Blueprints:

Creating, Describing, and Implementing Designs for Larger-Scale Software Projects

version 2.2

Stephen Davies, Ph.D.
University of Mary Washington

Blueprints:
Creating, Describing, and Implementing Designs for Larger-scale Software Projects

version 2.2

Stephen Davies, Ph.D.
Computer Science Department
University of Mary Washington

University of Mary Washington
Department of Computer Science
Trinkle Hall
1301 College Avenue
Fredericksburg, VA 22401

The LaTeXsource for this book is available from: `https://github.com/rockladyeagles/blueprints`.

Contents

Chapter 1

Getting off the ground

Before we begin our study of object-oriented systems proper, we'll introduce the command-line toolset we'll be using to construct our programs. We'll take each of the most important tools out of our toolbox, lay them out before us on a little mat, and learn what they're for.

1.1 Why the command line?

Developing software in a command-line environment (sometimes abbreviated "CLI" for **command-line interface**, as opposed to a "GUI[1]" or **graphical user interface**) involves typing white text in little black boxes. It requires memorizing and regurgitating a variety of obscure commands. It demands exact adherence to an inconsistent syntax, and exacts heavy penalties for mistakes, all while providing only a very crude and clunky-looking interface.

It's natural to wonder why we would want to do this. After all, aren't computer systems immeasurably more sophisticated now? If even end users run fancy, graphical, forgiving apps, shouldn't computer scientists expect even easier-to-use and sexier-looking stuff?

It may seem so, and in terms of the *power* the tools provide, we'll discover that indeed software developers are aptly equipped. But in

[1]Commonly pronounced "gooey."

some ways it's a false expectation to assume that our toolset would be as *easy* to operate as that of an everyday user. After all, which is easier: to drive a car, or to be a mechanic? Even though I enjoy cruise control and auto-adjusting seats, I don't find it strange at all to learn that mechanics still use socket wrenches to adjust piston assemblies.

Much of what makes a CLI so powerful is its expressiveness. A driver can press any of the three or four cruise control functions the manufacturer provided. But a mechanic can take any of hundreds of tools, tweak dozens of different parts, and combine these adjustments in uncountable ways. That's the kind of flexibility the command line provides.

The difference between a CLI and a GUI is that with the latter, the user can essentially do *only what the tool designer anticipated she would want to do.* There's no way she can express something that isn't one of the tailor-made menu options.

When you use the command line, think of it as composing sentences, word by word. A GUI comes with a repertoire of standard sentences you can choose from. That makes it easy to do standard things, and hard to make silly mistakes. But a CLI, being inherently language-based, is immeasurably more flexible. You can write any (legal, grammatically correct) sentence you choose, even one the designers of the CLI never thought of, and even one that you didn't know you'd want to type until a moment ago. The bits and pieces can be combined in a myriad of ways, just as nouns and verbs can.

There are other reasons as well that many developers live on the command line. Among them are[2]:

- **Speed.** It turns out to be way, way faster to type commands – in combination with the various shortcuts and recall/edit operations – than it is to sift through menu options and such with a mouse. Trust me.
- **Remote access.** When you're running programs on your own device, it's possible to do it with a GUI. But computer

[2]Thanks to Ian Finlayson for capturing much of this list.

scientists very often have to connect over the network to distant machines in order to tell them what to do. Every time you need to configure a web server, for instance, or update a publicly-accessible database, or run a time-consuming job on a parallel cluster, or correct the data on your mobile device, you need a way to issue commands to another machine through a very low-bandwidth channel. Opening a command line "shell" to that remote device is by far the most common and effective way to do this.

- **Scriptability.** There's just no good way to automate a sequence of GUI operations. To explain to someone else how to accomplish something, you have to painfully walk them through each operation ("go to the Start menu and find Accessories, then in the Math menu choose Calculator...when it comes up, right-click in the background and enable Advanced Options...") which is tedious and error-prone. It'd be nice if you could just send them a custom command which would do all that. As a matter of fact, it would be nice if *you* could make a custom command which would do all that, so that you could execute it many times without rehashing the same rigmarole. You'll find that CLIs are eminently automatable in this way. You can create custom commands called "scripts" that are combinations of other interacting commands, and in this way you become master of your whole world.

- **Consistency.** Graphical user interfaces are more different from each other than CLIs are. Partly this is because nearly any CLI you're likely to use is Unix/Linux-based[3], and hence they all "speak the same language." It's great to be able to log on to different laptops, web servers, your phone, your Kindle, or a Raspberry Pi and get the same prompt that understands the same stuff.

[3]For our purposes, you can consider the terms "Unix" and "Linux" exact synonyms. The Mac OS X command line (available through the "Terminal" app) is Unix-based, too. Windows machines aren't, but programs like "Cygwin" can be downloaded for free and provide a Linux-like command-line veneer over the operating system.

- **Stability.** CLIs rarely change. When they do, it's very very rarely in a non-backwards-compatible way. By contrast, every time a new graphical user interface is released, you have to go through a period of hunting around and finding out where everything is. With Unix/Linux, you can literally run commands that were written last century and they will likely still work as is.

There's always a few students who, despite the above benefits, resist learning this material at first. I get it. It's like learning a new language, and the immense effort to understand an alien world sure doesn't feel like it's going to pay off any time soon. All I can say is that if you're not convinced it's worth it, for now just think of it as something you have to master "just because your professor and the industry says so." My hope is that by the end of this course, you're pleasantly surprised by seeing some payoff for your hard work.

1.2 The filesystem

Okay. The backdrop for all our use of the Linux command-line interface is the **filesystem**.[4] Any general-purpose computer, no matter its architecture or OS, has an area of permanent storage for user data. Interestingly, and conveniently, all computers organize their filesystems in pretty much the same way: as a **tree** of **files** and **directories**. (Windows/Mac users will be familiar with the term "**folder**," which *means exactly the same thing as "directory."*) In what follows, we'll be using a different syntax (text instead of visual icons) to work with what is conceptually the same organizational structure you're used to on your own computer.

Files and directories

A file is simply any named chunk of stuff on your disk. Images, .mp3 tunes, Word docs, and (importantly) plain text files are all in this category. On Windows, you're used to each of these files having a filesystem "extension" designating its type: ".doc" means

[4]Often, not always, written as a single word as I have it here.

a Word doc, and ".jpg" means an image file, for example. This is sort of true with Linux, although the rules are a bit looser. Not all files have extensions at all, and when they do, it's more a signal that they're intended to be treated a certain way than it is a hard-and-fast requirement.

The most important files you'll work with in this class will have a `.java` extension. These are your Java **source files**. You'll also work with other various supporting files to make all the tools work correctly. It's important to realize that *a file is fundamentally just some data, which can theoretically be opened and dealt with by any program.* When we say that HamletPaper.doc "*is*" a Word doc, what we really mean is that its data is formatted in a certain way that the Microsoft Word application expects to see, so it can render it on the screen for editing. But it is possible to open that same HamletPaper.doc file with other programs and manipulate its contents. This may seem sketchy, but it is actually a force for good.

In particular, you'll be tempted this semester to think of a `.java` file as "a `vim` file," in the same way that you may think of an `.xls` file as "an Excel file." I hope to break you of this habit, as you learn to see a file as text or data that is actually independent of what kind of program might be used to open and manipulate it.

A directory is a container for files *and also other directories.* That last italicized phrase is what gives rise to the overall tree structure of the filesystem, as discussed in the following section.

By the way, every file and directory *in a particular directory* must have a unique name. You can't have a file called "DireStraits.mp3" and another one also called "DireStraits.mp3" sitting there in the same folder: it's a name collision. However, it's perfectly permissible to have two files with the same name in different directories. This is kind of like how there isn't more than one "Stephen" in my immediate family (that would be confusing[5]), but there are of course many "Stephens" in the world.

[5]With apologies to boxing legend George Foreman, who named all four of his children "George." That practice is not filesystem-compatible.

The filesystem tree

The files and directories in a filesystem form a nested, hierarchical structure called a **tree** (see Fig. 1.1). I have drawn two kinds of nodes in this tree: directories (yellow ovals) and files (blue boxes). As expected, some of the directories have arrows coming out of them, but none of the files do. The elements that a directory is pointing to are the contents it contains: *e.g.*, the left-most "`america`" directory contains another directory ("`nation`") and also the file `A.txt`. We use the term **parent directory** to mean the directory immediately above an entry in the filesystem; the left-most `A.txt` file's parent directory is the `america` directory we just spoke of.

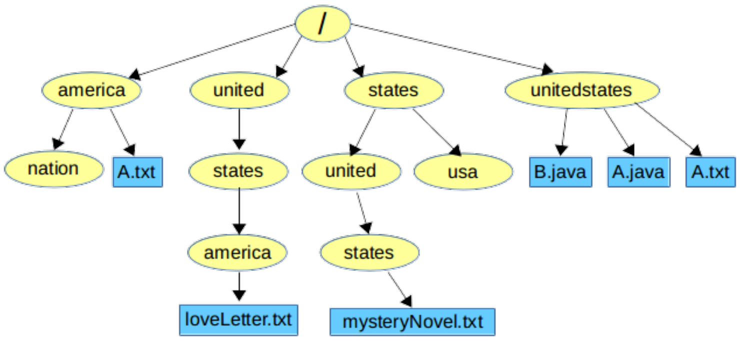


Figure 1.1: The Linux filesystem, in pictorial form.

In order to keep you on your toes, I've given several entries in this example filesystem the same name: in addition to a couple different `america`s, we've got several `states`, multiple different `A.txt`s, etc. In no case, however, are the duplicately-named entries in the same directory. (Convince yourself of that fact.)

Only one surprise

So far this is pretty easy. And it won't get much harder. But here's the one thing you have to get used to: with a CLI, *we won't ever actually see that filesystem picture visually.* It's there, but we don't explicitly view it in graphical form. Instead, there will be a textual way of referring to every file and directory. It's straightforward, but can be a bit of a shock to those coming from point-and-click systems like Windows.

The "current" (or "working") directory

One vital concept to grasp is that every time we issue a command or run a program in Linux, we are doing so *within the context of a particular directory.* Conceptually, we think of being "in" a certain directory at any point in time. We call this directory "the **current directory**" or "the **working directory**", and we'll learn commands to find out what it is and to change it to something else.

Which one we're "in" has a crucial impact on what happens when we execute a command. For instance, if our current directory is the far-left `america` directory, and we issue a command that does something to "`A.txt`", it would act on the left-most `A.txt` file, since it's the one within the current directory. But if our current directory were `unitedstates`, "`A.txt`" would instead mean the far-right blue node.

I've found that failure to understand the "current directory" concept is one of the most common trouble spots for beginning Linux programmers.

The root directory

Okay, back to the filesystem as a whole. At the top of the tree is the **root directory**, which has no parents. (This is often disorienting to non-computer-scientists, since in the real world you may have noticed that trees actually grow *up*, not down. But in computer science, we always draw trees growing down from the root.)

The root directory is the anchor point of the entire filesystem: it ultimately contains everything under it. It also has a very strange name: "`/`", pronounced "slash." (This is a "forward slash," by the way, to the left of your right-most Shift key, not a "backslash." Oddly, most Windows systems use a backslash "\" for this instead.) Stay awake, because this "`/`" character will shortly mean something very different as well.

Paths

It should be apparent to you that as a consequence of this nested tree structure, you can "reach" every element from the root directory by traversing from arrow to arrow. Furthermore, you can do so in only one way. For instance, the `B.java` file can be reached from the root by going from "/" to `unitedstates` to `B.java`. And that's the *only* way to get there. You can reach `loveLetter.txt` by going from "/" through `united`, `states`, and `america`, in that order. This is true for every file and directory.

What this means is that every entry has a unique **path**, and we can express it in text as well as in a diagram. Take the `B.java` file for example. Its path is:

```
/unitedstates/B.java
```

Look very carefully at that string as we dissect it. The most important thing to grasp is that the two slash ("/") characters *each mean something different.* The first one means "the root directory, which is called slash." But the second one is merely a separator, delimiting the `unitedstates` from the `B.java`. So this path means "start at the *root* directory, go down to its `unitedstates` entry (which itself is a directory), and there you have the `B.java` file."

Similarly, the path to `loveLetter.txt` is:

```
/united/states/america/loveLetter.txt
```

(Note that the slash between `united` and `states` makes all the difference in the world: if it weren't there, we'd be starting our descent through the right-most `unitedstates` directory as before.)

These paths are called **absolute paths** because they *start with a slash.* This means that they give the complete, start-from-the-top position of a particular file or directory. It's kind of like referring to a building by its complete address, including city, state, zip code, country, and planet. Often we want a short-hand way of referring to an entry without specifying its entire absolute path. To do so, we use a **relative path**.

A relative path is relative *to the current directory.* And it does ***not*** begin with a slash. Instead, it gives directory names, separated by slashes, indicating where to start descending from the *current* directory.

For example, let's say the current directory was "`/states`". And suppose I used this relative path:

```
united/states/mysteryNovel.txt
```

(Note carefully that it has *no* initial slash!) This relative path would start at the *current* directory (`/states`) and from there traverse down to `united`, `states`, and then finally `mysteryNovel.txt`. Obviously, where you end up is critically dependent on where you start – on what the current directory is.

To test your understanding, realize that in this case where the current directory is `/states`, there is no such file called `united/states/america/loveLetter.txt`. In fact, even `united/states/america` doesn't exist. However, if we changed the current directory to be the root ("`/`"), suddenly the relative path `united/states/america/loveLetter.txt` would be legit.

1.3 Linux A-B-C's

With the filesystem always hovering in the background, let's introduce the first basic Linux commands to work with the files and directories. These commands are so basic that they're like the alphabet of speaking any Linux sentence. Using them should eventually be as familiar and effortless to you as clicking the mouse.

In all that follows, I will precede anything you are to type on the Linux command line with a dollar sign **prompt**:

```
$
```

To execute a command, you do *not* type the prompt itself: it's just there to indicate "now is an appropriate time and place to enter a Linux command." Just type the stuff after it.

Also, depending on your system and configuration, your prompt may look different or have other information in it. One common setting, for instance, is for the current directory to always appear as the prompt. (I personally hate that, since it makes different commands start at different horizontal locations as I work, plus it consumes a lot of space.) No matter what, though, just mentally substitute "the dollar sign" for "whatever your Linux prompt is."

1. `pwd`

   Your first command stands for "**p**rint **w**orking **d**irectory," and simply tells you what the current directory is at any point in time. For example:

   ```
   $ pwd
   /united/states
   ```

   tells you that you're currently "in" the `states` directory, which is contained within the `united` directory, which is contained within the root directory.

   Tip: get in the habit of typing `pwd` a *lot*, especially at first. Get ingrained in your brain the question "where am I in the filesystem right now?" because it matters, yet is not in your face except when you type this command.

---

2. `cd`

   `cd` stands for "**c**hange **d**irectory" and is how you move to another place. You give it an **argument** (kind of like passing a parameter to a function call in a programming language, although we don't use parentheses or commas here) which is where you want to go:

   ```
   $ cd  /america/nation
   ```

   Here, I've specified an absolute path. If I now execute `pwd`, I see that it worked:

   ```
   $ pwd
   /america/nation
   ```

More common is to specify a relative path. If we first go back to our original location:

```
$ cd  /united/states
$ pwd
/united/states
```

we can then say "go *from here* into the `america` directory":

```
$ cd  america
$ pwd
/united/states/america
```

I can't overestimate how important it is to notice that in the previous `cd` command, I did *not* include a slash before `america`. If I had, it would have been an absolute path, and I would have gone to a completely different part of the filesystem:

```
$ cd  /america
$ pwd
/america
```

"Special" directory shortcuts

This is a good time to mention that when you are specifying paths, there are three very common shortcuts that you'll want to know about.

- The current directory: `.`

  A plain-ol' dot (period) is used to mean "the current directory." There's no obvious uses for this yet, but believe me, it comes up *all* the time, so just memorize it. A useless example for now:

  ```
  $ pwd
  /united/states
  $ cd  ./america
  $ pwd
  /united/states/america
  ```

So "`./america`" is another way of saying "`america`". (Told you this example was useless.)

