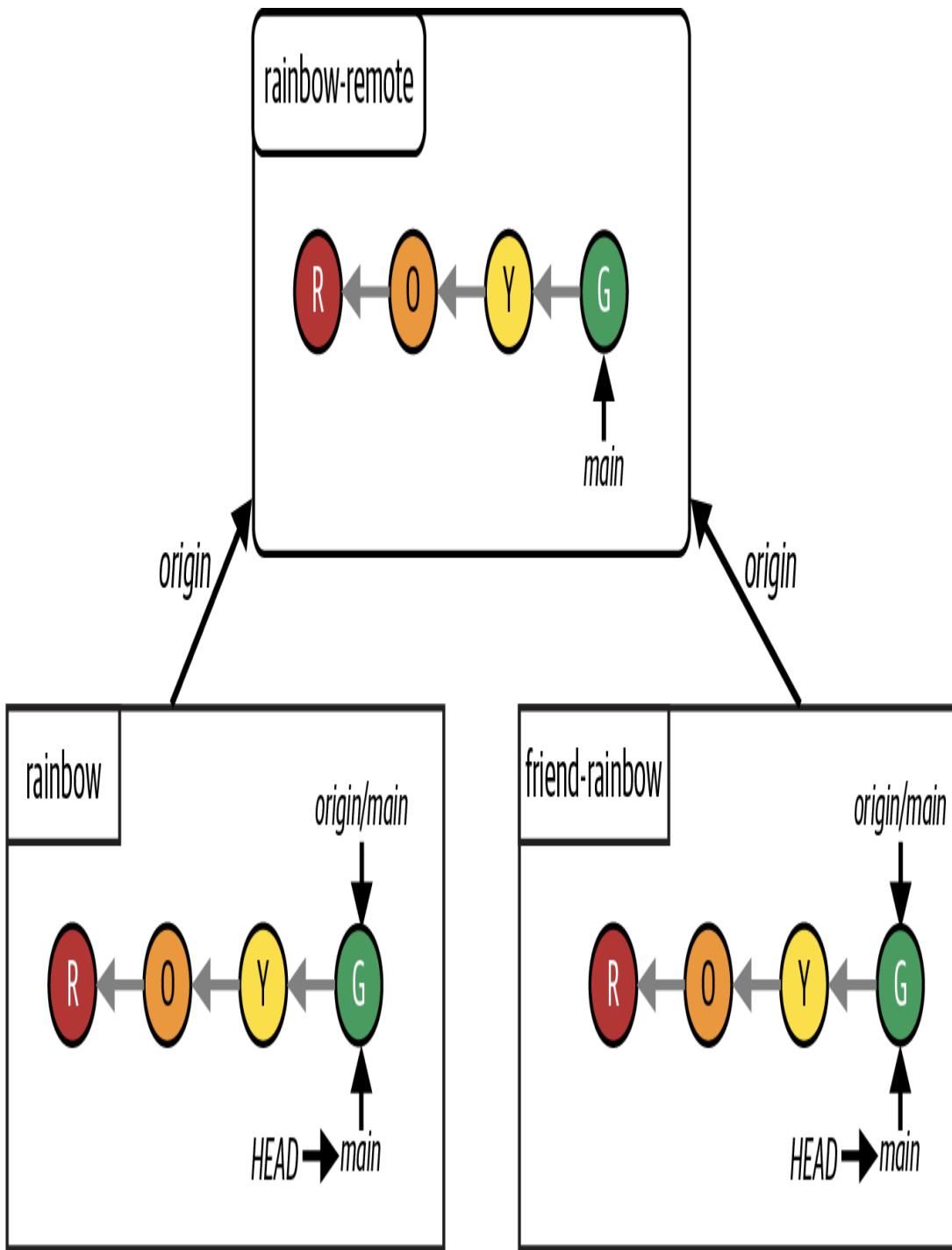


FIGURE C-7

An example of a Repository Diagram with two local repositories and one remote repository, representing the state of the Rainbow project at the end of [Chapter 8](#)

[*Index*]

A

- all (git branch), [107](#), [285](#)
- all (git log), [77](#), [284](#)
- ancestors of branches, rebasing and, [213](#)
- A option (git add), [36](#)
- a option (ls command), [11](#)
- app password, [92](#)
- arguments, [7](#)
- authentication
 - app password, [92](#)
 - credentials, [91](#)–[93](#)
 - personal access tokens, [92](#)
- author information, [40](#)

