**Excerpt from "Learning to Program" by Steven Foote
(Addison-Wesley Professional, 2014)**

15

Advanced Topics

> **Note**
> **Project:** Install Git and turn your Chrome extension project into a Git repository.

You made it! I want to thank you for letting me be a part of your journey to learn programming. You have learned so much already, and you are ready to keep learning. You can now have a conversation with software developers and understand what they're talking about (at least some of it). You are ready to pick up any programming book or start any programming tutorial online and actually understand what is going on, instead of being completely lost and frustrated. Most important, you are ready to start working on your own projects. You should be proud of what you have accomplished.

Your head is so full of new programming terminology and concepts right now that you might feel that nothing else will fit up there. If that is the case, you can close this book right now and walk away with a well-deserved feeling of accomplishment. This chapter is an introduction to a few advanced programming topics you should consider learning about next—a bonus, if you will. If now is not the time for advanced topics, I understand; take a break and come back in a few days. However, if you are itching to learn more, keep reading.

Version Control

You have learned about tools to write programs (text editors and IDEs), to compile and build programs (build tools such as Grunt), and to debug programs (such as Chrome Dev Tools). Like the others, a version control system is an important programming tool. A version control system is a tool for managing changes made to your program. Version control allows you to take a snapshot of your project every time a change is made to the source code. Each snapshot is accompanied by a description of what changes were made and why. This type of meticulous record keeping is useful even if you don't plan to publish a detailed biography of your project.

If your software stops working for some reason, version control enables you to easily roll back to a previous (working) version while you figure out the problem.

When a project is using version control, all the code for the project is stored in a central location, called a repository. To start working on the project, you must "check out" a local copy of the code from the repository (*local* means the files are on your computer). You make changes to your local copy of the code, and when your changes are ready, you "check in" to the repository. Checking in means sending to the repository the changes you made, along with a message where you describe those changes. By checking in, you create a new snapshot of your project at that point. See Figure 15.1 for an example of how this process works.

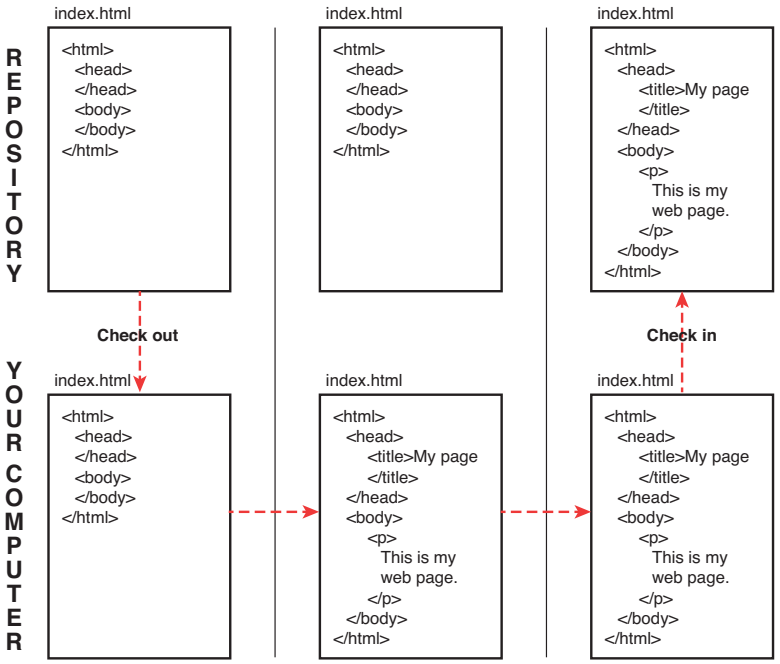

Figure 15.1 A simple visualization of how version control works

Why Use Version Control?

Beyond its capability to track changes in your project, one of the most compelling reasons to use version control is for collaboration. As you will see in the next section, the way version control is set up makes it ideal for working in small and large teams. Version control also helps you write code in a more focused manner. As with functions, each check-in should have a specific purpose. When working within a version control system, you should make only changes that relate to the current check-in, which helps you stay focused. Finally, version control enables you to experiment without fear of ruining everything. The capability to roll

back means that even if you do ruin everything, you can quickly get back to a good place. Beyond rollbacks, the concept of branches (see section on Branches below) means you can experiment without ever checking in. Version control is worth the time to learn, whether you are working alone or on a team.