- The parent directory: `..`

  More immediately useful is the double-dot, which means "the parent of the current directory." If we're currently in `/united/states` and want to go to `/united`, one way to do it is:

  ```
  $ cd ..
  $ pwd
  /united
  ```

  We can also join this with additional relative path stuff to move around the hierarchy in various ways:

  ```
  $ pwd
  /states/united
  $ cd ../usa
  $ pwd
  /states/usa
  ```

  Here we went to a "sibling" directory by "going up one, and then down to a different child."

- The home directory: `~`

  A shortcut for "the home directory" (which means "the current directory when you first log in") is a tilde. It's commonly used in conjunction with other relative path stuff, like the last double-dot example, above.

  Your home directory will probably be something like `/home/joeschmo` (which you can verify by just typing `pwd` when you first log in). Suppose it is. Then, you can use the tilde:

  ```
  $ pwd
  /somewhere/else/in/the/filesystem
  $ cd ~/shortStories/scifi
  $ pwd
  /home/joeschmo/shortStories/scifi
  ```

  to go to any of your subdirectories.

---

3. `ls`

   While `pwd` tells you what the current directory is, the `ls` command (which sort of stands for "`l`i`s`t") gives you its contents. If I type it while in the `/america` directory, for instance, it tells me:

   ```
   $ ls
   nation   A.txt
   ```

   These are the two entries from Figure 1.1.

   A few gotchas to be aware of. First, there's no way from that listing to tell that `nation` is a directory whereas `A.txt` is a file. If you want to see that, you need to add the "`-l`" option (a minus sign followed by the lower-case letter "ell"):

   ```
   $ ls  -l
   -rw-r--r-- 1 kyloren   sithlords   17 Sep  5 16:21 A.txt
   drwxr-xr-x 2 kyloren   sithlords 4096 Sep  5 16:21 nation
   ```

   Lots of clutter here. The key points:

   - The far-left character on each line is either a "`-`" or a "`d`", indicating file or directory.
   - Files in Linux have "owners," meaning specific users who created them and have permissions to manage them. Both of these entries are evidently owned by user `kyloren`.
   - The `17` and `4096` are file sizes (in bytes).
   - You can see the date and time each entry was last modified.

   The "`-l`" stands for "`l`ong file listing." Most Linux commands have a bevy of different options you can specify when you execute them, most often beginning with a minus sign.

   Another important one for the `ls` command is "`-a`" which stands for "`a`ll files, please." If that sounds like a strange option, that's because it is. It turns out that `ls` by default doesn't show you all the files; in particular, *it omits those* *whose names start with a dot (.).* Why? There are reasons. The only time this will be relevant to you soon is if you want to work with your `.bashrc` file.[6] You'd have to type "`ls -a`" in your home directory to actually see it in the listing.

The above three commands – `pwd`, `cd`, and `ls`, go together like Luke, Han, and Leia. Get in the habit of using them literally every minute you're working on the Linux command line.

---

4. `mkdir`

   To create a directory in the first place, use the `mkdir` command and give it the name:

   ```
   $ mkdir  evilplans
   ```

   This new `evilplans` directory will be created inside the current directory.

   Note carefully that *making a directory does not automatically put you in it!* Lots of beginners mistakenly think this will happen, but you can see that it does not:

   ```
   $ pwd
   /home/kyloren
   $ mkdir  evilplans
   $ pwd
   /home/kyloren
   ```

   You have to `cd` as a separate step if you want to now be *in* `evilplans`:

[6] If, in your *home* directory, you create a file with `vim` called literally `.bashrc` (pronounced "dot-bash-arr-see") then whatever Linux commands it contains will be executed automatically every time you log in. Once you get proficient with Linux, it's very handy to put shortcuts, aliases, and various preferences in it.

```
$ cd  evilplans
$ pwd
/home/kyloren/evilplans
```

A useful option to `mkdir` is the "`-p`" option which means "make all **p**arent directories as necessary." This lets us create a deeply-nested structure all in one fell swoop:

```
$ mkdir  -p  find/luke/skywalker/now
$ cd  find/luke/skywalker/now
$ pwd
/home/kyloren/find/luke/skywalker/now
```

---

5. `cp`

   To make a copy of a file, use `cp` and give it *two* arguments, a source and a destination. If I type:

   ```
   $ cp  A.txt  Q.txt
   ```

   I will now have two exact copies of the file which can be independently modified:

   ```
   $ ls
   nation   A.txt   Q.txt
   ```

   I can also use this to make a (same-named) copy of a file to a different location, by providing a directory as the second argument:

   ```
   $ cp  A.txt  /states/usa
   $ cd  /states/usa
   $ ls
   A.txt
   ```

---

6. `mv`

   `mv` has pretty much the same effect as `cp`, except that it does not retain the original copy. This command can be used to rename a file ("`$ mv oldfilename newfilename`") as well as to change a file's location.

---

7. `vim` (and `vimtutor`)

   It's really ludicrous to include this command in amongst all the others, when its ins-and-outs could (and do) occupy entire textbooks in their own right. `vim` is a text editor program with a zillion amazing features which you will use this semester to write your programs. The normal way of creating a file, in fact, will be this:

   ```
   $ vim  notesOnTheResistance.txt
   ```

   or this:

   ```
   $ vim  DestroyGalacticRepublic.java
   ```

   after which you will do loooooooooooots of other stuff way beyond the scope of this book. That stuff will be cryptic and agonizing at first, but will eventually become second-nature and give you the tremendous text editing power you need to be a truly efficient software developer. It's kind of like learning to use the Force for the first time.

   For now, I'll make this (strong) suggestion: to learn `vim` for the first time, type this command (all one word) at the command line:

   ```
   $ vimtutor
   ```

   Grab a Coke, and spend 30-40 minutes patiently reading and following the instructions. This tutorial is quite good, and will teach you the very basics of getting a file created and edited with this incredible tool.

---

8. `git`

   `git` is another one that doesn't really fit in this list, since it's much more than just "a command." For now, though, all you need to understand is that it's a **version control system** that allows you to track and manage the changes you make to your software over time.

   Up until now, you've been dealing with a paradigm like "the IDE always has the most recent copy of my code, and that's the only version of it that exists." You'll need much more flexibility than that when you work on large systems.

   Here's all you need to know at present, though:

   - The command "`git init .`" (don't forget the dot at the end, after a space) creates a git **repository** (or "**repo**") in the current directory. That just means that your current directory, and everything under it, are now "under git's management."
   - You use "`git add`" to make git aware of one or more files that you want it to track from that point forward. You'll type "`git add` file1 file2 file3" or however many files you want to add at that point. Ordinarily you'll want to `git add` all of your `.java` files.
   - When you've made a significant change to one or more of your files that you want git to be aware of, you'll enter this command:

     ```
     $ git commit -a -m "A message describing the change."
     ```

     Each such change is called a **commit**. Think of it as taking a snapshot of your code that you can return to later.
   - "`git status`" and "`git log`" are two useful commands that show the current state of your files as git sees them, and a history of all the different commits you've made. Type them occasionally just to get a feel for what kind of information they show.

   We'll talk much more about `git` later. For now, just know that it exists, and type the above commands verbatim when prompted.

---

9. `javac` and `java`

   Now, finally to some programming stuff. On your Linux system, the Java **compiler** (*i.e.*, the program that converts your source code into the form the computer needs to run it) is called `javac`, and the **virtual machine** (the interpreter that runs your compiled code) is called `java`. Both of these are part of the **JDK**, or "Java Development Kit," that you install in order to program in Java.[7]

   To compile, you give `javac` all the Java files that are part of your program:

   ```
   $ javac  DestroyGalacticRepublic.java  Bombs.java  SinisterPlans.java
   ```

   which will either produce a `.class` file for each `.java` file, or compiler errors for you to read. Finally, to run it, you give `java` the name of *the class that contains your `main()` method*:

   ```
   $ java  DestroyGalacticRepublic
   ```

   (Notice we don't include "`.java`" or "`.class`" here, and notice we don't mention every Java class, only the one that has the `main()`.)

[7]Just to confuse you, the JDK has sometimes been called the "Java SDK" ("Java Software Development Kit") and the "J2SE" ("Java 2 Standard Edition") in the past, and you'll likely run across those acronyms as well. To confuse you even more, the software you need to simply *run* a Java program (as opposed to writing your own) is called the "JRE" – Java Runtime Environment. Finally, to confuse you yet further, Java version numbers were originally all "one-dot-something" (like "Java 1.3") but in 2004 they ditched the "one-dot" and started naming the versions after the second number alone. (So, the successor to "Java 1.4" was "Java 5.") This book assumes you're on Java 8, by the way.

1.4 The quickest path through the woods

Whew. That was a lot. It's kind of like moving to another country: every little thing, all at once, seems different.

All I can do is promise you it will get easier as you get used to that new country. And there will be parts of it you will like – maybe you'll even like it better than the point-and-click country you grew up in.

In the meantime, let's pull together all the steps to get a "Hello World" Java program running on the Linux command line.

1. Log on to your Linux system (for instance, your Google Cloud instance), however you do that.
2. Create a directory to hold your project:

   ```
   $ mkdir  myFirstProgram
   ```

3. And make sure to actually go there:

   ```
   $ cd  myFirstProgram
   ```

4. Create a git repo to manage this project:

   ```
   $ git  init  .
   ```

   (and of course don't forget that pesky dot at the end.)
5. Now create a Java file:

   ```
   $ vim  HelloWorld.java
   ```

   (You are now in `vim`. Everything you learned during your `vimtutor` session, and everything you can get from a zillion different "vim cheat sheets" on the Internet, is relevant now. Good luck.)
6. Give it these contents:

   ```
   class HelloWorld {
       public static void main(String args[]) {
           System.out.println("yo sup dawg");
       }
   }
   ```

7. Save your file and exit `vim`.
8. Now compile it:

   ```
   $ javac  HelloWorld.java
   ```

9. And, since it gave you no errors, run it:

   ```
   $ java  HelloWorld
   ```

It's a big bright world ahead of us. Go take a break and I'll see you next chapter.

Chapter 2

The "software crisis" and encapsulation

This book is going to dive deeply into a huge pile of nuts and bolts. But before we take the leap into particulars, it's important to stand briefly at the precipice and understand why we're jumping. What does "object-oriented" mean? What problem was it intended to solve? When was it invented and why?

2.1 Ancient history

A long time ago, in our own galaxy, a situation emerged which has been labeled **the software crisis**. This crisis didn't happen at an instant in time; it was a set of disagreeable circumstances which gradually evolved until it became unbearable. The crisis is usually dated somewhere in the 1970's. This was just as the high-tech computing industry was really starting to heat up, on its way to permanently changing the lives of almost every person on the planet.

Now "crisis" is an alarming word, designed to get your attention. It's worth asking what all the hubbub was about. The immediate symptom may not strike you as a three-alarm fire: it was simply that software projects were tending to overrun their schedules.

The '70's were not a very plug-and-play era, since standards had not yet evolved to facilitate intercompatibilities between devices or programs. So the focus was often on building complete systems from the ground up. Engineering teams would plan releases of key product lines that involved numerous components, such as system architecture, hardware design and integration, data collection and organization, system and network configuration, and software development at both the operating system and the end user levels.

What managers discovered was that the *software* components of projects were consistently coming in late and over-budget. Sometimes, they didn't get finished at all. When they did, they were buggy and brittle. And they were especially vulnerable to requirements changes: if circumstances were discovered during the project that required a change in the way the software needed to work, the software team was often strikingly unable to adapt to this. They could be set back weeks or months to implement even a modest change.

This astonished everyone at the time. After all, "*soft*ware" – a pun on "hardware" – was a term intended to convey the flexible, malleable nature of computer programs as contrasted with physical devices. Software was supposed to be easy to write and easy to change. That was the point. You didn't need complex manufacturing processes: you needed a computer and a text editor. And you (seemingly) didn't face challenges of scale the way you did with hardware: you might run out of room to put logic circuits on a chip or a motherboard, but there was no limit to the size of a text file.

So building complex stuff quickly, and turning on a dime when necessary, ought to be easy to do in software. Right?

Quantifying the crisis

I've never seen any hard data quantifying the budget overruns and delays that software projects faced in the 1970's, but it's possible to sketch it conceptually. Take a look at Figure 2.1. This is my attempt to show the main dynamic at work.

On the x-axis is some measure of the *complexity* of a proposed computer program. Now complexity is devilishly difficult to quantify –

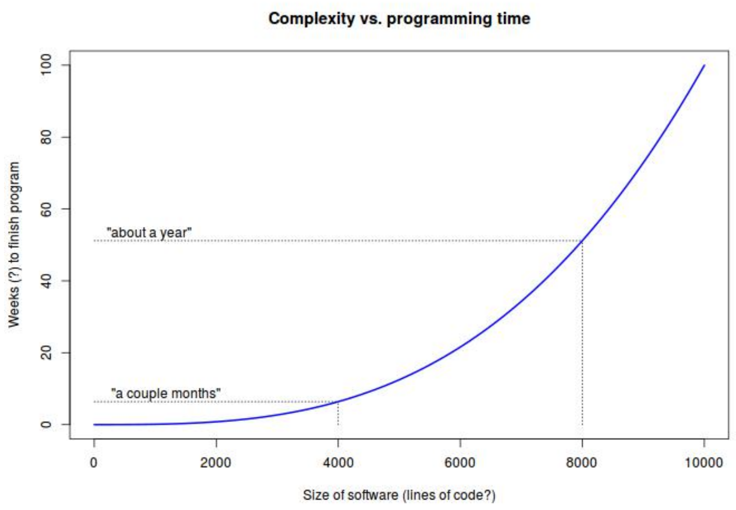


Figure 2.1: The software crisis quantified: how long it took to complete a program of various sizes. (Conceptual.)

League of Legends is more complex than Angry Birds, but by how much? Twice as complex? Ten times? A hundred times? I'll have a better answer to that in a few paragraphs, but for now, as a proxy we'll just use the *size* of the program, measured in **lines of code**[1] ("LOC" or "KLOC."[2])

On the y-axis is the corresponding amount of time it would take a programming team of a certain size to design, build (code), debug, and test the program.[3] As with the x-axis, we're making all kinds of simplifying assumptions here: we're not worrying about exactly

[1]I know what you're thinking, and you're right. Not all lines of code are equally complex. Some of them are just variable assignments, whereas others are parts of complicated loops or function calls. Heck, some are just comments. Heck, some are actually *blank*. While all true, this analysis is conceptual anyway, and we can surely say that raw program length is at least *somewhat* related to complexity. Show me a real-life ten-line program that's actually more complex than a real-life ten-thousand-line program and I'll change my mind.

[2]"Lines of code" is sometimes abbreviated "LOC." Even more common is the abbreviation "KLOC" for "thousands of lines of code."

[3]Note carefully that this has nothing to do with how long the code takes to *run*. We're talking about programmer-time here, not CPU-time.

how many developers there are, how much experience they each have, what language they're writing in, *etc.* That's okay. The point of this exercise is simply to recognize the nature of the curve, showing how these two fuzzy variables were related in the '70's.

And a daunting curve it is, too. And very counterintuitive to project managers. One would assume that if a 4,000-line program took the development team a couple of months to release, an 8,000-line program would take about twice that long. After all, it's twice as many lines, right?

The reality was not even close. A program with twice as many lines could easily take *four* times as long to build...or six, or ten, or twenty times. Worse, the programs that were built were also very hard to *change.* Take a large enough program and try to add a feature, fix a bug, or support a new data format, and you inevitably broke something else while making the change. Then you fixed what you broke, but *d'oh!!* broke something else by doing so, *etc.*

It was miserable, especially because advances in other areas (like hardware) were making exciting technologies possible for the first time. Everyone was rarin' to go, yet unexpectedly the software (supposedly the easy part) was gumming up the works.

For a time, it almost seemed as if the human race had uncovered some built-in limitation of the universe, like the speed of light. This hypothetical constant might have been called “maximum complexity,” meaning the greatest amount of sophistication one could build in to a single logical creation. That curve in Figure 2.1 starts to go up fast. Maybe, people depressingly thought, a functioning 200,000-line program isn't even *possible* to create? That threatened to put a damper on a lot of expectations.