B

- Bitbucket, [22](#), [90](#)
 - app password, [92](#)
 - HTTPS, [92](#)
 - pull requests, [234](#)
- b option (git checkout), [86](#)
- b option (git init), [23](#)
- branches
 - as pointers, [45](#)
 - collaboration, [125](#)–[131](#)
 - commits, [45](#), [51](#)–[54](#)

creating, [53–55](#)

 and switching to, [86, 241](#)

default, [24](#)

 naming conventions, [48](#)

deleting, [122–124, 137–139](#)

development history, [65](#)

feature branches, [45](#)

git log command, [45](#)

HEAD, [55](#)

listing, [49, 53, 80](#)

local

 deleting, [123](#)

 integrating changes, [135–136](#)

 pushing to remote repository, [107–112](#)

 tracking, [106](#)

main, [23, 48](#)

managing, team rules, [125](#)

master, [23](#)

merging, [64–65](#)

origin/HEAD pointer, [120](#)

primary, [45](#)

rebasing and, [195](#)

 information, saving, [214](#)

 in practice, [218–223](#)

 preparations, [211–213](#)

remote, [106](#)

 deleting, [123, 249–251](#)

 tracking, [106](#)

remote-tracking, [106](#), [120](#)
source branch, [64](#)
switching, [57–60](#)
 creating and, [86](#), [241](#)
 merges and, [71–73](#)
 working directory and, [75–77](#)
target branch, [64](#)
topic branches, [45](#), [126](#)
upstream, [129](#), [147–150](#), [242–245](#)
uses, [44–45](#)
visualizing, [57](#)

C

cd command, [12](#), [283](#)
checking out commits, [80–85](#)
clear command, [8](#), [283](#)
clearing command line, [8](#)
CLI (command line interface), [3](#)
cloning, [134](#)
 origin shortname, [103](#)
 project directory names, [116](#)
 upstream branches and, [147](#)
collaboration, [2](#)
 branches and, [125–131](#)
 simulation, [118–119](#)
command line window, [3](#). *See also* integrated terminals
 applications, [4](#)
 clearing, [8](#)

closing, [16–17](#)
command execution, [5](#)
command prompt, [3](#)
current directory, [3](#)
cursor, [5](#)
directories, [9–10](#)
opening, [3–4](#)
Vim text editor, [161–163](#)

command prompt, [3](#)
directories, [5](#)
commands, [5](#)
arguments, [7](#)
cd, [12](#), [283](#)
clear, [8](#), [283](#)
git add, [36](#), [38](#), [284](#)
git branch, [53](#), [123](#), [148](#), [284](#), [285](#), [286](#)
git cat-file, [52](#)
git checkout, [57–58](#), [80–85](#), [284](#)
git clone, [116](#), [285](#)
git commit, [8](#), [38](#), [284](#)
git config, [17](#), [283](#)
git fetch, [133](#), [285](#)
git init, [23](#), [96](#), [97](#), [283](#)
git log, [40–42](#), [284](#)
git merge, [70](#), [78](#), [284](#), [286](#)
git pull, [168–171](#), [286](#)
git push, [96](#), [106](#), [123](#), [129](#), [147](#), [285](#)
git rebase, [195](#), [213](#), [286](#)

git remote, [104](#), [285](#)
git restore, [205](#)–[206](#), [206](#)
git status, [34](#)–[36](#), [49](#), [73](#), [185](#)–[187](#), [205](#), [284](#)
git switch, [57](#)–[58](#), [71](#), [284](#)
git version, [6](#), [7](#)
ls, [11](#), [283](#)
mkdir, [14](#), [283](#)
options, [7](#)
output, [6](#)
pwd, [5](#), [9](#), [283](#)
comments, pull requests, [236](#)–[237](#), [247](#)
commit history, [29](#)
 Git Diagram, [29](#)
commit ID, [28](#). *See also* commit hash, under commits
commit message, [36](#), [38](#)–[39](#)
commits, [28](#)–[29](#)
 author information, [40](#)
 branches and, [45](#)
 checking out, [80](#)–[85](#)
 commit hash, [28](#), [51](#)
 date and time, [41](#)
 git commit command, [38](#)
 git log command, [45](#)
 git status command, [34](#)–[37](#)
 list, [40](#)–[42](#), [77](#)
 parents, [52](#), [64](#)–[65](#)
 uses, [34](#)
 visual representation, [52](#), [287](#)

configuration setting, [17–20](#)
conflict markers, [176](#), [182](#)
-c option (git switch), [86](#), [240](#)
cursor, command line window, [5](#)