Working with a Team

You have already seen how you can work with a repository by checking out and checking in. This process of checking out and checking in allows many people to effectively work on the same project at the same time without getting in each other's way. If you and I were to work on a project together, we would create a central repository and then each check out a local copy. Let's say we're building a website together, and you want to start writing the JavaScript while I start writing HTML. When you check in your JavaScript changes, the repository is updated, but my copy of the code is left unchanged. When I check in my HTML changes, the repository is updated, but I still don't have your JavaScript changes and you still don't have my HTML changes. To get the latest changes from the repository, we each have to request an "update" from the repository. The repository updates us with the code that has changed since our last update. Figure 15.2 shows a visualization of how this process works.

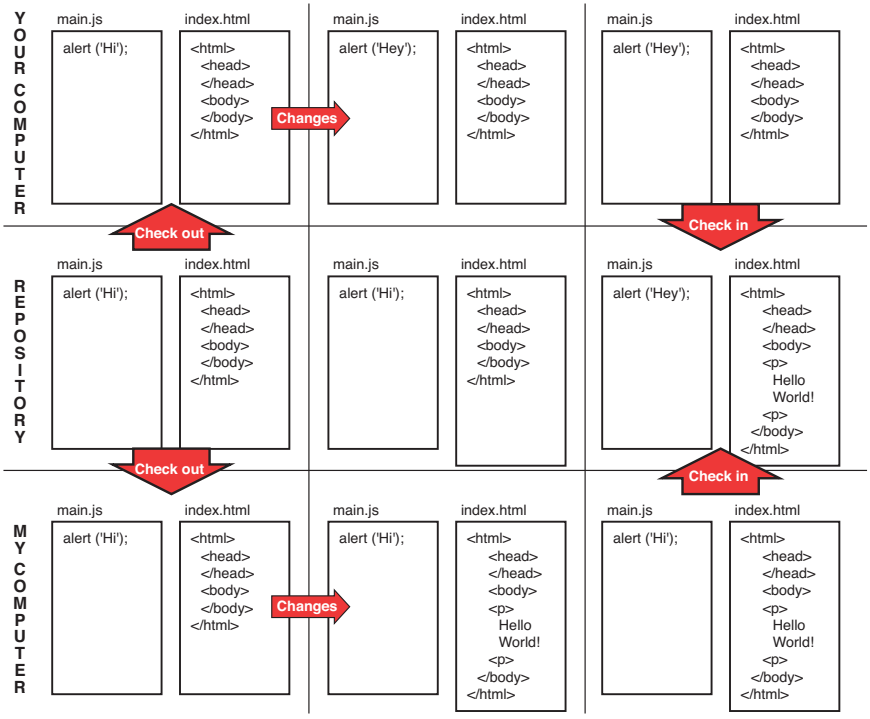

Figure 15.2 A series of check-ins and updates, to keep everyone in sync

Now let's say that you need to make some changes to the HTML file that I am still working on. How will that work? You make your change to the HTML file and check in, and then I finish my change to the HTML file and try to check in. When I try to check in, the version control system will notice that my version of the HTML file is outdated because you checked in your changes since the last time I updated. I will have to get your update from the repository before I am allowed to check in, to make sure that the changes I am checking in don't overwrite the changes you just checked in. This is where version control works its magic. When I request an update from the repository, the version control system tries to merge your changes into my copy of the HTML file. If conflicts arise, the version control system highlights the conflicts and informs me that I must resolve them before I can check in (see Figure 15.3). One example of a conflict is if you and I both try to edit the same line of a file. I can resolve that conflict by manually merging your changes into mine.