2.2 Software and complexity

Let's consider a different way to measure a program's complexity than simply counting the lines of code. Instead, let's quantify its number of *dependencies.*

A **dependency** between two chunks of software (be they individual lines of code, constructs like loops or if/else chains, functions, or something even bigger) means that *if one of them changes, the other may possibly be affected.*

For instance, suppose that in "code chunk A," I define a function `compute_sales_tax()` to take one argument: the price of an item. It will return the sales tax on that item as a simple percentage. Now, suppose that in "code chunk B," I call the method as follows:

```
// In code chunk B...
double item_price = 24.99;
double total_price = item_price + compute_sales_tax(item_price);
```

We say that code chunk B has a dependency on A. If we were to change A to require a second parameter (perhaps the state the customer lives in, since different states have different laws about whether and how to collect sales tax), that's great and all, *but B immediately breaks* unless we change it as well.

This example is a syntactic dependency: the compiler will fail when trying to compile code chunk B because its parameter list is wrong (*i.e.*, doesn't match A's). In general, though, not all dependencies are merely syntactic. There are also *logical* dependencies, in which one chunk of code depends on another's functionality working a certain way.

For example, suppose that we change `compute_sales_tax()` in a different way: instead of returning the sales tax, we make it return the cost of the item *plus* the sales tax. If our tax rate is 5%, then the original version of `compute_sales_tax(24.99)` would return 1.25 (the sales tax on the item), but our new version would return 26.24 (the item's price with its sales tax added in).

This may seem like a good change, since it prevents code like that in chunk B from having to add the item's price back in. However, if we don't change B in tandem with A, *B breaks again.* It's not a compilation error this time, but a logic error: B's `total_price` variable is now going to contain 51.23 because we didn't keep the two chunks of code in sync.

Dependencies == complexity

Now why do I bring all this up? Because it turns out that the *length* of a program was not the cause of the software crisis. Instead, it was *the number of dependencies* programs had. That turns out to be a different, and more salient, measure of a program’s complexity.

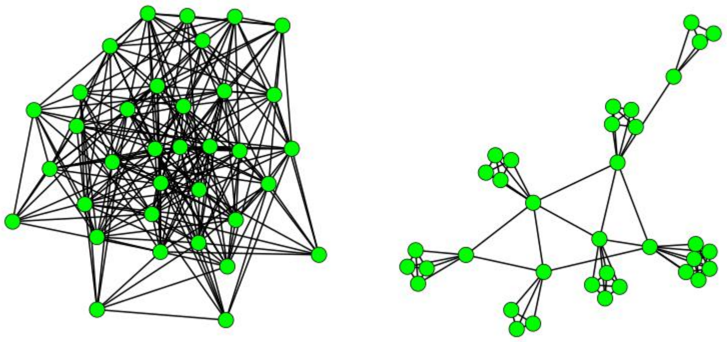


Figure 2.2: Two programs, each with 35 code chunks. On the left, there are 230 dependencies. On the right, there are only 83. The right program is properly **encapsulated** and **modular**, whereas the left one is neither.

Consider the two graphs[4] in Figure 2.2. Imagine that each green circle represents one chunk of code. Each line between circles represents a dependency: the two chunks of code it connects rely on each other *not* to change, because if one of them changes, the other one might break.

The kind of program illustrated by the left-hand graph sometimes goes by the name **spaghetti code**. You can see why just by looking at it. Essentially, every line of code potentially depends on everything else.

By contrast, the right-hand graph depicts a **modular** program. It has the same amount of functionality – 35 green circles in each case – but far fewer dependencies between them (only about a third as many). Looking further, you can see how most of the circles are

[4]A **graph** in computer science terms is a data structure that consists of **vertices** (or **nodes**) and **edges** (or **links**) connecting them. They’re often drawn with circles and lines, as Figure 2.2 does.

"hiding" behind a gatekeeper circle that connects to the main group. The majority of circles are shielded from the morass of dependencies by living in an isolated world, and only communicating with the rest of the program through their gatekeeper.

Now look back at the left-hand graph. Choose one of the circles at random, and imagine that it represents a chunk of code you need to change (maybe there's a bug in it you have to fix, or you need to extend it in some way). Think about the repercussions of that task. Fixing the green circle is a job in itself, but once you've done so, how can you be sure you didn't break something else? There might be twenty other chunks of code that depend on the first one staying the way it was in order to work correctly. Just identifying all of them is an enormous task, to say nothing of verifying that they all still work, and fixing the ones that don't. And oh, by the way: if you *do* end up having to fix another green circle because your first change caused a ripple effect...that second change is going to cause the exact same problem.

The situation is obviously much easier with the right-hand program. Again, choose a circle at random, and then ask yourself how onerous it is to change it. If you choose one of the many circles that are "hiding" behind their gatekeeper, the possible damage is minuscule: only three or four other circles might be affected. If you have to change a gatekeeper, the news is worse, but still far better than it was with the left-hand program. Just count how many dependencies there are for even the most densely connected circle of the modular program – there ain't many.

Figure 2.3 quantifies this further, and in fact finally gives us the insight we need to understand the cause of the curve in Figure 2.1 (on p. 23). If you've taken a Discrete Math class, you may remember that the number of possible edges in a graph goes up as the *square* of the number of vertices. (Specifically, a graph with n vertices can have up to $\frac{n(n-1)}{2}$, or about $\frac{n^2}{2}$, edges.) So doubling the number of code chunks approximately *quadruples* the number of possible dependencies in the program. That explains why Figure 2.1 was **superlinear** (*i.e.*, increased faster than a straight line would have), and why everyone in the '70's was underestimating how much time

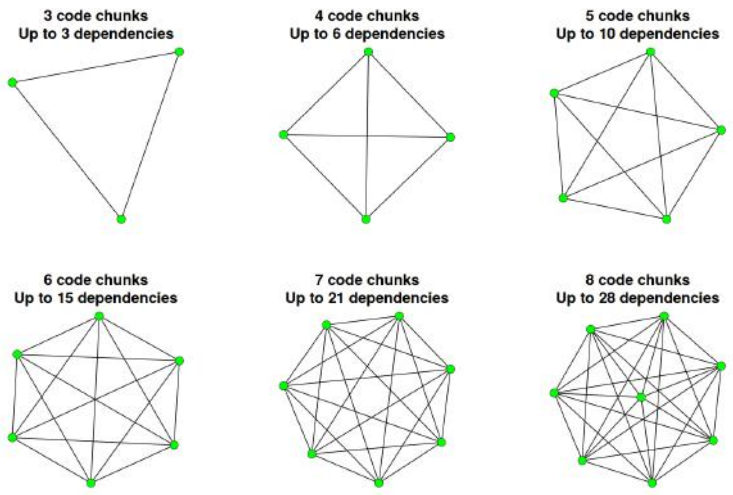


Figure 2.3: The maximum possible number of dependencies for programs of different sizes.

it would take to write and maintain large programs.

2.3 Encapsulation

This, then, was the root cause of the software crisis. Larger programs, which had more parts, inevitably produced too many interlocking dependencies between their parts. Those dependencies were a killer: changing or adding any one part threatened to break a dozen other parts. So a program with twice as many features didn't take twice as long to construct; it took way more than twice as long. And a program with ten times as many features looked plumb out of reach.

And now finally, the punchline. The way the human race conquered the dependency problem, and overcame the software crisis for good, was by means of the single most important aspect of object-oriented programming: **encapsulation**. This feature gets far, far less press than it should. It made possible all the complex software applications that the world now relies on every day. It's one of the most important principles – perhaps *the* most important – in all of computer science.

So what is encapsulation? In a word, it's an organizational principle that permits many more green circles to be added to a program without also having to add a zillion more pesky lines. It's a way to keep a program's dependencies under control, so that as it grows larger, it doesn't also grow more brittle and bug-prone.

Glance back at Figure 2.2 on p. 26. Simply put, the right-hand program is employing encapsulation, and the left-hand program is not. On the right, each of the little clusters of tightly-knit green circles is "encapsulated" from the other clusters. That isolates them safely behind a gatekeeper such that changing them will *not* trigger a chain reaction and require other changes.

In all the code we write, we want to strive for this. We want to make our units of code highly **cohesive** yet loosely **coupled**. We want a lot of small components (not a few large ones), and we want each component's internal workings to be invisible to the outside world. It's the only way to avoid the hell of spaghetti code.

OOP's encapsulation solution: the `class`

I've been using vague terms like "chunks" and "units" and "components" to refer to these bits of interacting software. The key innovation of OOP (object-oriented programming) was a particular kind of "unit," structured in a particular way: the **class**. It changed the world.

We'll be talking lots and lots about classes throughout this book. Every single line of code we write, in fact, will be part of a class. For now, I want you to imagine a class as being comprised of two parts: a **public interface** and a **private implementation**. In terms of Figure 2.2, the public interface is the gatekeeper node that connects each cluster to the whole, whereas the private implementation is the other nodes in the cluster that hide behind the gatekeeper.

Another way of viewing this is Figure 2.4. The concentric yellow circles represent a **class**, with its two components. To evoke a biology metaphor, you can think of the public interface as the membrane of a cell: nothing goes in or out of the cell body except through the membrane.

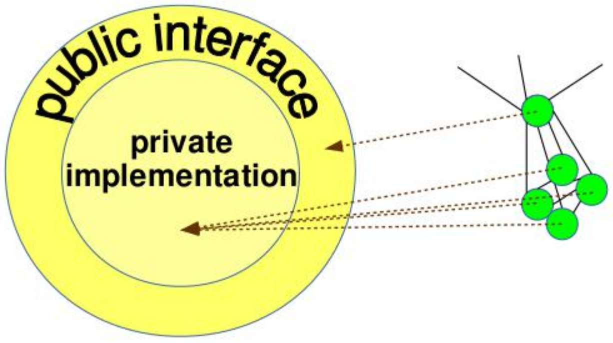


Figure 2.4: Encapsulation, visualized abstractly.

Almost all the code for the class, including the variables it uses and the bodies of its functions, are in the *inner* circle, safely sequestered away from the membrane. This means they are *free to change* without impacting any other part of the code that's using the class. The only things in the outer circle are the function **signatures** (*i.e.*, the names, return types, and argument lists of the functions). This is the information other parts of the code must know in order to make use of the class.

The right-side of the diagram is my way of drawing a connection between Figures 2.2 and 2.4. Each cluster of green code chunks on that earlier diagram will form a **class**. The "gatekeeper" node through which all of the other code chunks must communicate gets mapped to the public interface of the class, while all of the other chunks are put in the private implementation.

2.4 Features of OO

The object-oriented revolution came about simply by taking what was formerly spaghetti code, and learning how to organize it into encapsulated classes. It'll take the whole book to completely unpack that, but for now let me give you a glimpse of some of the features of this paradigm shift:

1. **A higher level of abstraction.** Before OOP, encapsulation was already sort of a thing, since ordinary **functions** gave programmers a way to group code chunks together into cohesive bits. In this older style of **procedural programming**, developers wrote code to compose and combine these functions to achieve a larger purpose. The difference with OOP is that the fundamental building block is no longer the function, but the class, which is a bigger, richer, more sophisticated entity. It encompasses functions, variables, and more besides. Being able to program "at a higher level of abstraction" means you have larger, coarser-grained, more powerful pieces with which to assemble your whole. It's like building a story out of whole paragraphs instead of out of individual words or letters.

2. **Nouns, not verbs.** An old-school function is conceptually a verb: it represents a command to *do* something. As we'll see, a class is conceptually a noun: it has the ability to *be* something. Thus, with OOP you don't think so much about what you want to *execute* – first do this, then that, then print the output – as about what you want to *model* – how can you best represent the important entities in your system? OOP is about building a representation of a world, rather than giving instructions.

3. **Data and behavior *together*.** Before the object-oriented paradigm shift, programmers specified their data separately from the code that operated on that data. They did this deliberately. In a C++ header file, they'd write a number of `struct` definitions, each specifying an assortment of related variable names and types. Then, in many separate source files, they'd have lots of functions that used various of these structures. The "openness" of all this – any function, anywhere, could see and refer to any field of any data structure – was thought to be an advantage.

After many years of painful discovery, it turned out this isn't the right way to do it at all. Instead, you want the opposite. The data associated with a particular type of entity (be it a friend request, a sweater, or a magic sword) ought to be closely bound to the operations (`accept()`, `purchase()`, `wield()`) that make use of that data. This new way of doing things is built in to the object-oriented **class** construct: a class represents some type of entity, and it specifies both the data needed to characterize instances of that entity and the operations that can be performed on those instances. The two are defined right next to each other and maintained in lock-step.

4. **Code reuse.** I'm old enough that I remember when "code reuse" was a pipe dream. People would think, "okay, I need a linked list (or a heapsort algorithm, or a binary search tree, or...), and surely zillions of people have written this sort of thing before. However, it's just too hard to find someone else's code, figure out how to use it, trust that it works, and assimilate it into my program. So I'll just write it from scratch." Seriously. That's how the world worked. I wrote many programs in my early days without using a single line of code from anyone else.

   There were several reasons code reuse was hard, including poor documentation, primitive search engines, and an overall lack of awareness in the software community. But the #1 reason was assuredly the absence of encapsulation. In order to incorporate someone else's linked list (or whatever), you had to locate portions of several different files, all of which were intertwined with other stuff irrelevant to your purpose. You had to understand it enough to gingerly insert it into multiple places in your own files, hoping it could peacefully co-exist alongside your own code. The chances of this were low.

   Nowadays, code reuse is absolutely the standard. If you're writing a program these days, you should only write about 20% of the code yourself; the rest should come from standard libraries or other public sources. You can just grab stuff and use it and be confident it'll work. Why is this easy? Because that "stuff" is encapsulated. It's modular, with no external dependencies. It's all assembled coherently together in nice & neat packages (called classes) that are plug-and-playable. Writing a program is more like building with Legos® than it ever has been.

Postlude

This chapter was very abstract and qualitative. The rest of the book won't be that way. But I felt it was important to lay some groundwork so that you would appreciate what problem object-oriented programming was intended to solve, and how through the miracle of encapsulation it did so. Thanks for making it through.

Chapter 3

Classes and objects

Java is called an "object-oriented" programming language. Now if *I* were King of the World, I would have called it a "*class*-oriented" language instead. That's because in Java, you don't write code for objects, but for *classes*, and the code then defines the behavior of the objects that are based on them.[1] You'll sometimes hear people mistakenly say stuff like, "I wrote some code for the DatabaseConnection object today." It makes me wince. They weren't writing "code for the object," but the code for a class.

3.1 Terms

So here's a crucial pair of definitions. A **class** is a *category* of things. An **object** is a concrete *example* of a class. If "University" is a class, then "UMW" is an object; if "Course" is a class, then "CPSC 240" is an object. The difference is real, and it is vitally important to keep at the forefront of your mind as you begin your OO quest. Getting them mixed up is like Peter Venkman crossing the streams.

You'll sometimes hear alternate definitions of these terms, like "a

[1]There are other languages, for instance JavaScript (no relation to Java), which do IMO deserve the term "*object*-oriented," since you can create code for individual objects rather than classes, and not every object has to have a class at all.

class is a template, and objects are copies of that template." This is better than out-and-out confusion, but it still misses something important. It's an operational definition, instead of a conceptual definition. It describes "class" and "object" in terms of the mechanical way the virtual machine carries out its duties, rather than in terms of their role in *modeling*, which is what OOA&D is all about.

In our world, every single software object will be a member of a category, and that category will define everything about its inner structure and rules of behavior.

By the way, an important near-synonym for class is **type**. (It's only a *near*-synonym because primitive, non-classes like `int`s and `boolean`s are also types.) An important exact synonym for object is **instance**.

In addition to those nouns, an important verb in our vocabulary will be the term **instantiate**. It means "to actually create an object of a particular class." Some people use words like **construct** or **create** for this, or even "`new`" (or "`new` up") as a verb, but for the most part we'll stick with instantiate.