D

default branch, [48](#), [100](#)
 naming conventions, [48](#)
deleting branches, [122–124](#), [137–139](#)
desktop directory, [13](#)
detached HEAD state, [56](#), [81–84](#)
development history, [67](#), [143](#), [159](#), [211](#)
directories, [9](#)
 command prompt, [3](#)
 contents, viewing, [10](#)
 creating, [14–16](#)
 current directory, [3](#)
 desktop, [13](#)
 friend-rainbow, [115](#)
 .git, [22](#)
 heads, [47](#), [57](#)
 hidden, [22](#)
 navigating into and out of, [12–15](#)
 objects, [29](#)
 rainbow, [15](#), [21](#)
 working directory, [26](#)
-d option (git branch), [123](#)

E

editors. *See* text editors

email. *See* user.email variable

explicit merges, [238](#)

F

fast-forward merges, [64](#), [65](#), [70–78](#)

feature branch, [61–62](#)

- development history, [65](#)

- main branch and, [55–56](#)

- merges and, [70](#)

- pushing to remote repository, [110](#)

feature branches, [45](#)

File Explorer (Microsoft Windows), [9](#)

files

- adding to projects, [30–32](#)

- editing

- multiple times between commits, [151–158](#)

- save in text editor, [48](#)

- .gitconfig, [17](#)

- hidden, viewing, [22](#)

- index, staging area, [27–28](#), [36–38](#), [205](#)

- merge conflict markers, [176](#)

- modified, [48–51](#)

- staging, [201–211](#)

- state, [49](#)

- tracked, [35](#), [36](#), [40](#)

- unmodified, [48–51](#)

- unstaging, [201–211](#)
- untracked, [31](#), [35](#), [40](#)
- working simultaneously, [158–161](#)

filesystem window, [9](#)

Finder (macOS), [9](#)

friend-rainbow repository, [115–116](#)

- origin/HEAD pointer, [120](#)
- origin shortname, [122](#)

G

- .git directory, [22](#)

Git

- installing, [7](#)
- overview, [1](#)
- version number, [7](#)

git add command, [36](#), [284](#)

Git Bash, [4](#)

git branch command, [53](#), [123](#), [148–149](#), [284](#), [285](#), [286](#)

git cat-file command, [52](#)

git checkout command, [57–58](#), [80–85](#), [284](#)

git clone command, [116](#), [285](#)

git commit command, [8](#), [38](#), [284](#)

git config command, [17](#), [283](#)

.gitconfig file, [17](#)

Git Diagram, [288](#)

- building, [23–31](#)
- commit history, [29](#)
- local repository, [25](#)

staging area, [27–28](#)
working directory, [26–27](#)

git fetch command, [133](#), [285](#)

GitHub, [22](#), [90](#)
 HTTPS and, [92](#)
 personal access token, [92](#)
 pull requests, [234](#)

git init command, [23](#), [96](#), [97](#), [283](#)

GitLab, [22](#), [90](#)
 HTTPS and, [92](#)
 passwords, [92](#)
 pull requests, [234–235](#)

git log command, [40–42](#), [284](#)
 branches, [45](#)
 output, [41](#)

git merge command, [70](#), [78–80](#), [284](#), [286](#)

git pull command, [168–171](#)

git push command, [96](#), [106](#), [123](#), [129](#), [147](#), [285](#)

git rebase command, [168](#), [195](#), [213](#), [286](#)

git remote command, [104](#), [285](#)

git restore command, [205–206](#), [286](#)

git status command, [34](#), [49](#), [185–187](#), [205](#), [284](#)

git switch command, [57–58](#), [71](#), [284](#)

git version command, [6](#), [7](#)

--global option (git config), [17–19](#)