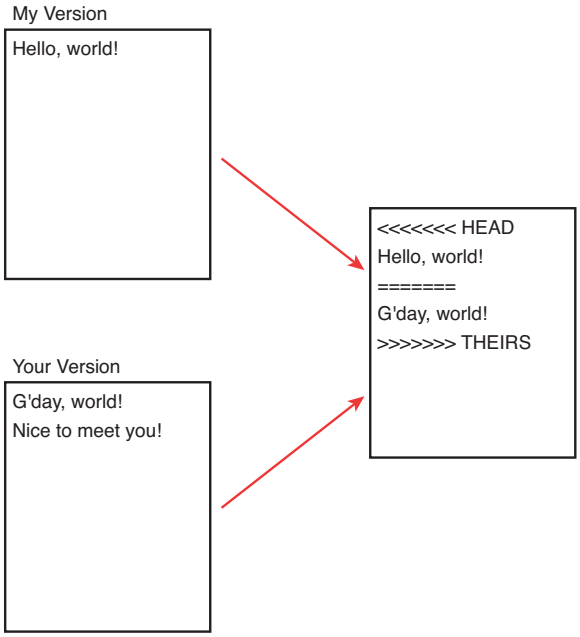

Figure 15.3 The version control system merges as much as it can for you and highlights any conflicts that it is unable to merge.

The process of merging and conflict resolution allows us both to work on the same code at the same time without fear of overwriting each other's changes. In the context of working on a team, this process can have a huge impact on productivity. I can work on any file in our project at any time, and I don't have to worry about whether you or anyone else is already working on it. The impact is even greater when you work on a team of 20 or 200 or larger. Working with a team on a programming project would be nearly impossible without version

control. You would basically need to ask permission before editing any file. The following email conversation is how I imagine collaborative software development without version control might work. It's not pretty.

> Monday, April 14, 2014, 8:32 AM
>
> Hey team,
>
> Happy Monday :P I wanted to start working on some changes to index.html this morning. Is anyone else already editing that file?
>
> Thanks,
>
> Steven

> Monday, April 14, 2014, 8:50 AM
>
> Hey Steven,
>
> I think Jerry was working on index.html this morning. You better check with him.
>
> Paul

> Monday, April 14, 2014, 8:52 AM
>
> DO NOT TOUCH index.html! Jerry is working on it, and I'm next in line.
>
> Harry

> Monday, April 14, 2014, 9:03 AM
>
> Hey everyone,
>
> I am in line for index.html, too. Maybe we should set up a sign-up sheet in the breakroom?
>
> Thanks,
>
> Sally

> Monday, April 14, 2014, 9:08 AM
>
> Thanks everyone.
>
> Sally, that sounds like a great idea. Can you put my name on the list as well?
>
> Thanks,
>
> Steven

> Tuesday, April 15, 2014, 12:17 AM
>
> Just a minute, mates. A sign-up sheet in your breakroom won't help those of us down here in Sydney. I need to makes some changes to index.html, too.
>
> Cheers,
>
> Ryan

With version control, I can edit any file whenever I want and let the software handle the merging. If there are any conflicts, I can deal with them as they arise. Version control makes working on teams hugely more efficient. No sign-up sheets, no lines, no email—just do your work and let version control handle the hard part.

> **Code Reviews**
>
> One of the greatest advantages of working on a team of developers is that you can learn from the code the other members of your team write, and they can learn from your code. A common practice on software development teams is code reviews, in which other members of the team must review and approve your code changes before they can be checked in. These code reviews might seem intimidating at first, but they can be incredibly valuable for everyone involved. The overall quality of the project's code tends to increase, and so does the knowledge and expertise of the team as a whole. If you are not working on a team, you can still get code reviews online. The people who built StackOverflow created a public place for code reviews at http://codereview.stackexchange.com/. Don't be shy—ask for a code review!

Subversion

Subversion is one of the more popular version control systems available today. The basic features of version control are available in all version control systems, but the features and implementation of each system are slightly different. Subversion is a centralized version control system, which means that Subversion has a central server where the repository is stored. Each developer checks out a copy of the repository at a given point in time, but only the server contains the actual revision history. Whenever you want information about the history of the project, you must have a working network connection and a running server. Therefore, if the server breaks down, the entire system breaks down. If one piece of a system can cause the entire system to break down, that piece is referred to as a single point of failure, and a single point of failure is not a good thing. The server that holds the central repository is Subversion's single point of failure. Nevertheless, Subversion is still a popular version control system in use on some very large projects (including Google Chrome). As long as you take precautions to keep your server safe and running, Subversion works well. Other centralized version control systems exist, but Subversion is the most prevalent.