3.2 A different kind of language

Classes and objects are among the basic building blocks of any OO program, and they will play a prominent role on various **UML diagrams**. UML ("Unified Modeling Language") is a *design* language, not a programming language. It is expressed in visual diagrams, not streams of text. Even though it's not text-based, though, and even though there's no "compiler" forcing us to adhere to the syntax, it still has rules that must be followed, and precise meanings that can be inferred.

Figure 3.1 shows what a class, and an object, look like in UML. (I'm putting classes in yellow and objects in blue, but those colors aren't part of UML itself, just the black-and-white stuff.) Both are boxes, but notice the class box has three compartments in it while the object box has two.

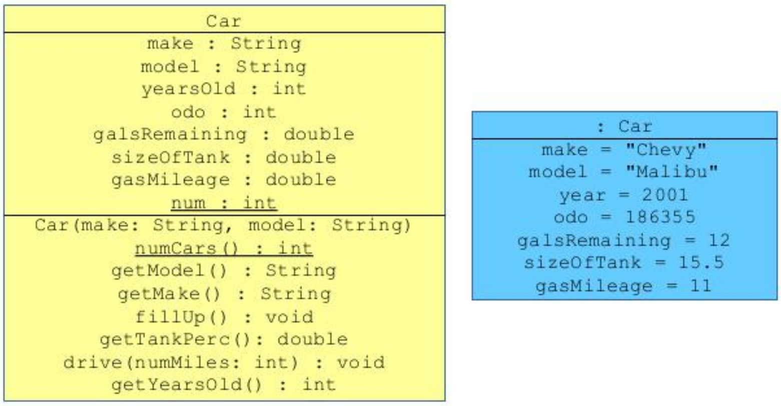


Figure 3.1: A class (left) and an object in UML.

Classes in UML: the first two compartments

Let's look at the class in detail. In the top box is its name; so far so good. One thing to point out, though, is that in Java, *the names of all classes are **Capitalized**.* Don't ever violate this rule, for convention's and confusion's sake!

The second compartment has the class's **instance variables**. You'll hear people use other terms for these like like "member variables" and even "class variables," but I strongly prefer instance variables (or "inst vars" for short) and here's why: *every instance of a class has its own copy of its instance variables.* This truth is absolutely fundamental to OOP, and it's worth re-reading that sentence again and again until it's part of your core being. As you know, declaring a plain-ol' variable like "`int x;`" creates a single storage location in which a value can be stored. But declaring an *instance* variable is a far-reaching choice that destines every `Car` (or whatever) that will come about in the future to have its own copy of that variable. It's our way of defining the very structure of Cars in perpetuity.

One slight headache is that the UML syntax differs from Java's a bit: instead of listing the variable's type and then its name, we reverse them, we use a colon instead of a space, and we omit the semicolon. Otherwise, it's pretty straightforward to interpret that second compartment.

By the way, one important piece of syntax in that second compartment is an underline. If an inst var is underlined, then it actually isn't an "instance variable" after all: it's a **class variable**. This means that *there's only one shared variable for the entire class, rather than a different variable for each object.* In Figure 3.1, the integer `num` variable is the only underlined one. So even though every `Car` has its own `make`, `model`, `odo`meter reading, *etc.*, they all share one `num` (which presumably represents the total number of `Car` objects instantiated so far). This makes sense, since after all such a variable is not specific to a certain `Car`. We'll see that in Java, class variables are created by using the "`static`" keyword where the variable is declared.

Classes in UML: the third compartment

The third compartment isn't much harder: it contains the **methods** for the class. Like everything it seems, programmers have multiple terms for this too: they're called **member functions** or **class functions** on occasion. We'll stick with **method**.

Functions vs. methods

Many programming languages (including Python and C++) allow the programmer to create **functions**, which are coherent chunks of code that can be **called**, **passed arguments**, and **return** a value. You've undoubtedly seen, and written, many such functions in your previous programming courses. They're also sometimes called "**subroutines**."

Java is somewhat strange in this regard in that you normally don't write ordinary functions, but rather **methods**.[2]

The crucial distinction between a method and a regular ol' Joe function is this: while you can simply call a function to trigger it, you must call a method ***on an object***. In the example, we have a `fillUp()` method defined on the `Car` class. Since it's not an

[2]As we'll see, adding the word "`static`" before a method declaration makes it a so-called "**static method**," which *is* essentially a function.

ordinary function, but rather an OO method, we must call it on a particular instance of a `Car`. In Java code, this does *not* work:

```
fillUp();        // NOPE
```

nor does this:

```
Car.fillUp();    // NOPE
```

Instead, one must call `fillUp()` like this:

```
johnsMercedes.fillUp();     // Correct!
```

where `johnsMercedes` is the name of a valid `Car` object, previously instantiated. This is what we mean by "calling `.fillUp()` ***on*** the `johnsMercedes` `Car` object."

Beginners sometimes view this as a syntactic nuisance. It is not. It is fundamental to what your code *means.* Conceptually, it makes sense to have a particular car, and to fill it up. It does *not* make sense to say "hey universe, fill up cars" (which is what "`fillUp()`" seems to say) nor to say "hey Cars-in-general, fill yourself up" (which is what "`Car.fillUp()`" seems to say).

By the way, notice in the example I just gave, `johnsMercedes` is *not* capitalized. (The capital "`M`" in the middle doesn't count; that's just an artifact of camelCase, which is a way of making multiple words easier to read.) This is *always* true: in Java, object names should always begin with a lower-case letter.

Back to the third compartment. You can probably tell that the things inside the parentheses are arguments to the respective methods, with the same name-first-then-type colon-syntax, and you can probably tell that after the closing parenthesis, you have the return type of the function. All of this looks vaguely Java-like, and that's because even though a UML diagram is technically programming-language-independent, language-specific things like `int` and `String` can't help but creep in in practice. Our thoughts betray us.

Various "special" methods

A few of those methods are worthy of special note. The first one listed, called simply "`Car`", is a very special kind of method called a **constructor** which we'll be talking about a lot. Here's an iron-clad rule which is fundamental to much that follows: *whenever an object is instantiated, one of its class's constructors is called.* This happens automatically; it's not something we have to do ourselves. (Java's syntax for this, as we'll see, makes it kind of look like we're calling the constructor ourselves. This is a mixed blessing.) In Java, there are two things that "make" a method a constructor: (1) it must have exactly the same name as the class, and (2) it must have *no* return type. (Note that "no" return type is not the same as a `void` return type! I mean *no return type at all.*[3])

By the way, just as a class can have multiple methods with the same name as long as those methods have different argument lists, so it can have multiple constructors subject to the same conditions. This is a very common practice, although in this first example we have only one `Car` constructor.

Also, just as in the second compartment, an underline indicates that the method "goes with the whole class, not with each object." And just as before, this implies the use of the `static` keyword. A `static` method is essentially a function: *i.e.*, you *don't* call it on an object. Instead, you just call it *on the class itself.* In the example above, `numCars()` method is `static`, which means that you could write "`Cars.numCars()`" to retrieve the number of `Car` objects that have been instantiated to that point. Static methods are quite rare, but they do arise occasionally, and are always indicated with an underline in UML.

The other methods I'll draw your attention to are the ones that begin with "`get`". People call these methods "**getters**," and normally

[3]If you mistakenly include the word `void` before your constructor when you write the code, it is officially no longer a constructor! It's now just an ordinary method – weirdly named the same as the name of the class it's in – which will *not* be automatically invoked at instantiation time as a constructor should. I once had a nasty bug at the eleventh hour of a software release because of this exact issue.

they simply return the value of the instance variable in question. Often one also has "`set`" methods to set the values of inst vars, although our example doesn't have any of those. Btw, some people also call getters and setters **accessors**, and sometimes specifically call setters "**mutators**," a term which always made me chuckle.

Objects in UML

Now let's examine the blue box in Figure 3.1, which represents an object rather than a class. It has only two compartments, not three. That's because there's no need (in most OO languages) to say anything about an object's methods when focusing on the object: after all, the methods are simply defined by the class, and are common to all instances of that class. It is important, however, to specify the current **`state`** of the object, which means the values, as of now, of all its instance variables. In the picture, you can see that there is a `Car` object in memory representing an old Chevy Malibu with a zillion miles on it and other suboptimal features.

Perhaps the strangest thing about a UML object is the top compartment. Notice that it says "`: Car`" ("colon-Car"), which is not a typo. Here's the sitch. The top compartment of a *class* has the class's name, since that's all there is to say about it. The top compartment of an *object*, meanwhile, has the *object's* name, followed by a colon and then its class. Just like we said "`make : String`" earlier, so here we can say "`johnsMercedes : Car`".

Okay...but why, then, is Figure 3.1 missing the name before the colon entirely? Because we've chosen *not to name this object in the diagram.* It's just "a Car" with certain properties, not a named Car. This may seem odd, but in fact 99% of the time we will do exactly this. And that's because bizarrely, *objects don't have names in Java*, even though it may seem at first that they do.

More on that later. For now, just accept the fact that UML diagrams can depict objects, and normally we don't choose to specify the object's name – only its type and its instance variable values.

The value of “design”

Before we move on to implementation, take a step back for a moment and consider the *information* contained in that Figure 3.1.

Suppose you were given the job to write a car maintenance tracking program, and you were getting started figuring out how to accomplish that. I think you’ll agree that if someone handed you that diagram, it would be valuable indeed. There’s no code in it *per se*, but a great deal of the work has already been done for you! You already know what to name your class, the names and types of all its constituent variables, and what methods its objects should support. With the diagram alone, I’d say 70% of the work has been done for you.

The blueprint communicates a ton of information about decisions that have already been made. With your structure defined, you now just need to bust out a hammer and some nails.

3.3 Classes in Java

In Java, every class is in its own file[4] named the same as the name of the class (including the capital letter) with a `.java` extension. Operationally, we can use `vim` to create it and edit it:

```
$ vim  Car.java
```

The skeleton of any class file – after the `package` and `import` statements we’ll talk about later (p. 101 and p. 121, respectively) – is the class definition, with curly braces:

```
class Car {

}
```

[4]Technically there can be some exceptions to this, but don’t worry about them now.

You may be used to putting the word "`public`" before the word `class` here. For now, we won't do this, and I'll encourage you to ditch the habit of making classes `public` by knee-jerk reaction. As we'll learn, you want to lean towards making things "as private as possible" until you have reason to do otherwise.

Instance variables in Java

Instance variables go directly inside the class definition, and outside of any method:

```
class Car {
    String make, model;
    int yearsOld;
    int odo;
    double galsRemaining;
    double sizeOfTank;
    double gasMileage;
}
```

You may be used to seeing the word "`private`" before each instance variable, and I do applaud that practice. More on that later. For now, we'll leave it off just because it's not necessary to compile. Realize that it's not the word `private` that makes something an instance variable; rather, it's the fact that it's defined directly inside the class, rather than within a method.

Constructors in Java

Next on the diagram is our constructor. We put in the boilerplate to get us started:

```
class Car {
    ...

    Car(String make, String model) {

    }
}
```

and now for the first time we have to actually think.

A constructor, as I said, is automatically called whenever an object comes into existence. This is our "hook" to set up the object for success when methods are called on it later. Think of it this way: your constructor is called whenever a new object is about to come off the assembly line and enter the real world. Your job in the constructor is to do everything necessary to make sure it's ready for prime time.

Often this will involve initializing the instance variables to reasonable values. Sometimes it will include other things, like registering its existence in some global repository of objects, or initializing a connection to a network, or writing itself to a database. The key question to ask yourself is, "what do I need to do to ensure this object is 'legit' and doesn't break anything when it's being used?"

Analyze `this`

In our case, initializing the instance variables are all we need to do in the constructor. First, let's set the object's `make` and `model` to what was passed:

```
class Car {
    ...

    Car(String make, String model) {
        this.make = make;
        this.model = model;
    }
}
```

If this is the first time you've seen the odd word "`this`" in a program, have a good chuckle. What a weird word choice! But Gosling & Co. chose this word to denote a central OO programming concept. The word "`this`" means one of two different things, and they both need to be memorized:

1. Inside a *constructor*, "`this`" means "the object that is currently being instantiated."
2. Inside a *method*, "`this`" means "the object the method was called on."

×. (Anywhere else, "`this`" is illegal.)

It's weird and meta and self-referential, but it's also necessary. There are times when we need to have a name for "the very object I'm 'in' right now," and "`this`" is our (awkward) name to refer to that.

So in our constructor, when we say "`this.make`" we mean "the `make` instance variable of this very object that is in the process of being birthed." We set that to the `make` argument that was passed to the constructor. Ditto with `model`. Oftentimes, using `this` is optional, but in the present case it's required because we named our argument the same as the instance variable, and there has to be a way to distinguish between the two.

Now for our other inst vars. Some of them make sense to be set to zero:

```
class Car {
    ...
    Car(String make, String model) {
        this.make = make;
        this.model = model;
        this.yearsOld = 0;
        this.odo = 0;
        this.galsRemaining = 0.0;
    }
}
```

since brand new cars are in fact zero years old, have a 000000 odometer, and have no gas in their tank (maybe). Zero values for the other two don't make sense, however; brand new cars still have a gas tank of a certain size, and they certainly get more than 0 mpg. For this example, I'm going to go with a very limited notion of automotive properties:

```
class Car {
    ...
    Car(String make, String model) {
        this.make = make;
        this.model = model;
        this.yearsOld = 0;
        this.odo = 0;
        this.galsRemaining = 0.0;

        if (make.equals("Chevy") || make.equals("GM")) {
            sizeOfTank = 21;
        } else {
            sizeOfTank = 13;
        }
        if (make.equals("Chevy") && model.equals("Malibu")) {
            gasMileage = 3;
        } else {
            gasMileage = 24;
        }
    }
}
```

I'm totally not bitter about my car's gas mileage, by the way.

Methods in Java

The other methods follow a similar syntactic pattern. But it's super important to keep this truth in the front of your mind: *because they are methods (not functions), they are called* ***on an object.*** That means that you can refer to instance variables inside of them – and when you do, you're talking about *the instance variables of the object the method was called on.* Put another way, you're talking about the instance variables of `this`.

"Client code" and thinking reactively

When you write methods in an OO program, you have to think reactively, not proactively. What I mean is this. When you write a procedural, old school program, you're the one in control. You set the agenda. In your `main()` you say, "first do this, then do that; create these three variables, perform a computation, and then print the result."

We all learned how to program this way. But in OO, you kind of have to think backwards from that. Writing a method isn't like calling it; instead of giving orders, you're providing a service to whoever called you. So when you write a method, you have to think, "okay, some other part of the code is now calling me, for reasons of its own. What do I do in response to that?"

Our term for "that other part of the code that is now calling me" is **client code** (or sometimes just "a **client**.") The word "client" is used in a lot of different ways in high-tech, but here we just mean "the code that wants to use a particular object." The word connotes a respected customer, whom we want to treat well.

Very well, some client code calls one of our methods. How should we react?

Often we'll react by updating the object's **state** to reflect what should happen to it as a result of the method being called. Sometimes we'll produce (return) a value that is stored by the object in question or computed on the fly. Other times we'll trigger some side effect, like printing to the screen, writing to a database, or calling some *other* method(s) on the same or a different object.

This is best seen through examples. Let's implement[5] the `.fillUp()` method first. Don't think about Java; think about cars. If I fill up a car, what happens?

Does the make or model or mileage change? Of course not. The amount of gas in the tank does. And "fill 'er up" means to raise it to its maximum. The correct implementation of `.fillUp()`, then, is simply:

```
class Car {
    ...
    void fillUp() {
        galsRemaining = sizeOfTank;
    }
}
```

[5]The verb **to implement** means "to take a design and actually build it out." It is a synonym for the verb **to code**.

We could equally well have written this as:

```
class Car {
    ...
    void fillUp() {
        this.galsRemaining = this.sizeOfTank;
    }
}
```

to make it explicit that we're talking about two instance variables here, and assigning the value of one to the other. It's a matter of style.

In the same vein, we ask ourselves, "suppose some client code asks me what percentage full my tank is. What answer do I give them?" The proper response involves these same two inst vars and a little math:

```
class Car {
    ...
    double getTankPerc() {
        return galsRemaining / sizeOfTank * 100;
    }
}
```

I chose to omit `this`, but again it's a personal choice.