GUI (graphical user interface), [3](#)

hashes, [28](#)
HEAD, [55–57](#), [81–85](#)
 rebasing, [215](#)
heads directory, [47](#), [57](#)
hidden directories, [22](#)
hidden files, [22](#)
histories. *See* commit history; development history
hosting services, [22](#), [89](#)
 account setup, [90](#)
 authentication, [91](#)
 Bitbucket, [22](#), [90](#)
 GitHub, [22](#), [90](#)
 GitLab, [22](#), [90](#)
 HTTPS (Hypertext Transfer Protocol Secure), [91](#), [92](#)
 licenses, [100](#)
 pull requests, [234–235](#)
 creating, [245](#)
 remote repositories, [90](#)
 creating, [100](#)
 direct work, [112](#)
 SSH (Secure Shell), [89](#), [93–94](#)
HTTPS (Hypertext Transfer Protocol Secure), [89](#), [92](#)

|

index file, staging area, [27](#), [37–38](#)
init.defaultBranch variable, [24](#)
initializing repositories, [22–25](#)
integrated terminals, [19](#)

L

licenses, hosting services, [100](#)
--list (git config), [17](#)
local branches
 creating, [53–55](#)
 deleting, [123](#)
 merges, [64–69](#), [135–136](#)
 pushing to remote repository, [107–112](#)
 switching, [57–60](#)
 tracking, [106](#)
local repositories, [21](#)
 cloning from remote, [114–122](#)
connections
 listing, [104](#)
 multiple, [102](#)
friend-rainbow, [115](#)
Git Diagram, [25](#)
initializing, [22–26](#), [97](#)
interaction with remote repositories, [97](#)
name same as remote repository, [100](#)
rainbow, [24](#)
Repository Diagram, [44–45](#), [288](#)
starting work from, [96–97](#)
ls command, [11](#), [283](#)

M

macOS, [4](#)
 command line application, [4](#)

command prompt, [3](#)
filesystem application, [9](#)
installing Git, [9](#)
viewing hidden files and directories, [22](#)
main branch, [23](#), [48](#)
 commits, [47](#)
 feature branch and, [55](#)
master branch, [23](#), [48](#). *See also* main branch
merge conflicts, [69](#), [143](#), [175–177](#)
 aborting merge, [184](#)
 markers, [176](#), [182](#)
 rebasing, [175](#), [217–218](#)
 resolution process, [176](#), [182–184](#)
 git status and, [185–187](#)
 scenario setup, [177–183](#)
merge requests, [234](#). *See also* pull requests
merges, [64–65](#)
 aborting, [184](#)
 explicit, [238](#)
 fast-forward merges, [64](#), [65](#), [70–78](#)
 git merge command, [70](#), [78–80](#)
 non-fast-forward merges, [239](#)
 pull requests and, [233](#), [237–240](#), [248–249](#)
 source branch, [64](#)
 steps, [70](#)
 switching branches, [71–73](#)
 target branch, [64](#)
 three-way merges, [64](#), [67–70](#), [182](#)

executing, [163–168](#)
histories and, [143](#)
importance, [143–146](#)
merge conflicts, [175–176](#)
scenario setup, [146–147](#)
Vim and, [161–163](#)
types of merges, [64–69](#)

Microsoft Windows, [4](#)
command line application, [4](#)
command prompt, [3](#)
filesystem application, [9](#)
installing Git, [9](#)
viewing hidden files and directories, [22](#)

mkdir command, [14, 283](#)
modified files, [48–51](#)
-m option (git commit), [38, 161](#)

N

naming conventions, default branch, [48](#)
non-fast-forward merges, [239](#)

O

objects directory, [29](#)
options, [7](#)
origin/feature remote-tracking branch, [111, 118, 120, 124, 137–138](#)
origin/HEAD pointer, [120](#)
origin/main remote-tracking branch, [109–110, 118, 128–130, 135](#)
origin shortname, [122](#)

origin/topic remote-tracking branch, [249–250](#)

P

parent of commit, [52](#)

password, GitLab, [92](#)

personal access tokens, [92](#)

-p option (git cat-file), [52](#)

-p option (git fetch), [137](#)