Git

Git is my version control system of choice. The original author of Git (Linus Torvalds) is also the original author of Linux, and Git was written to be the version control system for the Linux development team. Git was designed to be fast and to work well for both small and large teams working on small and large projects. Git also solves the single point of failure problem of

centralized version control systems. With Git, the entire revision history is on every computer, not just the server. If the server breaks down, any of the other computers can immediately step in and act as the server. You have already learned a bit about Git in the context of GitHub, but now you'll learn how it actually works.

Clone

Because Git has a different philosophy about how version control works, it uses a different set of terms. First, you don't just "check out" a copy of the code on a repository; with Git, you are getting the code and the entire revision history. So instead of using the "check out" phrasing, Git uses the word *clone*—your local repository is a full clone of the server's repository. You clone a repository from the command line using the command `git clone`, with the URL of the server repository as an argument (see Listing 15.1).

Listing 15.1 **Cloning a Git Repository Using `git clone`**

```
# Create a clone of one of the first repositories I created on GitHub, a clock.
sfoote@sfoote-mac:projects $ git clone https://github.com/smfoote/html5-clock.git
sfoote@sfoote-mac:projects $ cd html5-clock
sfoote@sfoote-mac:html5-clock $ # Now I'm ready to start making changes
```

Add

After you have cloned the repository, you are ready to start making changes. Git gives you fine-grained control about how your changes are tracked. After you have made some changes, you need to tell Git which of those changes you want to track, using the `git add [filename]` command. Only changes that have been included in a `git add` will be included in the next snapshot (snapshots in Git are called commits). If you want to include all your changes in the next commit, you can use `git add -A` (the `-A` is for *all*). Note that `git add` does not add anything to Git's revision history; it only tells Git which changes to include in the next commit. You can see which changes will be included in the next commit by running the command `git status`. See Listing 15.2 for an example of using `git add`.

Listing 15.2 **Adding Files in Preparation for a Commit**

```
# make some changes to index.html
sfoote@sfoote-mac:html5-clock $ vim index.html

# Check which files will be included in the next commit
sfoote@sfoote-mac:html5-clock $ git status
```

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

      modified:   index.html

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


# add those changes to the next commit
sfoote@sfoote-mac:html5-clock $ git add index.html
sfoote@sfoote-mac:html5-clock $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  modified:   index.html
```

Commit

When you're ready to take a snapshot of your project, you use `git commit`, which adds a new entry to the revision history with all the changes you have included using `git add`. Any changes that you have made but not included with `git add` will still be there, but they will not be committed. The new entry you make in the revision history by running `git commit` exists only in your local repository. See Listing 15.3 for example commands for this process, and Figure 15.4 for a visualization of how it works. If you are working on a project by yourself, committing is the last step. However, if you're working with a team, you must take one more step to share your changes with everyone else on your team.

Listing 15.3 **Committing Changes with a Commit Message**

```
sfoote@sfoote-mac:html5-clock $ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

      modified:   index.html

sfoote@sfoote-mac:html5-clock $ git commit -m "Fix typo in index.html"
[master 1234567] Fix typo in index.html
 1 file changed, 3 insertions(+), 2 deletions(-)
```

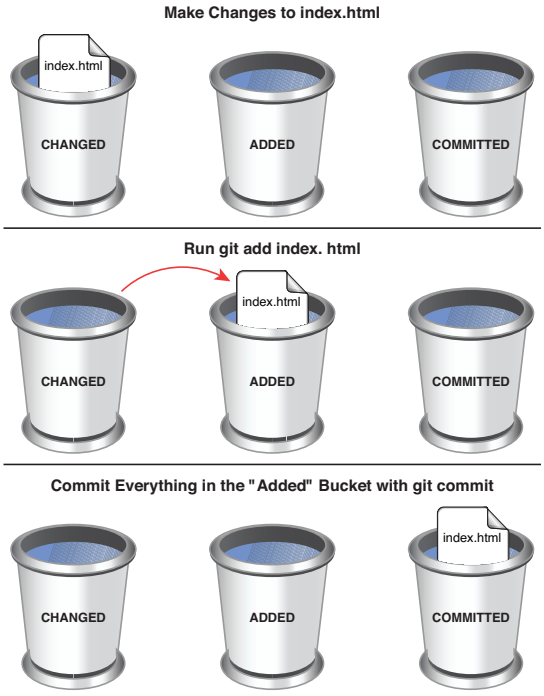

Figure 15.4 Only files in the Changed bucket can be moved to the Added bucket, and only files in the Added bucket are moved to the Committed bucket.

Push and Pull

Up to this point, none of the Git commands we have run (with the exception of `git clone`) required an Internet connection. That is one of the powerful features of Git; you can work on a Git repository if you're on a train, a bus, an airplane, or anywhere else with no Internet connection, whereas, almost everything you do with Subversion requires an Internet connection. However, when the time comes to sync your changes with the server's repository, you do need a connection. In Subversion, you check in changes, but in Git, you have already taken the snapshot with `git commit`, so you "push" your commits. To update your local repositories with commits made by other developers, you "pull." For basic Git usage, the server's repository is called the origin, and when you push and pull, you have to tell Git the name of the server's repository (see Listing 15.4 for an example).

Listing 15.4 **Syncing with the Server's Repository Using `git pull` and `git push`**