Some methods are no-brainers:

```
class Car {
    ...
    String getModel() {
        return model;
    }
}
```

If a client asks me what my model is, I tell them my model, duh. The same is true for the other accessor methods.

Finally, what if a client instructs me to drive n miles? How should my internal state be adjusted to reflect that?

This is the most difficult one, and again it requires you to think about cars rather than about Java. Mentally run through the variables we've chosen to represent a car, and ask yourself which ones need to change, and how? You'll realize that the odometer and the gas tank level are the two we need to modify. When someone drives a car n miles, the odometer needs to increase by n miles (else it ain't legal); also, the gas tank needs to be reduced by $\frac{n}{m}$ gallons, where m is the car's gas mileage in mpg. So here we go:

```
class Car {
    ...
    void drive(int numMiles) {
        double galsBurned = numMiles / this.gasMileage;
        this.galsRemaining = this.galsRemaining - galsBurned;
        this.odo += numMiles;
    }
}
```

This time, I did include the `this`'s where appropriate, since we also have a couple of local variables involved and I wanted to be explicit. Our math is a mix of function parameters, temporary variables, and permanent attributes of the `Car`.

3.4 Objects in Java

We've now coded a class from the ground up (the complete code listing is in Figure 3.3.) We did this so clients can instantiate objects of that type and do something with them. Let's make a simple `main()` method to do just that.

You'd be surprised how many beginning programmers try to drive 23 miles like this:

```
    public static void main(String args[]) {
        drive(23);    // WRONG!
    }
```

or this:

```
    public static void main(String args[]) {
        Car.drive(23);    // equally WRONG!
    }
```

Yes you'll get compiler errors, but those errors reflect a deeper and more fundamental misunderstanding. In OOP, you have to call a method *on an object.* Conceptually, nothing else makes sense. In real life you don't "drive in general," and you don't ask "automobiles in general" to drive you places. Instead, you have to drive *a particular car* somewhere. Here's how:

```
    public static void main(String args[]) {
        Car minivan = new Car("Toyota","Sienna");
        minivan.drive(23);    // correct!
    }
```

The keyword "`new`" is utterly crucial here. In Java, *the only way to instantiate an object is to use `new`.* It causes a fresh object of the appropriate type to spring into existence, complete with memory to hold its instance variables. And the appropriate constructor is called, of course, to set that object up for prime time.

We got errors before because we didn't even *have* a car to do anything with. There was no memory set aside, no constructor called to set up the object, nothing. We tried a shortcut, and that was madness. But now that we know how to instantiate objects, we can do so to several and create a whole new world, as in Figure 3.2 on p. 51. All the code in that figure is legit, and shows that our `Car` class has uses.

Printing an object

One last thing before we bring this chapter to a close. Suppose we're debugging our program, and we want to print out the values of various variables to help us hunt down an error. Printing an `int` or other standard type is straightforward:

```
public static void main(String args[]) {

    // The archaic Davies family vehicles
    Car minivan = new Car("Toyota","Sienna");
    minivan.setYear(2002);
    Car stephensLemon = new Car("Chevy","Malibu");
    minivan.setYear(2001);

    // Grammy lives in Colorado
    Car grammysCar = new Car("Lexus","ES");
    grammysCar.setYear(2021);

    // Caravan to Disneyworld!  (Grammy's meeting us there.)
    minivan.fillUp();
    minivan.drive(500);
    stephensLemon.fillUp();
    stephensLemon.drive(500);
    System.out.println("The van is " + minivan.getTankPerc() +
        "% full, while the chevy is " +
        stephensLemon.getTankPerc() + "% full.");
    grammysCar.drive(1899);  // a long way from Colorado
}
```

Figure 3.2: A client `main()` program that uses the `Car` class.

```
int numEnchiladas = 3;
System.out.println("We'll eat " + numEnchiladas + "enchiladas.");
```

and will produce a message like "`We'll eat 3 enchiladas.`" What happens, though, if we print out an *object*, like a `Car`? How can such a complex entity be reduced to a string of text?

Heck, let's try it:

```
  Car porsche = new Car("Porsche","Carrera");
  porsche.setYearsOld(2);
  System.out.println("My car is: " + porsche + ".");
```

The output we get is:

```
My car is: Car@4aa298b7.
```

Whoa. The word "`Car`" is perhaps not surprising, but what's the rest of that gunk?

It turns out that Java's default way of rendering an object as a `String` is to concatenate the name of the class, an "at" sign, followed by *the memory address* in which it is stored. We'll talk much more about memory in the next chapter. For now, just think of the memory address as a unique number[6] that identifies the object, like an SSN.

The cool thing is, we can **override** this functionality at will, and change the way `Car`s will be printed. Check it out. Create a method in the `Car` class called `.toString()`. It must:

1. be called exactly "`.toString()`".
2. take no argument.
3. return a `String`.
4. have the word `public` immediately before the return type. (We'll talk a lot about what `public` means in future chapters. For now, it just has to be there.)

Here's one:

```
    public String toString() {
        return "a " + yearsOld + "-year-old " + make + " " + model;
    }
```

We're assembling various aspects of the vehicle into a sensible, readable string. Now, when we run *the same code* as above, our output is this:

```
My car is: a 2-year-old Porsche Carrera.
```

Notice that we didn't explicitly ever call `.toString()`! Instead, we just used a `Car` object in a context in which a `String` was

[6]Yes, it is indeed a number, despite the fact that it has letters in it like '`a`' and '`b`'. It's printed here in **hexadecimal**, which is a base-16 number system instead of the base-10 system non-computer-science humans use.

required, and Java faithfully called our method instead of the one that generated the memory address. Pretty cool.

This is actually our first foray into a really deep and powerful technique called “inheritance,” about which much more will come in later chapters. For now, just grasp the idea that Java lets us override its general behavior for specific kinds of objects, which gives us tremendous power and flexibility.

```
class Car {
    String make, model;
    int yearsOld, odo;
    double galsRemaining, sizeOfTank, gasMileage;

    Car(String make, String model) {
        this.make = make;
        this.model = model;
        yearsOld = 0;
        odo = 0;
        galsRemaining = 0;
        if (make.equals("Chevy") || make.equals("GM")) {
            sizeOfTank = 21;
        } else {
            sizeOfTank = 13;
        }
        if (make.equals("Chevy") && model.equals("Malibu")) {
            gasMileage = 3;
        } else {
            gasMileage = 24;
        }
    }

    public String toString() {
        return "a " + yearsOld + "-year-old " + make + " " + model;
    }

    String getMake() { return make; }

    String getModel() { return model; }

    int getYearsOld() { return yearsOld; }

    void setYearsOld(int x) { yearsOld = x; }

    void fillUp() {
        this.galsRemaining = this.sizeOfTank;
    }

    double getTankPerc() {
        double perc = galsRemaining / sizeOfTank * 100;
        return perc;
    }

    void drive(int numMiles) {
        double galsBurned = numMiles / gasMileage;
        galsRemaining = galsRemaining - galsBurned;
        odo += numMiles;
    }
}
```

Figure 3.3: A complete Java implementation of the `Car` class.

Chapter 4

Memory matters

This chapter is near and dear to my heart. The concepts here are vastly undertaught by computer science educators today, and yet they are at the epicenter of most intermediate students' understanding (or misunderstanding). A failure to master this material slaps a hard ceiling on what you can accomplish as a programmer. Successfully mastering it is the key to the next level.

The key idea is that there are two ways of looking at a computer program. One is to look at the static lines of code as they are written on a screen or on paper. This is how novices think about programs: they look at the lines of code, and ask themselves whether lines need to be added, removed, changed, or moved.

The other way is to think about what happens to the computer's **memory** as the program runs, and how its variables and structure change as the program unfolds. Whether they realize it or not, this is how all proficient programmers think. It turns out that **the "purpose" of almost any line of code is to change the contents of memory in a particular way.** The name of the game is recognizing what impact on memory each line of code has – and conversely, what line of code is required to make a particular change to memory.

4.1 Memory diagrams

The focal point of this chapter will be the **memory diagram**, which incorporates the UML object representations we discussed in section 3.2. A memory diagram depicts the contents of the computer's memory at a *snapshot in time.* At any given moment, as a program is running, you could say "Freeze!" and look at the memory diagram. You'd see the exact state of the system at that moment.

The stack and the heap

A program's memory, it turns out, is divided into two realms with funny names: "**the stack**" and "**the heap**." It is vital to understand the difference between the two, and which one is used for what. The stack contains **statically-allocated**, and the heap **dynamically-allocated**, memory. We'll unpack what all this means, but first let me show you a full list of differences:

stack	**heap**
statically-allocated	dynamically-allocated
holds named things	holds unnamed things
holds primitive types and references	holds objects[1]
items have limited lifespan	items have unlimited lifespan[2]

This is best understood by example, and in fact can be illustrated with just a small function:

```
void illustration() {
    int year = 2021;
    Car minivan = new Car("Toyota","Sienna");
}
```

[1]This is true in Java, but C++ permits programmers to store objects on the *stack* as well as the heap. I will argue that's universally dumb, and it's a large part of what makes programming in C++ difficult: you have to account for that occurrence with a ton of tedious and error-prone bookkeeping.

[2]Not completely unlimited, but things on the heap stick around as long as they're needed, rather than evaporating at the end of their current function.

This teensy function, when it runs, produces memory contents as depicted in Figure 4.1. Let's go through it carefully.

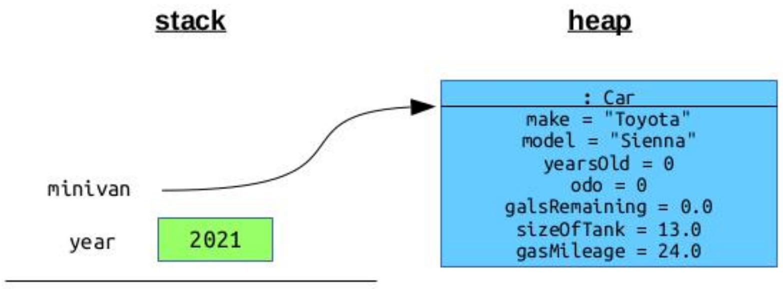


Figure 4.1: The stack and the heap.

The first line of `illustration()` creates a simple integer variable and sets it equal to 2021. Since an `int` is a **primitive type**[3], it is stored on the stack. "On" the stack should make you think of layering items vertically on a surface. Before this line of code executed, nothing existed in the program's memory at all, so the stack was nothing but a bare floor (think of it as a horizontal line). Our first variable goes right on top of that floor.

There's a ton packed into that second line of code, so hold on to your seats. The first thing to realize is that *it encompasses both stack and heap.* We have a named **reference variable** called `minivan`, which, as with all named things, goes on the stack (right on top of `year`). A "reference variable" means a variable that has the ability to reference (or "refer to," or "point to") an object. However, the object itself is created in the heap, because in Java that's where all objects live. The word `new` is a "heap word": using it is the only way to make an object at all, and therefore, the only way to make something on the heap. Finally, to carry out the equals sign ("`=`") in that line of code, we draw an arrow from `minivan` to the object to indicate that's what it's currently referring to.

Okay, now a head-scratcher. Look at Figure 4.1 again. What would

[3]If you've never heard this lingo, a "primitive type" is one of the very basic lower-case Java variable types, like `int`, `double`, or `boolean`. Importantly, a primitive type is *not* an object.

```
void illustration() {
    ...
    Car other = new Car("Ferrari","F355");
    Car t = minivan;
    minivan = other;
    other = t;
}
```

Figure 4.2: (Continuing the previous example.)

you answer if I asked you, "what's the *name* of that blue object?"

If you're like 99% of novice programmers (including myself, long ago), you would confidently answer, "`minivan`. Its name is `minivan`." That seems to make perfect sense. But unfortunately it is *wrong.* The truth is that *the object has no name.*

Again, you may think I'm being pedantic. Let me demonstrate why I'm not. Suppose we expanded our previous code with four more lines, as depicted in Figure 4.2. Study it carefully.

Let's deal with the first two of these lines. The first one creates a new reference variable called `other` on the stack, and points it to a brand new `Car` object (unrelated to our Toyota Sienna) in the heap. Notice that unlike with the stack, I didn't carefully put the new `Car` exactly on top of the first one. Instead, I just threw it in there helter skelter. This is how the heap works, and in fact why it's called a "heap": it's a disorganized mess of stuff that comes and goes in response to the program's unpredictable needs. The stack is as tidy as the Library of Congress; the heap is a teenage boy's room. Seems weird, but it turns out things have to be that way.

The second line creates a new stack variable called `t` but emphatically does *not* create a new `Car` object. Let that sink in deeply. Many programmers, upon seeing a line begin with "`Car t = ...` would naturally assume that line is making a new `Car`. But it's actually only creating another variable that has the *potential* to refer to a `Car`. And in fact, after the equals sign, we do point it to a `Car`...but one of the ones we've already instantiated (namely, the Sienna).

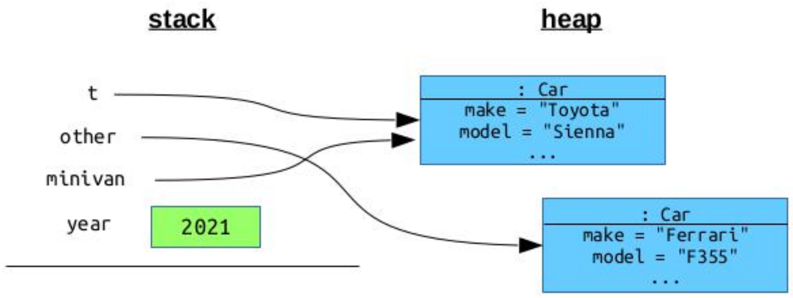


Figure 4.3: After executing the first two lines of code Figure 4.2.

The result of executing these two lines is shown in Figure 4.3. Stare very carefully at that figure and mull over each box and line. We have four named variables, three of which are of type `Car`, and yet there are only *two* `Car` objects because we only executed two `new`'s. And two of our named variables – `t` and `minivan` – are pointing to *the same object*. This turns out to be okay. We'll have multiple references to the same object all the time, and it's entirely healthy. What's critical not to miss is that `t` and `minivan` are not referring to identical copies of the `Car`, but literally *the same* `Car`. If we were to change the state of `t`'s `Car` by, say, increasing its odometer instance variable, `minivan` would instantly experience the same change. And that's because they *are* the same.

Okay, now the punchline of this whole example. I'm going to complete the bait-and-switch, just to prove I was correct back when I said "the name of that first blue box is *not* `minivan`." Let's do the *second* two lines of code in Figure 4.2:

```
...
minivan = other;
other = t;
```

The result of those two operations is to change what the `other` and `minivan` variables are pointing to. Memory now looks like Figure 4.4. And so I ask you again: "what's the *name* of that Toyota Sienna object?" I think you'll agree that `minivan` is most

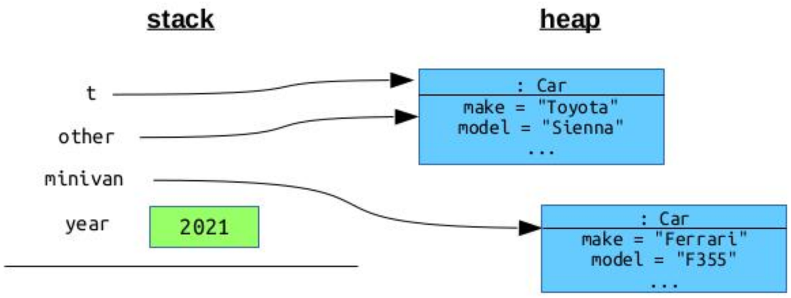


Figure 4.4: Finally, after executing the rest of code Figure 4.2.

certainly *not* its name. Two valid ways to refer to it are `t` and `other`, both of which point to it. But neither one is its name. Objects simply have no name.

Names are ephemeral, momentary: they're only used temporarily so we can get at the stuff in the heap, which is ultimately what matters.