-p option (git pull), [250](#)

prerequisites

all chapters, [260](#)

chapter 2, [261](#)

chapter 3, [261–263](#)

chapter 4, [262–263](#)

chapter 5, [263–265](#)

chapter 6, [265–267](#)

chapter 7, [265–267](#)

chapter 8, [267–271](#)

chapter 9, [270–273](#)

chapter 10, [273–276](#)

chapter 11, [276–280](#)

chapter 12, [279–282](#)

private repositories, [100](#)

project directories

cloning names, [116](#)

converting to local repository, [22](#)

projects

files, adding, [30–32](#)

sample, [15](#)
public repositories, [100](#)
pulling data, from remote repository, [167–171](#)
pull requests, [233](#). *See also* merge requests
 approving, [246–248](#)
 benefits, [235–238](#)
 closing, [233](#)
 collaborators and, [247](#)
 comments, [247](#)
 hosting services, [234–235](#)
 creating on, [245](#)
 merging, [237–240](#), [248–249](#)
 opening, [233](#)
 preparations, [240–242](#)
 reviewing, [246–248](#)
 steps, [234](#)
pushing to remote repository, [96](#), [129](#)
pwd command, [5](#), [9](#), [283](#)

R

rainbow directory, [15–16](#), [21](#)
 convert to Git repository, [24](#)
 empty, [23](#)
 rainbowcolors.txt file, [30–32](#)
Rainbow project, [6](#)
 collaboration, [115–119](#)
 othercolors.txt file, [146](#), [175](#), [204–208](#)
 rainbowcolors.txt file, [30–31](#)

- as tracked file, [40](#), [48](#)
- as unmodified file, [49](#)
- text editor, [19–20](#)
- rainbow-remote repository, [100–102](#)
 - cloning, [116](#)
- rainbow repository, [22](#)
 - commit history, [29](#)
 - Repository Diagram, [44](#), [288](#)
 - working directory, [26](#)
- remote branches, [106](#)
 - deleting, [123](#), [249–251](#)
 - tracking, [106](#)
- remote repositories, [21](#)
 - authentication, [91–93](#)
 - cloning, [114](#), [116](#)
 - local, [103](#)
 - connections
 - adding, [102–106](#)
 - listing, [104](#)
 - creating with data, [99–103](#)
 - fetching changes, [133](#)
 - hosting services and, [90](#)
 - interaction with local repositories, [97](#)
 - project names, [100](#)
 - public versus private, [100](#)
 - pulling data, [167–171](#)
 - pushing local branches to, [107–112](#)
 - reasons to use, [98](#)

shortname, [102](#)
starting work from, [97](#)
URLs/URIs, [100](#), [102](#), [268](#)
working directly on hosting service, [112](#)

repositories

- cloning, upstream branches, [129](#)
- friend-rainbow, [115](#)
 - origin shortname, [122](#)
- initializing
 - git init command, [23](#)
 - master branch, [23](#)
- interactions, [97](#)
- local, [21](#)
 - connections, multiple, [102](#)
 - friend-rainbow, [115](#)
 - Git Diagram, [25](#), [288](#)
 - initializing, [22–26](#), [97](#)
 - rainbow, [24](#)
 - Repository Diagram, [44–45](#), [288](#)
 - rainbow, [27](#), [115](#)
 - commit history, [29](#)
 - Repository Diagram, [44](#), [288](#)
 - working directory, [26](#)
- remote, [21](#), [268](#)
 - cloning, [114](#), [116](#)
 - connections, adding, [102–106](#)
 - creating with data, [99–103](#)
 - fetching changes, [133](#)

hosting services and, [90](#)
project names, [100](#)
public versus private, [100](#)
pulling data, [167–171](#)
pushing local branches to, [107–112](#)
reasons to use, [98](#)
starting work from, [97](#)
URLs, [100](#), [102](#)
working directly on hosting service, [112](#)

Repository Diagram, [44](#), [288–289](#)
local repository, [44–45](#)

S

settings, special, [113](#)
--set-upstream option (git push), [242](#)
--set-upstream-to option (git branch), [148](#)
shortnames, [102](#)
 origin shortname, [122](#)
snapshot, [28](#)
source branch, [64](#)
SSH (Secure Shell), [89](#), [93–94](#)
--staged option (git restore), [205](#)
staging area, [27](#)
 files
 adding, [36](#)
 modified, [205](#)
Git Diagram, [27–28](#)
rainbowcolors.txt file, [37–39](#), [50](#)

unstaging, [205](#)