```
# Pull updates from the server's repository
sfoote@sfoote-mac:html5-clock $ git pull origin master
```

```
From github.com:smfoote/html5-clock
 * branch            master     -> FETCH_HEAD
Updating 68125d1..4525da7
Fast-forward
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

 # Push the commit I made locally to server's repository
 sfoote@sfoote-mac:html5-clock $ git push origin master
```

Git Workflow

Git takes a little while to learn and get used to. You might even be wondering why you'd want to go through all the hassle of using `git add` and `git commit` and `git push` and `pull`. Seems like an awful lot of extra work just to keep track of code, right? Let me just say that version control systems exist for a reason, and it is a very good reason. I didn't know there was such a thing as version control until about two years after I started programming, and at that point, I still didn't think it was necessary. After all, I had been programming for two years and I hadn't needed version control yet, so why would I take the time to learn some crazy system of check-ins, commits, updates, and pulls?

Then one day, the entire website I was building was broken. I had no idea why, and I had only a vague idea of what code I had been updating in the past few days. I had no way to roll back to a previous version of the code because I *had* no versions. I finally found and fixed the problem, but I still didn't want to learn version control. Instead, I set up a Google doc, and every time I made a change to the website, I wrote down the date, the purpose of the change, and what files I had modified. In other words, I was trying to create a version control system, but I was missing out on all the real advantages of a version control system. When I finally learned how to use Git, I never looked back; the benefits are worth all the hassle of adding, committing, and pushing. I now use Git on every project I work on (including this book).

One of the reasons I was hesitant to learn Git was that I didn't really understand how I was supposed to use it. I don't want you to have the same problem, so consider this very basic Git workflow:

1. Make some changes to the code in your repository.
2. Add your changes to the set of changes you want to commit.
3. Commit the changes you have included with `git add` using `git commit`.
4. Update your local repository with changes made by other developers using `git pull origin master`.
5. Push the commit snapshots you have created to the server's repository using `git push origin master`.

Kittenbook Gets Git

It's time to put your skills to the test. You have learned a lot about finding answers to your own problems, and now you have a problem. Kittenbook needs version control, and you need to figure out how to set it up. The first step is to install Git (*hint:* GitHub has some great resources on how to get started with Git). Second, when you have Git installed, you need to turn your kittenbook directory into a Git repository (*hint:* this requires only a single Git command from the command line, and it's not `git clone`). Finally, you need to create an initial snapshot of your project, which you can do by following the Git workflow just described.

I recognize that the task of setting Git for kittenbook might seem unnecessarily difficult. I have done my best throughout this book to give you as much direction as possible because I know how frustrating it can be when a book assumes that I know how to do something that I don't know how to do. But you are ready to walk on your own. I know you don't already know how to install Git, but I also know that you have the tools to figure it out on your own. You'll do great—but if you do get stuck, remember to use resources such as StackOverflow or experienced friends/acquaintances.

Branches