Let me conclude this example by explaining what I meant earlier about "limited lifespans." After executing the "`other = t;`" line, we are done with the function. It's time to return control to whoever called `illustration()` in the first place. And at this point, all of our named variables – `t`, `other`, `minivan`, and even `year` – cease to exist. Their destiny was only to provide service during the time that `illustration()` was being executed.

But the stuff on the heap lives on after. Long after a function is completed, the objects it may have created or changed have a presence that will affect the behavior of other, future functions. In this case, since we weren't passed any arguments and didn't return anything, our Toyota and Ferrari *will* actually peacefully go away. But in general there are meaningful, long-term effects, and in the next section we'll see an example in action.

Most methods are just like this. They create a few named variables so they can change the contents of the heap in some way, and then clean up their toys and return with the heap thus changed. That is their *raison d'etre*. It's a short but happy life.

```
class Simulator {
    public static void main(String args[]) {
        int year = 2021;
        String greeting = "Play ball!";
        Ballplayer oldGeezer;

        ArrayList yankees = buildDaTeam();
        int rosterSize = yankees.size();
    }

    static ArrayList buildDaTeam() {
        String name = "Yankees";
        int year = 1927;
        ArrayList team = new ArrayList();

        Ballplayer ruth = new Ballplayer("Babe Ruth");
        ruth.setUni(3);
        ruth.setPos("OF");
        Ballplayer gehrig = new Ballplayer("Lou Gehrig");
        ruth.setUni(4);
        ruth.setPos("1B");
        Ballplayer babe = ruth;
        babe.setUni(3);       // (Pointless, as it turns out.)
        babe.setPos("OF");    // (Pointless, as it turns out.)
        team.add(babe);
        team.add(gehrig);
        team.add(ruth);

        return team;
    }
}
```

Figure 4.5: Some code that calls a function.

4.2 Calling functions

One thing our previous example didn't include was calling a function or method. In this section, we'll see what happens to memory when we do this. There will probably be a few eye-openers for you.

First, take a look at our code listing (Figure 4.5). We'll switch from an automotive domain to part of a baseball simulator.

Let's see how the memory diagram emerges line-by-line in response

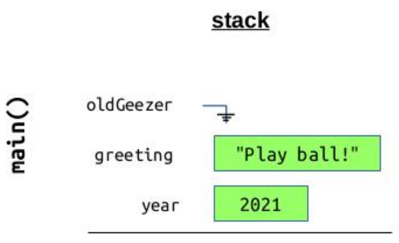


Figure 4.6: Memory contents after executing the first three lines of `main()`.

to the code executing.

First, let's execute the first three lines of `main()` and "Freeze!" the picture. The result of these lines is shown in Figure 4.6. There are three new things here worth mentioning. First, notice that our `greeting` variable, although it is a `String`-with-a-capital-S and therefore an object, is shown on the *stack*, just like the `int year` is. The reason for this is that Strings are a kind of in-between case (between primitive types and objects) – they're neither fish nor fowl. Technically they're objects, but Java actually treats them somewhat specially, and even has an inline syntax to create what are actually instances, so it ends up making more sense to treat them as primitive types on the stack. That's what we'll always do with `Strings`.

Null references and NPEs

The next new thing is that bizarre symbol next to `oldGeezer`. *Whazzat?* If you look at the code listing, you'll see that we declare a variable of type `Ballplayer` named `oldGeezer`, but we never set it equal to a `new` instance, nor to anything else for that matter. This means that `oldGeezer`, which as you'll recall is a "reference variable" (capable of referring to a `Ballplayer`) currently refers to *nothing*. In Java, this is called a **null reference** (or **null pointer**) and is indicated with the keyword `null`. In fact, this line of code has exactly the same effect as the one above:

```
    Ballplayer oldGeezer = null;
```

Some people prefer to be explicit like this. I don't care either way; just realize that at this point, if you attempted to call *any* method on `oldGeezer`, like:

```
    Ballplayer oldGeezer = null;
    oldGeezer.strikeout();
```

you will then be hit with the most ubiquitous of all Java run-time errors, the **null-pointer exception** (or "**NPE**"):

```
Exception in thread "main" java.lang.NullPointerException
    at Simulator.main(Simulator.java:5)
```

This is quite reasonable behavior, if you think about it. What can Java do if you try to "call a method on an object" but there *is* no such object? It can only throw up its hands, which it does here.

Remember: an NPE means *you tried to call a method on an object, but the variable name you called it on wasn't actually an object; it was* `null`. The way to diagnose an NPE is to look at the line number it gives you, and find the dot ("`.`") (or dots) on that line. The variable or expression to the *left* of one of those dots is an uninitialized, null reference. Guaranteed.

Stack frames

The last new thing in Figure 4.6 is easy to miss: it's the word `main()` off on the left-hand side of the diagram. What this means is that *all the variables to the right of it "belong" to the `main()` function.* This group of variables, which goes with a particular call to a function[4], is called a **stack frame**. The way a program works

[4]Note carefully that a stack frame is associated with each ***call*** to a function, not each function. This may seem pedantic, and it is...until we consider **recursion**. A recursive function will call itself, which will call itself, which will call itself...many times. *Each call* to the function generates its own stack frame, which is separate from all the others. This is how recursive functions are able to work without clobbering the values of the variables contained in previous, still-active calls.

is this: every time a function is called, a new stack frame is "pushed" on top of the stack (above a horizontal line that we'll draw.) While we're in the function, *Java can only see the variables in that current stack frame.* The ones in `main()`'s stack frame, or any other stack frame for currently-in-progress functions, are safely nestled away to be resumed later, but they are not immediately available to the program.

This is exactly how it should be. If we call a function `foo()` from within a function `bar()`, control transfers to `foo()`. Now how could `foo()` possibly refer to `bar()`'s variables? Heck, whoever wrote the code for `foo()` didn't even know it would be *called* from `bar()`! Any other function could have called it just as well, in which case `bar()`'s variables wouldn't even exist. All we know for sure is that `foo()` was called from "somewhere," and thus must work no matter what the context. If Java allowed us to talk about variables in another stack frame, our function would instantly become non-reusable; it would only make sense if called from some specific other function. And that defeats most of the purpose of even having a function.

Okay, now the big moment. We run this line:

```
        ArrayList yankees = buildDaTeam();
```

which calls the `buildDaTeam()` function and transfers control to it.

Hang on to your hats. A lot happens here. First, a *new stack frame* is created, labeled `buildDaTeam()` in the diagram to carefully distinguish it from the other. Then, `buildDaTeam()` starts executing. Let's do the first three lines. We create two new variables on the stack (a `String` and an `int`). One of these (`year`) has *the same name* as a variable that was declared down in `main()`. This is perfectly okay, and the two `year`s in fact have nothing whatsoever to do with each other. As long as `buildDaTeam()` has control, "`year`" means *the top year*, in `buildDaTeam()`'s stack frame.

In the third line, we create our first heap object of the entire program. It is created (as all objects are created) with the `new` keyword. This newly instantiated thing is an `ArrayList`, and we'll draw it

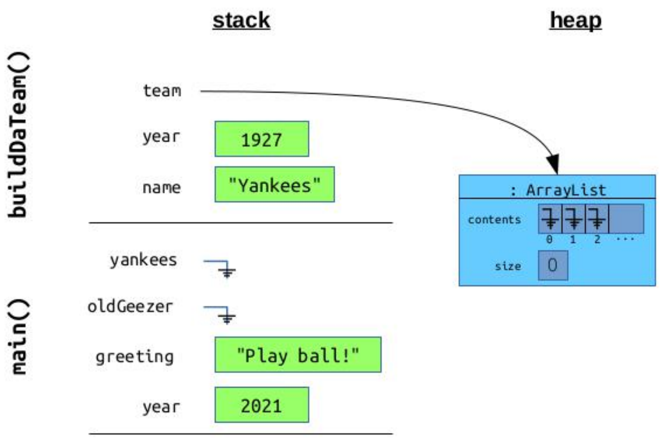


Figure 4.7: Memory contents after calling the function and executing the first three lines of `buildDaTeam()`.

as indicated in Figure 4.7. It has some `contents`, which is a zero-based, array-ish list of references, each of which has the potential to point to an object.[5] Currently none of them do so, and therefore the diagram shows the `null` symbol for each.

Freeze! The program's memory now looks like Figure 4.7. Run your eyeballs over it and make sure you understand every box and line.

Object craziness

Now for the next part of code listing 4.5. These three lines:

```
        ...
        Ballplayer ruth = new Ballplayer("Babe Ruth");
        ruth.setUni(3);
        ruth.setPos("OF");
        ...
```

[5]You may be more used to seeing `ArrayList<Ballplayer>` instead of just plain `ArrayList`, which actually is a better choice. When we declare something as type "`ArrayList<Ballplayer>`" we're saying "Java, please prevent me from storing anything in this `ArrayList` *except* `Ballplayer`s." See section 8.3 for more details.

instantiate a new `Ballplayer` object (on the heap, of course) and set it to some initial values. You need a little imagination to envision what the `Ballplayer` class does in response to these method calls, but only a little: obviously it has a constructor that takes a `String` (the player's full name) and a couple of accessor/mutator methods to set the player's uniform number and position.

We then do the same sort of thing again, for another player:

```
    ...
    Ballplayer gehrig = new Ballplayer("Lou Gehrig");
    gehrig.setUni(4);
    gehrig.setPos("1B");
    ...
```

to get another one. Then, we do this:

```
    ...
    Ballplayer babe = ruth;
    babe.setUni(3);       // (Pointless, as it turns out.)
    babe.setPos("OF");    // (Pointless, as it turns out.)
    ...
```

which you know by now does *not* instantiate a new object. (After all, there's no `new`.) Instead, the first line *points the new variable* ***`babe`*** *at the same object* ***`ruth`*** *is currently pointing to.* Get very, very comfortable with the idea that except for primitive types, "`=`" in Java does not do anything resembling a "copy" operation. It simply makes a reference variable refer to something else. So we now have three variables of type `Ballplayer`, but only *two* `Ballplayer` objects.

Finally, we add these players to our `ArrayList`:

```
    ...
    team.add(babe);
    team.add(gehrig);
    team.add(ruth);
    ...
```

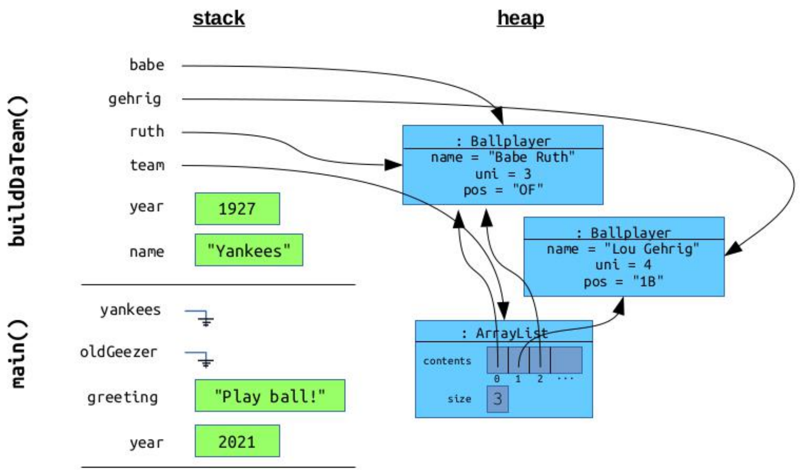


Figure 4.8: Memory after all the object creation done in `buildDaTeam()`.

Stare closely at all those crazy arrows in Figure 4.8 and make sure you understand where they're all going and why. Our `ArrayList` object, instead of showing `null` pointers, now has each of its slots pointing to a particular `Ballplayer` object. Elements 0 and 2 point to the *same* object, of course, because we added "`babe`" and later "`ruth`" and those two variables are pointing to the *same* object. (So we're cheating here, baseball-wise: you can't actually have the same player twice in the lineup! This is just an example.)

"I shall return"

And now, we're ready to polish off this bad boy.

```
        ...
        return team;
    }
```

That "`return`" statement packs a wallop. When the function is completed, two huge things happen. First, the function's stack frame *is entirely wiped out.* Like, off the face of the planet. Every single variable in there is irrevocably deleted and never mentioned again.

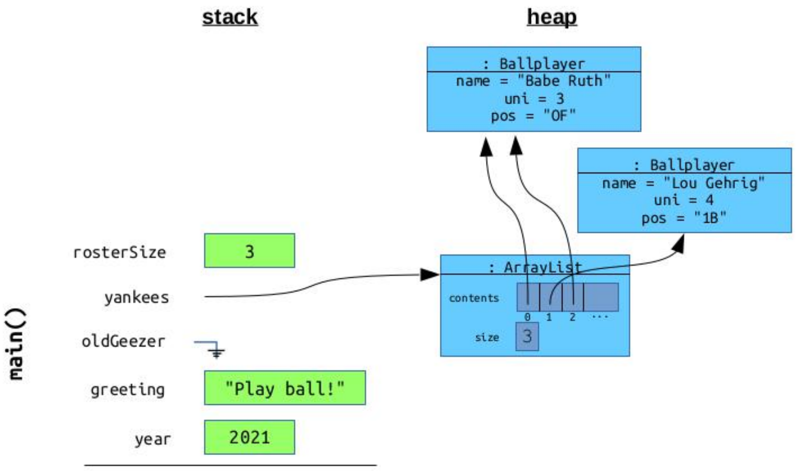


Figure 4.9: What memory looks like when we reach the end of `main()`.

When students first hear this, they're sometimes dismayed – "what's the point of calling a function then, if every single thing it creates is erased?" Ahhhh...but they're only thinking of the *stack*, not the heap. The heap-ish things that a function accomplishes *do* live on, and as I said earlier, they are the reason the function existed in the first place. Almost all functions' sole job is to inspect or manipulate the heap in some way.

When I say the stack frame is wiped out, here's what's wiped out: (1) all the named variables in the stack frame, (2) all the primitive type values in the stack frame (green boxes), (3) all the arrows emanating from the stack frame's reference variables, (4) the word on the side of the diagram that names the function, and even (5) the horizontal line that separates it from the stack frame below.

The result is that the top stack frame gets vaporized, leaving `main()`'s stack frame open to the outside air. And that is exactly what we want, because it's `main()`'s turn to take over now. Note that all the heap stuff is still there: objects on the heap have an unlimited lifespan, you'll remember.

The other thing the `return` statement does, of course, is put the function's return value in the proper place, just before it's nuked. In this case, since our original line of code was:

```
        ArrayList yankees = buildDaTeam();
```

it makes `yankees` refer to the object that was "returned," namely the `ArrayList` that the shortly-to-die `team` variable is pointing to.

We run one more line of code just to show we can do something with the returned object ("`int rosterSize = yankees.size();`") and the final result is as in Figure 4.9. There's no record of us having called a function at all – `buildDaTeam()` simply did its job dutifully and quietly, and `main()` gets to reap the result.

Calling a function from a function

By the way, this point is probably obvious by now, but let me clarify anyway: if you call a function, and that function *itself* calls *another* function, the same thing happens. The second function gets its own stack frame with its own variables, while both the first function and `main()` both get put on pause. There are at that moment *three* stack frames. When the second function returns, its stack frame disappears and the first function becomes active; and when the first function returns, *its* stack frame disappears and `main()` becomes active.

The terminology we use to describe this is somewhat obscure: when we create a new stack frame for a newly-called function we call it **pushing** a new frame on the stack. When we return, and get rid of it, we call it **popping** the frame off the stack. Push and pop are lingo you'll see in Data Structures class, when a data structure called a "stack" is introduced. That stack data structure is a more general category of memory management technique, of which "the stack" of our present chapter is an example.

Anyway, this whole push-a-frame-every-time-you-call-a-method thing (and pop-the-top-frame-every-time-you-return thing) is central to how any computer program operates. It's how your program breathes.

```
class Ballplayer {
    String name, position;
    int uni, numHits, numAtBats;

    Ballplayer(String name) {
        this.name = name;
        numHits = 0;
        numAtBats = 0;
    }

    void strikeout() {
        numAtBats++;
    }

    void getAHit() {
        numHits++;
        numAtBats++;
    }

    double getBattingAverage() {
        return ((double) numHits)/numAtBats;
    }
    ...
}
```

Figure 4.10: Part of the `Ballplayer` class.