T

target branch, [64](#), [175](#)
Terminal (macOS), [4](#)
text editors, [19–20](#)
 integrated terminals, [19–20](#)
 modified files, [48](#)
 special settings, [113](#)
 unmodified files, [48](#)
 Vim, [161–163](#)
 Visual Studio Code, [19](#)
three-way merges, [64](#), [67–70](#), [145](#)
 executing, [163–167](#)
 histories, and, [143](#)
 importance, [143–146](#)
 merge conflicts, [175–176](#)
 scenario setup, [146–147](#)
 Vim and, [161–163](#)
topic branch, [241–242](#), [248–250](#)
topic branches, [45](#), [126](#)
tracked files, [40](#)

U

unmodified files, [48–51](#)
unstaging files, [201–211](#)
untracked files, [40](#)
-u option (git branch), [148](#)

upstream branches, [106](#)

cloning and, [147](#)

defining, [129](#), [147–150](#), [242–245](#)

URLs, remote repositories, [100](#)

user.email variable, [17](#)

user.name variable, [17](#)

V

version control system, [1](#)

Vim, [161–163](#)

Visual Studio Code, [19](#)

-v option (git remote), [104](#)

-vv option (git branch), [129](#)

W

Windows Explorer (Microsoft Windows), [9](#)

word processors, [19](#)

working directory, [26](#)

files

modified, [48](#)

unmodified, [48](#)

git add command, [38](#)

Git Diagram, [26–27](#)

switching branches and, [75–77](#)

About the Author

Anna Skoulikari is a creative who has used her communication skills to teach Git in a simple, tangible, and visual manner. She teaches Git through a variety of media including her highly rated online course. Throughout her tech journey, she has worked as a UX designer, frontend developer, and technical writer.

Colophon

The animal on the cover of Learning Git is a gold-fronted green leafbird (*Chloropsis aurifrons*), a bird native to the Indian subcontinent, southwestern China, and southeast Asia.

This species of leaf bird is a small, bright-green bird with a black face and throat with a patch of orange and gold on its forehead. Long, sharp beaks and brush-tipped tongues helps them feed on insects from bark and tree leaves. This bird is a common sight in tropical forests and gardens, where it feeds on insects, fruit, and even—when hovering in a hummingbird-like holding pattern—nectar.

This bird is very territorial and aggressive towards other birds, and it is known to attack other animals, including humans, if it feels threatened.

Chloropsis aurifrons breeds in the spring, building a small, cup-shaped nest from sticks and leaves. The female lays two or three eggs, which the male incubates for two weeks. The chicks stay in the nest for another two weeks before fledging.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. The cover illustration is by Karen Montgomery, based on a black and white engraving from Lydekker's Royal Natural History. The cover fonts are Gilroy Semibold and Guardian Sans. The text fonts are Scala Pro and Gotham Narrow; the heading fonts are Gotham Condensed and Gotham Narrow; and the code font is Courier Prime.

I dedicate this book to my parents.

*Mom, Dad, thank you for all your support
through this crazy journey we call life.*

[*contents*]