You should be aware of one more feature of Git, and that is branches. The concept of Git branches is at first a little difficult to understand (at least, it was for me). A Git branch is a way to keep different tasks neatly organized and separated. For example, let's say you are working on adding a new feature to kittenbook (for example, kittens on LinkedIn), but you have an ongoing task of updating all your documentation. Without branches, you would need to manually track which changes are documentation changes and which changes are LinkedIn kitten changes; you wouldn't want a partially complete feature to be committed with your documentation changes, and vice versa. With branches, you can create a branch called `documentation` where you work on your documentation changes, and you can create another branch called `linkedin-kittens` where you work on your new feature. You can easily switch between your branches to work on the different tasks; the changes you make on one branch are saved to that branch, but they don't come with you when you switch to another branch. When all your changes are ready, you can merge your branches into the "master" branch using `git merge`.

This introduction to Git is clearly incomplete. Git involves so much more than I can possibly cover in a few pages. If you are interested in learning more, I highly recommend that you check out the book *Pro Git,* by Scott Chacon. I did not think Git was that great, or really even understand Git, until I read this book. After reading it, I wanted to use Git for everything. The book is very well written, good for Git beginners, and available for free online at http://git-scm.com/book.

OOP (Object-Oriented Programming)

Now for something completely different, let's talk about object-oriented programming (OOP). You might already be familiar with this term, either from previous experience or from research you've done while reading this book, but you might not really understand what it means.

First let me say that OOP is not as confusing as it might seem, but it is very powerful. OOP is an effective way to organize and share code—and as a result, the code becomes easier to work with. The name *object-oriented programming* comes from the idea that your programs should literally be based on objects (the type of objects you learned about in Chapter 5, "Data (Types), Data (Structures), Data(bases)"). I don't find the name to be that helpful in understanding what OOP is, nor in explaining why I would want to learn and use it. Instead of objects, I like to think about my OOP programs in terms of actors. Video games are probably the easiest way of thinking of computer programs in terms of actors. For example, the game *Pac-Man* has a PacMan actor, several Ghost actors, a GameBoard actor, and Fruit actors, among others. In OOP terms, each type of actor is called a class, and an actual member of a class is called an instance.

When you think in terms of actors, you can write your program so that all the behaviors of a given actor are connected to that actor. For example, the PacMan actor has the ability to move around on the screen, so we could give the `PacMan` class a `move` method. Then whenever PacMan needs to move up, our code can just call `pacMan.move('up')`. As PacMan moves across the screen, his mouth opens and closes, so we could create another method called `chomp` and call `pacMan.chomp()` each time PacMan's mouth should open and close. After the methods of `move` and `chomp` are written, you don't have to worry about how they work. You know that they do work, so it doesn't really matter how. In this way, OOP lets you write a lot more functionality a lot faster with a lot less code.

Classes

An object-oriented program consists of a bunch of objects that have attributes and behaviors. A class is a description of the attributes and behaviors of a type of object. For instance, the code for the `PacMan` class would include the `move` and `chomp` methods, which are descriptions of how a PacMan object moves and chomps, respectively. The class is just a description of what a PacMan would be like, but it does not create a PacMan (that is the job of the instance). One of the big advantages of using classes is how it organizes your code. When you have a `PacMan` class, you know that all the code that describes a PacMan must go into the `PacMan` class definition. If sometime in the future you decide that you want PacMan to be able to jump, you will know just where to put the `jump` method.

Inheritance