4.3 Calling methods

The mechanics of calling a function are just the same as when calling a method, except for one thing: `this`. It turns out that when you call a method on an object, you're adding one more thing to the stack: a reference to the object the method was called on. And that, of course, is precisely what "`this`" means.

Let's pan over to a different part of our fictitious baseball simulator: the `Ballplayer` class itself. Part of the code for it is in Figure 4.10.[6]

We're going to have a different class for pitchers, since they have dif-

[6]Apologies to non-baseball fans. All you really need to understand this example is that in baseball, every batter accumulates a number of "at bats" (chances to come to the plate and hit against a pitcher) and a number of "hits" (times he/she actually hit the ball and made it at least to first base). A player's "batting average" is the hits over the at bats; it ostensibly tells you how likely (on a scale of 0 to 1) that player is to get a hit if he/she bats.

```
class Pitcher {
    String name, handedness;   // L or R
    int uni, numKos;
    double koDominance;        // between 0 and 1
    static java.util.Random rng = new java.util.Random();

    ...
    void face(Ballplayer batter) {
        double koRandNum = rng.nextDouble();
        double batterRandNum = rng.nextDouble();
        if (koRandNum < koDominance) {
            batter.strikeout();
            this.numKos++;
        } else {
            if (batterRandNum < batter.getBattingAverage()) {
                batter.hit();
            } else {
                batter.strikeout();
                numKos++;
            }
        }
    }
}
```

Figure 4.11: Part of the `Pitcher` class.

ferent stats (see Figure 4.11).[7] The only method we'll show on the `Pitcher` class is `.face()`, which is where a pitcher "faces" (pitches to) a batter in our simulation. The result will either be strikeout or a hit in our extremely simplified view of the baseball world.

One item of note is the `static` variable `rng`, which stands for **r**andom **n**umber **g**enerator. It's an instance of the `java.util.Random` class, which the Java API provides to roll random numbers. Every time you call `.nextDouble()` on a `Random`, it generates a new random number between 0 and 1. It makes sense for this to be a `static` variable, since the random number generator itself is an object that all objects will share and use.

The specifics of the `.face()` algorithm aren't important to understand. What is important is what happens in memory as this

[7]Here, we're going to model each pitcher as having a "`koDominance`" ("KO" is baseball lingo for "strikeout," btw). This is a number between 0 and 1 indicating the probability of overwhelming the batter with a strikeout without that batter being able to do anything about it.

```
    public static void main(String args[]) {

        Ballplayer joltinJoe = new Ballplayer("Joe Dimaggio");
        joltinJoe.setUni(5);
        joltinJoe.setPosition("OF");

        Ballplayer theSayHeyKid = new Ballplayer("Willie Mays");
        theSayHeyKid.setUni(24);
        theSayHeyKid.setPosition("OF");

        Pitcher bestOfAllTime = new Pitcher("Sandy Koufax");
        bestOfAllTime.setUni(32);
        bestOfAllTime.setHandedness("L");
        bestOfAllTime.setKoDominance(.5);

        bestOfAllTime.face(theSayHeyKid);
    }
```

Figure 4.12: A mighty showdown on the diamond.

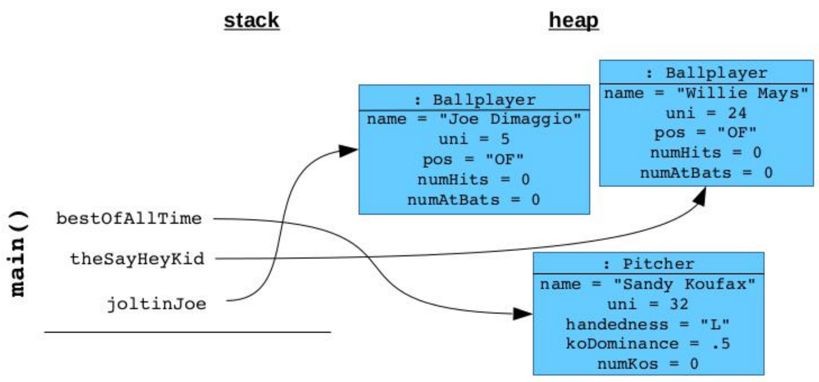


Figure 4.13: The baseball simulator's memory immediately before executing the *last* line of `main()` ("`bestOfAllTime.face(theSayHeyKid)`").

method is called. Let's say our `main()` has the code in Figure 4.12. After executing all lines but the last one, we have the picture in Figure 4.13. Take a moment and convince yourself it's correct in all details.

And now for the moment we've all been waiting for: the first pitch of a new (fantasy) baseball season, in which Sandy Koufax, the great-

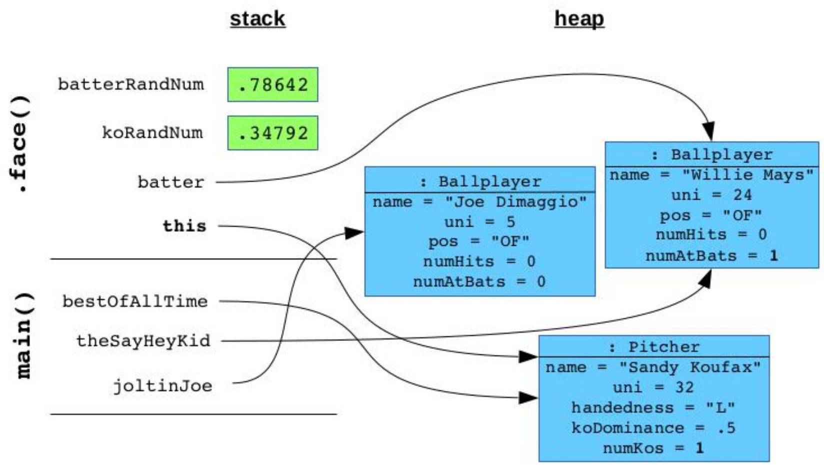


Figure 4.14: Memory while on the last line of `.face()`.

est pitcher of all time, will face down Willie Mays, quite possibly the greatest hitter of all time. I can't stand the suspense!!

Figure 4.14 shows how memory looks during this thrilling matchup. We're inside the `Pitcher`'s `.face()` method, and so it has its own stack frame as expected. But I want to draw your attention to two crucial aspects of this diagram:

1. First, notice we have a visitor. On the stack frame, in addition to the other expected variables, is none other than "`this`". Realize that `this` is really just a reference variable like any other. What does it refer to? *The object the method was called on*, of course. In this case, it's Sandy Koufax. How do we know? Because we didn't just say "`face(theSayHeyKid)`" but "*`bestOfAllTime`*`.face(theSayHeyKid)`". So the object that `bestOfAllTime` refers to will be pointed to by `this` while we're inside the method. Ponder this deeply.

2. Second, recognize that the `batter` argument – which is a reference variable of type `Ballplayer` – is referring to the *same* object that `theSayHeyKid` is pointing to back in `main()`. It is emphatically ***not*** a copy of the object. That's critical, be-

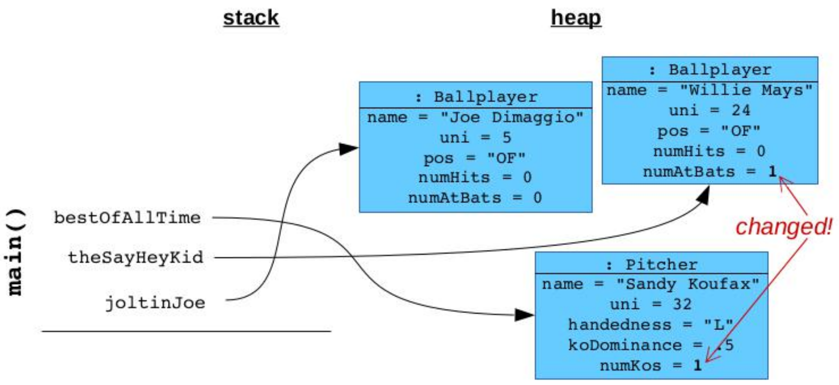


Figure 4.15: The final memory picture. Note the changed inst vars!

cause otherwise our `.face()` method would have no way of updating Willie Mays' stats as a result of this confrontation.[8]

Drum roll, please, before we hear the announcer: *"...and it's a scorching four-seam fastball from Koufax: swing and a miss, strike three!"* The way the random numbers turned out in this example, Koufax was so overpowering that he struck out Mays without the latter having a fighting chance. (See how `koRandNum` was less than Koufax's `koDominance`, so he blew him away without the `else` statement coming into play – *i.e.*, without Mays' batting prowess even having a chance to shine).

Don't worry, Willie: maybe you'll get one of your 660 lifetime home runs next time you're up to bat. Console yourself with this: a different "Willie" Hall of Famer (Willie Stargell) once quipped, "trying to hit against Sandy Koufax is like trying to drink coffee with a fork."

The last diagram of the chapter, Figure 4.15, shows the situation when we return to `main()`. It's important not to miss the main

[8]If you learned these terms in 220, this point can be equivalently stated as follows: "Java uses **pass-by-reference** for objects, not **pass-by-value**." (Java *does* use pass-by-value for *primitive types*, as we've seen: `int`s and such have their own presence on the stack, and so are *copied* from stack frame to stack frame.)

point here: both objects (`Pitcher` and `Ballplayer`) have their stats updated as a result of this showdown. If you're coming from a language like C++, which passes objects by value, you might be raising your eyebrows right about now. Get used to it. In Java, passing an object to a function/method makes *that exact object* available to the function/method, not a copy. And certainly `this` is a reference to the very object the method was called on, not a copy of it. This turns out to be almost always what we want.

Chapter 5

Exceptions

Let's go back and revisit our `.drive()` method from the `Car` example on p. 54. It looked like this:

```
class Car {
    ...
    void drive(int numMiles) {
        double galsBurned = numMiles / gasMileage;
        galsRemaining = galsRemaining - galsBurned;
        odo += numMiles;
    }
}
```

The shrewd reader (and driver!) will realize that this method is a bit optimistic: when told to drive `numMiles`, it blindly does so, even if there's not enough gas to get that far. We ought to guard against this kind of wishful thinking by *not permitting* a drive that's outside our range. If told to drive 1000 miles when we only have enough gas to go 200, we'll just say "no." That's way better than ending up with a negative gas tank!

The first step in implementing this kind of defensive coding is to figure out *when* to refuse to carry out orders. That's not too hard in this case: our local `galsBurned` variable is exactly what we need: if it turns out to be higher than the gas remaining in the tank, we are officially in Nonsense Land. A simple `if` statement can take care of that.

5.1 Bad ways of handling error conditions

The second step is figuring out *what to do* when this occurs. Most people's first instinct is to blare out a siren:

```
// Inadequate approach #1
class Car {
    ...
    void drive(int numMiles) {
        double galsBurned = numMiles / this.gasMileage;
        if (galsBurned > this.galsRemaining) {
            System.out.println("Not enough gas!!");
        }
        this.galsRemaining = this.galsRemaining - galsBurned;
        this.odo += numMiles;
    }
}
```

This is done in the hopes that someone will hear us and be alarmed. The problem is, this error will go to the console, where it may or may not ever be seen; and even if someone notices it, we've still already done the dirty deed. We have a `Car` object with an **illegal state**: a negative gas level.

Slightly better, but still not good enough, is to print the error and also refuse to carry out orders:

```
// Inadequate approach #2
class Car {
    ...
    void drive(int numMiles) {
        double galsBurned = numMiles / this.gasMileage;
        if (galsBurned > this.galsRemaining) {
            System.out.println("Not enough gas!!");
            return;                        //  <---  NOTICE THIS!
        }
        this.galsRemaining = this.galsRemaining - galsBurned;
        this.odo += numMiles;
    }
}
```

Now, in addition to printing the error, we also `return` from the method prematurely, instead of carrying out the nonsensical operation.

The problem with this approach is that *the client code is not alerted that anything went wrong.* We'll see some actual client code in action in the next section, but for now just realize that whoever called `.drive(1000)` is none the wiser. The client merrily chugs along thinking that the thousand-mile drive was plain sailing, oblivious to the fact that no such drive actually occurred.

5.2 The right way: `throwing` and `catching`

The right way to handle this is with Java's `Exception` mechanism. We don't return prematurely, as above; instead, *we don't return at all.* Exceptions are Java's way of allowing a method *not* to return, but rather to raise a big red flag to indicate that carrying out the method just flat didn't work. It's the only responsible thing to do.

Our first step is to "throw an exception" instead of returning. Here's how:

```
// Correct approach (not finished yet)
class Car {
    ...
    void drive(int numMiles) {
        double galsBurned = numMiles / this.gasMileage;
        if (galsBurned > this.galsRemaining) {
            throw new Exception("Not enough gas!!");  //  NOTICE!
        }
        this.galsRemaining = this.galsRemaining - galsBurned;
        this.odo += numMiles;
    }
}
```

Operationally, throwing an exception has the same immediate effect as returning: the method instantly terminates and returns control back to the client code. The differences, as we'll see in the next section, are that the client code is aware that something unusual happened, does not get a return value, and can take evasive action.

If you try to compile the above code, though, you get an error, which says:

```
error: unreported Exception; must be caught or declared to be thrown
```

It's actually nice of Java to give this error, and here's why. Inside our method, we've created a possibility that we won't return at *all*, and will instead barf because of an insoluble problem. Java requires that if we do that, we 'fess up and declare that this is a possibility. That prevents unwitting programmers from blithely calling our method and not accounting for the fact that it might not even run to completion.

Fixing it is simple; we just change the first line of the function to:

```
    void drive(int numMiles) throws Exception {
        ...
```

That first line now says: "you can call me on a `Car`, pass me an integer argument, and get no return value. *But* there's a possibility that it won't work, and you should be aware of that." It's only honest, and as we'll see, it allows the code that uses `Car`s to properly deal with the problem.

5.3 Calling a method that `throws Exception`

The only remaining fly in our ointment (don't worry; he's easily swatted) is that when client code *calls* `.drive()`, it might not necessarily run to completion. In fact, if we compile the above main program, we'll get the same kind of compile error that we did when we were midway through implementing the Exception-throwing stuff. It'll say we're being unconscionably remiss by refusing to deal with the error that might occur any time we tell one of our cars to drive.

The Java way to handle this is with a **try/catch block**. Essentially, this just builds a little scaffolding around our call to suspicious methods like `.drive()` so that if an exception is indeed "thrown" when we call it, we can "catch" is and do something sensible. Here's what the code looks like:

```
    public static void main(String args[]) {
        ...
        // Caravan to Disneyworld!  (Grammy's meeting us there.)
        minivan.fillUp();
        try {
            minivan.drive(500);
        } catch (Exception e) {
            System.out.println(e);
            System.exit(1);
        }
        ...
    }
```

Instead of simply calling `minivan.drive()` and throwing caution to the wind, we put that code in a **`try` block**. The code in a `try` block is executed normally, step-by-step, just like anywhere else in a Java program. But the `try` block has one or more (here just one) contingency plans connected to the bottom of it, which can handle any special (or "exceptional") conditions. In this case, that call to drive the minivan 500 miles through traffic will either work in its entirety and have the desired effect, or it will abort in the middle of it and control will be immediately transferred to the relevant catch block below it. The flow continues in that **catch block**, in this case printing the message of the `Exception` and then terminating the program.

What to do in each exceptional situation depends on the situation itself. Sometimes, there are meaningful things one can do in response to an error: like if a network connection fails, the code can retry connecting; or if a checking account withdrawal fails, the system can cut over to the savings account and cover the amount from those funds instead. In our case, if our program tracked things like routes and desired destinations, a caught exception would indicate that we need to find a new, temporary destination other than the one we're currently seeking: one that has gas so we can fill up.

Once all the method calls that are defined as "`throws Exception`" have been enclosed in try/catch blocks that at least nominally handle the errors, the code compiles again and it can hopefully run error-free.