[[Preface](#)] xi

[Chapter 1. Git and the Command Line](#) 1

[What Is Git?](#) 1

[The Graphical User Interface and the Command Line](#) 2

[Opening a Command Line Window](#) 3

[Executing Commands in the Command Line](#) 5

[Installing Git](#) 7

[Command Options and Arguments](#) 7

[Clearing the Command Line](#) 8

[Opening the Filesystem Window](#) 9

[Working with Directories](#) 9

[Closing the Command Line](#) 16

[Setting Git Configurations](#) 17

[Preparing a Text Editor](#) 19

[Integrated Terminals](#) 19

[Summary](#) 20

[Chapter 2. Local Repositories](#) 21

[Current Setup](#) 21

[Introducing Repositories](#) 21

[Initializing a Local Repository](#) 22

[The Areas of Git](#) 26

[Adding a File to a Git Project](#) 30

[Summary](#) 31

[Chapter 3. Making a Commit](#) 33

[Current Setup](#) 33

[Why Do We Make Commits?](#) 34

[The Two Steps to Make a Commit](#) 34

[Viewing a List of Commits](#) 40

[Summary](#) 42

[Chapter 4. Branches](#) 43

[State of the Local Repository](#) 43

[Why Do We Use Branches?](#) 44

[Unmodified and Modified Files](#) 48

[Making Commits on a Branch](#) 51

[Creating a Branch](#) 53

[What Is HEAD?](#) 55

[Switching Branches](#) 57

[Working on a Separate Branch](#) 61

[Summary](#) 62

[Chapter 5. Merging](#) 63

[State of the Local Repository](#) 63

[Introducing Merging](#) 64

[Types of Merges](#) 64

[Doing a Fast-Forward Merge](#) 70

[Checking Out Commits](#) 80

[Creating a Branch and Switching onto It in One Go](#) 86

[Summary](#) 87

[Chapter 6. Hosting Services and Authentication](#) 89

[Hosting Services and Remote Repositories](#) 90

[Setting Up a Hosting Service Account](#) 90

[Setting Up Authentication Credentials](#) 91

[Summary](#) 93

[Chapter 7. Creating and Pushing to a Remote Repository](#) 95

[State of the Local Repository](#) 95

[The Two Ways to Start Work on a Git Project](#) 96

[The Interaction Between Local and Remote Repositories](#) 97

[Why Do We Use Remote Repositories?](#) 98

[Creating a Remote Repository with Data 99](#)
[Working on a Remote Repository Directly on a Hosting Service 112](#)
[Summary 112](#)

Chapter 8. Cloning and Fetching 113

[State of the Local and Remote Repositories 113](#)
[Cloning a Remote Repository 114](#)
[Deleting Branches 122](#)
[Git Collaboration and Branches 125](#)
[Incorporating Changes from the Remote Repository 132](#)
[Deleting Branches \(Continued\) 137](#)
[Summary 139](#)

Chapter 9. Three-Way Merges 141

[State of the Local and Remote Repositories 141](#)
[Why Are Three-Way Merges Important? 143](#)
[Setting Up a Three-Way Merge Scenario 146](#)
[Defining Upstream Branches 147](#)
[Editing the Same File Multiple Times Between Commits 151](#)
[Working at the Same Time as Others on Different Files 158](#)
[Three-Way Merge in Practice 161](#)
[Pulling Changes from a Remote Repository 167](#)

[State of the Local and Remote Repositories 170](#)

[Summary 171](#)

[Chapter 10. Merge Conflicts 173](#)

[State of the Local and Remote Repositories 173](#)

[Introducing Merge Conflicts 175](#)

[How to Resolve Merge Conflicts 176](#)

[Setting Up a Merge Conflict Scenario 177](#)

[The Merge Conflict Resolution Process 182](#)

[Resolving Merge Conflicts in Practice 185](#)

[Staying Up to Date with a Remote Repository 187](#)

[Syncing the Repositories 188](#)

[State of the Local and Remote Repositories 191](#)

[Summary 191](#)

[Chapter 11. Rebasing 193](#)

[State of the Local and Remote Repositories 193](#)

[Integrating Changes in Git 195](#)

[Why Is Rebasing Helpful? 196](#)

[Setting Up the Rebasing Example 199](#)

[Unstaging and Staging Files 201](#)

[Preparing to Rebase 211](#)

[The Five Stages of the Rebase Process](#) 213
[Rebasing and Merge Conflicts](#) 217
[Rebasing a Branch in Practice](#) 218
[The Golden Rule of Rebasing](#) 223
[Syncing the Repositories](#) 226
[State of the Local and Remote Repositories](#) 229
[Summary](#) 229

Chapter 12. Pull Requests (Merge Requests) 231
[State of the Local and Remote Repositories](#) 231
[Introducing Pull Requests](#) 233
[Hosting Service Specifics](#) 234
[Why Use Pull Requests?](#) 235
[Understanding How Pull Requests Are Merged](#) 237
[Preparing to Make a Pull Request](#) 240
[An Easier Way to Define Upstream Branches](#) 242
[Creating a Pull Request on a Hosting Service](#) 245
[Reviewing and Approving a Pull Request](#) 246
[Merging a Pull Request](#) 248
[Deleting Remote Branches](#) 249
[Syncing the Local Repositories and Cleaning Up](#) 250
[State of the Local and Remote Repositories](#) 254

[Summary](#) 255

[[Epilogue](#)] 257

[Appendix A: Chapter Prerequisites](#) 259

[Appendix B: Command Quick Reference](#) 283

[Appendix C: Visual Language Reference](#) 287

[[Index](#)] 291