We now have a pretty good idea of what the `PacMan` class would look like, with methods such as `move`, `chomp`, and (maybe) `jump`, but what about the other actors in the game? Most notably, what about the Ghosts? If you have played *Pac-Man*, you consider the Ghosts to be the enemy. But if you're thinking about how to program *Pac-Man*, you will notice that the Ghosts and PacMan are a lot alike. For instance, they can all move around the game board, but they can't go through walls. In Chapter 8, "Functions and Methods," you learned the importance of keeping your code DRY (don't repeat yourself), but if we write a `move` method for the `PacMan` class *and* a `move` method for the `Ghost` class, we will be repeating ourselves. The answer to this problem is inheritance.

Inheritance means that a class can be a type of another class. As an example, I am from Nevada and am therefore a member of the `Nevadan` class. All members of the `Nevadan` class have certain attributes and behaviors that are specific to `Nevadans` (such as `surviveRidiculousHeatWithLittleWater`), but all members of the `Nevadan` class are also members of the `American` class. The `Nevadan` class (the subclass) inherits all the attributes and methods of the `American` class (the parent class). All `Nevadans` are `Americans`, but not all `Americans` are `Nevadans`. The `Texan` class, for instance, also inherits from the `American` class, but from not the `Nevadan` class. The concept of inheritance allows for code to be shared among different classes. Both the `PacMan` class and the `Ghost` class can inherit from a parent class that we'll call `MovableCharacter`. The `move` method would be defined in the `MovableCharacter` class and shared by both the `PacMan` class and the `Ghost` class (see Figure 15.5).

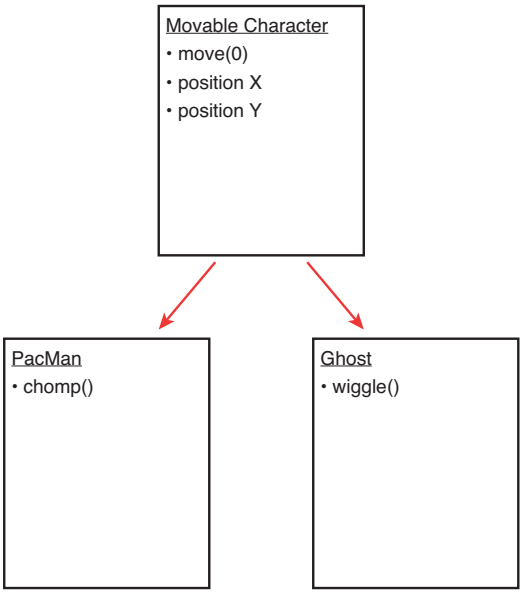

Figure 15.5 Although they have their differences, Pac-Man and the Ghosts share a common parent (class), `MovableCharacter`.

Instances

Like Coke, an instance is the real thing. Whereas the `PacMan` class is just the idea of what a real PacMan would be, an instance of the `PacMan` class is a real PacMan. It can `move`, it can `chomp`, it can `jump` (maybe?)—it's the real deal. The reason to keep classes and instances separate is so you can create multiple instances of a single class (for example, a game of *Pac-Man* has multiple Ghosts). Classes and instances keep your code organized and easy to understand.

Design Patterns

As you design and build software, remember that your software problems are not entirely unique, even if your product is. In fact, your problems are probably very similar to problems that have well-defined solutions. These well-defined solutions are referred to as design patterns. Your challenge is to choose the right design pattern and then implement it correctly to solve your problem. As you approach a difficult programming problem, don't reinvent the wheel; use a design pattern. Design patterns are meant to make your life a little easier.

The following list of design patterns is but a small sample. The purpose of this list is to give you an idea of what design patterns are good for and what types of problems they solve. Some design patterns describe how to architect an entire application; others describe how to design a relatively small part of an application. You can mix, match, and meld design patterns as you see fit. If you don't find an answer to your problem in this list, don't fret. Plenty of other established design patterns exist, and at least one of them will likely help you solve your problem.

Pub Sub

An entire software application usually consists of several parts, called modules. Consider, for example, a web browser, which has a Tabs module, an Address Bar module, a Bookmarks module, and a Webpage View module (see Figure 15.6). To keep the code clean and organized, the modules should not have direct references to each other (in programmer-speak, the modules should not know about each other), because if two modules directly reference each other, they might as well be one module. If all the modules directly reference all the other modules, you end up with one giant module that is hard to maintain and nearly impossible to understand. Yet the different modules need to be able to communicate in some way. For instance, when a tab is selected, the web page associated with that tab needs to be displayed. The Pub Sub pattern provides a solution to this communication problem.

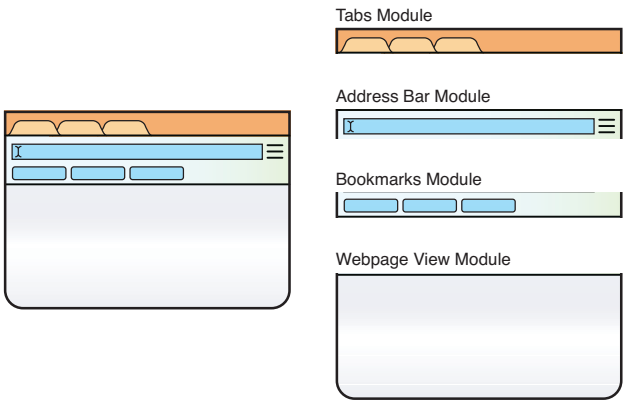

Figure 15.6 A web browser and its modules

Pub Sub (short for Publish/Subscribe) describes an events system (not unlike the events we discussed in Chapter 7, "`If`, `For`, `While`, and When"). When something important happens in one module, an event is published so that other modules can know that the event occurred and respond appropriately. A module must "subscribe" to an event to be notified that it has occurred. Returning to our web browser example, when a tab is selected, the Tab module would publish a `tabSelected` event, and the Address Bar module and Webpage View module (both subscribers of the `tabSelected` event) would know to update their content appropriately. One advantage of using events is that the program will behave correctly no matter where the event originates. If you want to add a feature that allows users to switch tabs by clicking a button in the Bookmarks module, all you have to do is publish a `tabSelected` event from the Bookmarks module, and everything else will just work.

Mediator

The Mediator pattern is like a more civilized version of the Pub Sub pattern. The Pub Sub pattern is just a bunch of modules yelling at each other: "I'M A TAB AND I HAVE BEEN SELECTED, IF ANYBODY CARES!" The trouble with all this yelling becomes apparent when there are conflicts. If two tabs publish the `tabSelected` event at the same time, which tab actually gets selected? If one tab publishes the `tabSelected` while another tab is in a frozen state (tabs can be frozen when an `alert` or `prompt` window is open), do the tabs switch anyway? Should they? When modules can communicate only through events, these types of conflicts can result in some strange behavior. If you try to fix this behavior while still using only events, your modules might start to directly reference each other, until eventually the whole Pub Sub system falls apart. Instead, your code needs a way to mediate these conflicts. The Mediator pattern provides a solution.

The Mediator is a module that is allowed to know about all the other modules. The mediator module contains the code that is used to resolve conflicts. For instance, if a tab wants to be selected (because the user clicked on it), the tab asks the mediator for permission to be selected. The mediator checks to see if any of the other modules have a reason why the tab should not be selected. If the tab is allowed to be selected, the appropriate code runs and the tab is selected. If there is a reason why the tab should not be selected, the mediator decides how to handle the conflict (see Figure 15.7). This Mediator pattern works great for programs in which modules might conflict with each other.

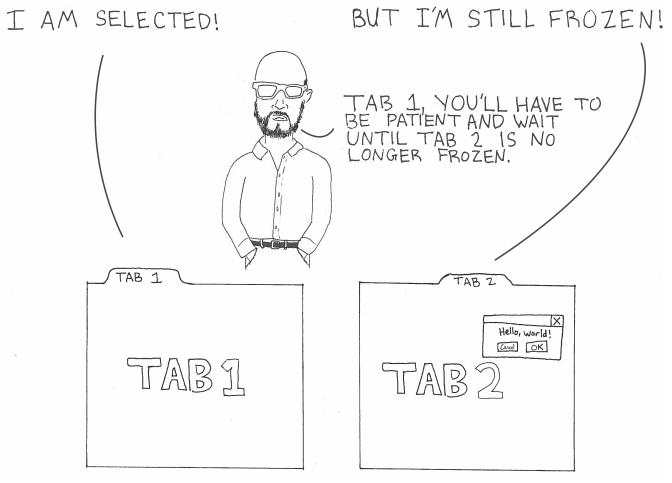

Figure 15.7 A good mediator resolves conflicts in a way that works for everyone.

Singleton

The Singleton pattern is a type of class that can have only one instance. Every time you try create to create an instance of a Singleton, you don't actually get a new instance—you are really getting only a reference to a single instance. For example, the `PacMan` class could be implemented as a Singleton. You might try to create an instance of PacMan in several different places in your code, but you really want only one PacMan. If the `PacMan` object has already been created, you don't actually want to create a second PacMan; you really just want a reference to the existing PacMan.

Summing Up

Congratulations! You are done, and you even made it through the advanced topics. You should celebrate. The skills you have learned in this book will change the way you see computers—and might even change the way you see the world. These skills are too valuable to keep to yourself. Now that you're done reading it, you should lend this book to a friend. Then you can joke together about using your grandma's toothbrush.

In this chapter, you learned about:

- Version control
- Subversion
- Git
- Object-oriented programming
- Classes
- Instances
- Design patterns

It's up to you to write the next chapter of your programming journey. Enjoy!