5.4 Stack traces

One more thing before we leave this chapter on exceptions. It often proves useful to look at an `Exception`'s **stack trace**, which is accomplished by simply calling `.printStackTrace()` on the `Exception`, often in the `catch` block. For instance, suppose we change the previous try/catch block to this:

```
        try {
            minivan.drive(500);
        } catch (Exception e) {
            e.printStackTrace();           <--- NOTICE THIS LINE
            System.exit(1);
        }
```

Now, what happens if there isn't enough gas in the old minivan to drive 500 miles? Here's what:

```
java.lang.Exception: Not enough gas!!
    at Car.drive(Car.java:37)
    at Car.main(Car.java:57)
```

At first, this output is an eyeful, and causes some students to look away in horror. But I'm going to encourage you to be brave and look again, and realize how much useful information is here. This output is called the exception's **stack trace**, by which we mean a readout of *what the stack looked like at the moment the exception was thrown.*

Just as we saw in Chapter 4, Java maintains a stack of frames, one for each method that's in progress. Whenever a method returns, the top stack frame gets popped off the top; whenever a method is called, a new stack frame gets pushed on the top. At any point in time, the state of the running program is nicely captured by its stack.

One of the most helpful debugging tools you could imagine would be a precise readout of the exact line of every function the program is currently "in" at the instant the error occurred. And this is exactly what the stack trace is. The above output tells us:

1. The exception was thrown on line 37 of `Car.java`. This was in the `.drive()` method, which was called from...
2. ...line 57 of `Car.java`. This was in the `main()` method.

In this case, there were only two active methods when the exception was encountered. Other cases are of course more complicated. Suppose your stack trace looked like this:

```
BattleException: Power not charged.
    at Hyperbeam.useAgainst(Hyperbeam.java:24)
    at Snorlax.attack(Snorlax.java:43)
    at Pokemon.takeTurn(Pokemon.java:181)
    at Simulator.battle(Simulator.java:215)
    at Simulator.main(Simulator.java:33)
```

Without even knowing anything about how this program works, you can tell a *ton* about the circumstances in which it crashed. Specifically:

1. An exception was thrown on line 24 of `Hyperbeam.java`. This was in the `.useAgainst()` method, which was called from...
2. ...line 43 of `Snorlax.java`. This was in the `.attack()` method, which was called from...
3. ...line 181 of `Pokemon.java`. This was in the `.takeTurn()` method, which was called from...
4. ...line 215 of `Simulator.java`. This was in the `.battle()` method, which was called from...
5. ...line 33 of `Simulator.java`. This was in the `main()` method.

This is usually your first line of defense when debugging: read the stack trace of the `Exception`, reconstruct exactly what was going on when the error occurred, and then set about figuring out why it was caused.

By the way, you can also print a stack trace from any place in your code, even if no error condition has arisen or exception is thrown. If you just type:

```
            ...
            new Exception().printStackTrace();
            ...
```

anywhere you please, then whenever that line of code is encountered, a stack trace like those above will be spewed to your screen. It's often quite helpful when you're scratching your head saying, "okay, I know something's going wrong in this method... ...but how did I even *get* here?" The stack trace tells you exactly how.

Oh, I should also mention that these stack trace outputs are a Java thing. C++'s equivalent output for many of these kinds of error cases is simply this:

```
Segmentation fault (core dumped)
```

I think you'll agree, that's not nearly as helpful. ☺

Chapter 6

UML class diagrams

We spent last chapter discussing the **dynamic** view of a program: what happens to memory, step by step, as it unfolds. In this chapter, we'll switch to a **static** view: long-term, what are the program's classes, methods, and relationships between them?[1]

If there's a type of UML diagram that deserves the name "blueprint," it's the **class diagram**. Class diagrams depict a high-level, stable perspective of a software system. When you want to figure out how a large OO program works, or when you're tasked with implementing a system that someone else has designed, the first thing you look at are its class diagram(s).

UML class diagrams contain a number of elements, each of which has a very specific meaning. We'll cover each in turn.

[1]The words "dynamic" and "static" are ubiquitous in computer science, and mean a zillion different unrelated things. For example, we've already seen the Java "`static`" keyword, and how it indicates class-level rather than an object-level ownership. We've also hinted at the stack having "statically-allocated memory" and the heap being "dynamically-allocated." These terms are ***unrelated*** to our use of the words in this chapter. At present, by "dynamic" we mean "the contents of memory changing as the program runs"; and by "static" we mean "the consistent, permanent characteristics of a program, quite apart from how it might be behaving at any moment, which include its classes, methods, and associations."

6.1 Classes

Unlike memory diagrams, which depict objects, class diagrams contain classes (duh). We've already seen what a single class looks like in section 3.2 (*e.g.*, the left side of Figure 3.1.) Most class diagrams contain many such classes. Recall that each class has three compartments, containing the class's name, its inst vars, and its methods, in that order.

By the way, one flexible (yet slightly annoying IMO) aspect of UML is that it allows **varying levels of detail**. In other words, on a particular diagram, you may or may not want to show all the instance variables and methods, because it may or may not be relevant to the purpose of that particular diagram. Similarly, you may or may not want to show all the aspects of each inst var or method; perhaps it's too early in the design process to completely specify all the parameters and return types, for example. To illustrate, all three pictures in Figure 6.1 are legit ways of representing the `Car` class. We can include as much or as little detail as we please.

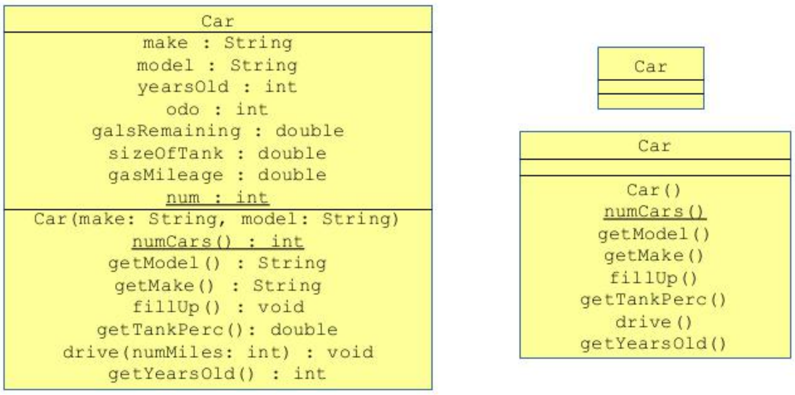


Figure 6.1: Three equally valid ways to draw the `Car` class on a class diagram, depending on how much detail it makes sense to include.

The reason I find this annoying, by the way, is that it's ambiguous. If you see no inst vars in the second box, does that mean (a) that class *has* no inst vars, or (b) the designer didn't think it was relevant to include them on this particular diagram? No way to really know.

6.2 Associations

Perhaps the most important bits of information on a class diagram are the **association**s between classes. An association means that two classes collaborate together in some way to achieve some larger purpose. It is indicated on a class diagram by a line connecting the two classes. Different types of lines represent different kinds of relationships between the classes. It's important not to mix them up, because if you do, you're dictating something incorrect to the programming team about how the classes are intended to work.

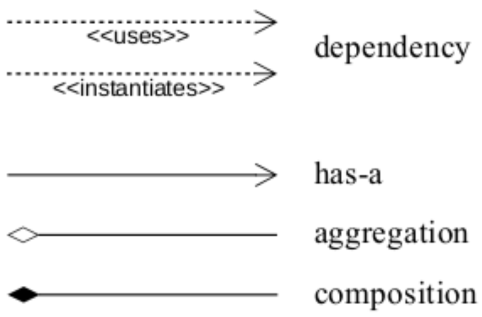


Figure 6.2: Diagrammatic elements for different association types.

Dependency associations

Figure 6.2 shows some of the UML association arrows and their meaning. (There are others we'll get to in future chapters.) The dashed line with a crow's foot arrowhead is called a **dependency**, and is the "weakest" of the association types. When I say weak, I mean that the relationship between the two classes isn't as important, nor as permanent, as with the other association types we'll discuss later.

A dependency between classes `A` and `B` can be thought of in a couple of ways:

- One or more methods of the `A` class will *call methods on* a `B` object.
- The `A` class *is dependent on the interface of* the `B` class.

The word **interface** – like stack, heap, dynamic, static, and many other computer science words – has multiple meanings. We've seen one usage already, in Figure 2.4 (p. 30). We'll talk about the Java `interface` keyword later in the book. For now, when I say interface I mean *those aspects of a class that a user of that type of object can see.* This boils down to: the methods you can call on it, together with their argument lists and return types. Specifically, the interface does *not* include the method implementations (the bodies of the methods), nor the instance variables.

If you think about it, you'll realize why the above two bullet points are actually equivalent. Suppose some class `A` method has this line of code in it: "`String s = B.scissorKick(15)`". Then clearly the code in the `A.java` file is *dependent* on the fact that class `B` has a `.scissorKick()` method, and that it takes an integer, and returns a `String`. If any of that ever changed in the `B.java` file, then class `A` would be impacted.

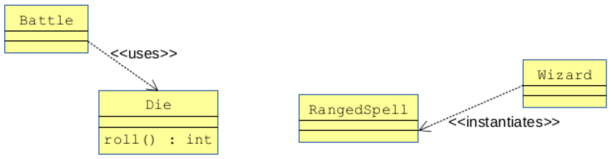


Figure 6.3: Examples of dependency associations.

The strange-looking words adjacent to the dependency arrows in Figure 6.2 go by the even stranger-sounding term **stereotypes**. A stereotype in UML is an extra bit of information that enhances part of a diagram (an association arrow, as here, or sometimes a class, method, or other element) by making its meaning more precise. Stereotypes are usually displayed enclosed by double-wakkas ("≪...≫").

In the case of dependency associations, the stereotype "≪uses≫" means pretty much what a dependency always means: that the designer intends class `A` to "use" (*i.e.*, get its hands on, and call method(s) on) object(s) of class `B`. The "≪instantiates≫" stereotype goes a bit further, and implies that some method of `A` will *instantiate* B objects in addition to merely calling methods on them.

The examples in Figure 6.3 are from a Dungeons & Dragons type combat simulator. A Battle object represents a fight between adventurers and monsters. While simulating this fight, a Battle will make use of one or more Die (singular of "dice") objects to roll random numbers that determine the outcome. This is a "≪uses≫" association, since Battle's code now depends on Die's interface not changing.

Elsewhere in the program, wizards sometimes cast ranged spells, like fireballs or lightning bolts, to damage distant enemies. So in the simulator, a Wizard object might instantiate a RangedSpell object to carry out this attack. Since somewhere in the Wizard class's code there will be a "new RangedSpell()" line, we say that Wizard ≪instantiates≫ RangedSpell.

Dependencies in code

Now what would we expect to see in the code that would reflect this kind of association? In the "≪uses≫" case, we expect to see one or more methods of the A class making method calls on B objects. Perhaps something like this:

```
class Battle {
    ...
    void resolveAttack(Adventurer a, Monster m, Die d) {
        ...
        if (d.roll() < a.currentWeapon().attackStat()) {
            ...
        }
    }
    ...
}
```

The design diagram doesn't specify exactly what A method will be called where, just that method calls are expected. This communicates something important to the programmer.

For "≪instantiates≫", we'd expect to see the word new somewhere in A:

```
class Wizard {
    ...
    void takeAction(ArrayList<Monster> enemies) {
        ...
        if (enemies.size() > 3) {
            RangedSpell fireball = new RangedSpell("Fireball", 60, 12);
            fireball.cast();
            ...
        }
    }
    ...
}
```

“Has-a” associations

The next strongest type of association has a bizarre name: it’s called “**has-a**.” We denote it with a solid arrow between classes, with a crow’s foot on one side or both.

When class `A` has-a class `B`, that is nearly always a signal to the programmer that `A` should have an *instance variable* of type `B`.[2] In other words, not only does an `A` object call methods on a `B` (as in the dependency association), but an `A` object actually holds on to one (or more) `B` objects for the long-haul.

Now in some cases, the “has-a” verbiage makes perfect sense. If our Domino’s Pizza delivery manager application had a `Pizza` class and a `Topping` class, it would be no-brainer to say that every `Pizza` has-a `Topping`. It conjures up in our minds a picture of containment, or ownership. Perfect. However, we also use this type of association in cases where containment doesn’t make sense at all.

For example, in the same application it would be quite sensible to say that “every `Pizza` has-a `DeliveryCar`.” But obviously the delivery car isn’t “inside” the pizza in the same physical way that the toppings are inside it. So what does it mean then?

The key is making sure you have the right mental model. Figure 6.4 shows both the wrong, and the right, way to envision a has-a relationship (at least, in Java). In memory, there is *no* “con-

[2]Or perhaps a **collection** of `B` objects rather than a single `B` object, as we’ll see later in the chapter.

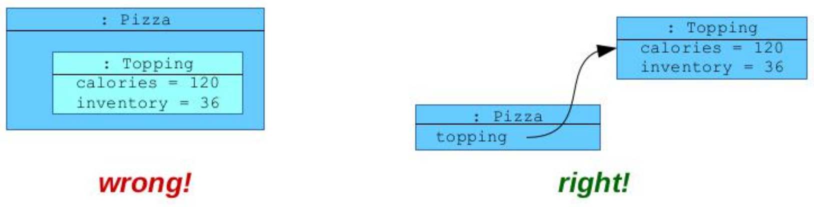


Figure 6.4: The wrong, and right, way to visualize a "has-a" association in Java.

tainment" as in the left-hand (wrong) image. The `Topping` object isn't enclosed inside the `Pizza`, or even exclusively owned by it. It's simply pointed to by one of the `Pizza` object's inst vars. The right-hand side of the figure is the correct one – and I daresay it's not problematic at all to think of a `Pizza` "having" a `DeliveryCar` in this way. All it really means is that a `Pizza` object "knows about" a `DeliveryCar`, which is the particular car that's delivering it.

Another reason that the correct mental model of "has-a" is important is that it is possible, and even common, for the association to go *both ways*. We use the term **navigability** for the question "which direction does the arrow go – from `A` to `B`, from `B` to `A`, or both?" When it goes both ways, we call it a **bidirectional** association.

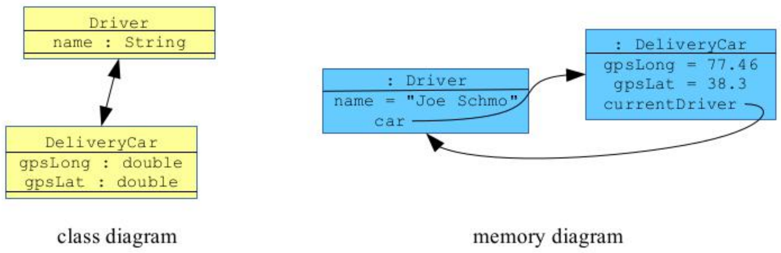


Figure 6.5: A bidirectional "has-a," depicted on a class diagram (left) and a memory diagram.

An example is the left-hand side of Figure 6.5. Here, our `Driver` class and our `DeliveryCar` class each know about the other, and in fact each hold on to an instance variable of the other type. If we viewed this `A`-having-an-instance-variable-of-type-`B` thing as the

`A` object *enclosing* the `B`, we'd blow a fuse. `A` would contain `B`, which would contain `A`, which would contain `B`, which... That way madness lies. But notice that nothing paradoxical happens at all in the corresponding memory diagram on the right-hand side of the figure. Each object points to the other, so that a `Driver` object knows which `DeliveryCar` he/she is driving, and a `DeliveryCar` also knows which `Driver` is driving it. No biggie.

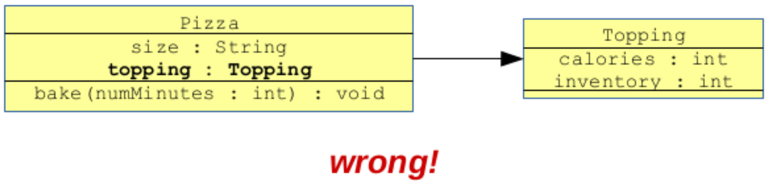


Figure 6.6: One incorrect way to model an instance variable. The "has-a" arrow *already* indicates that every `Pizza` has-a `Topping`: the extraneous `topping` entry in the `Pizza` class's second box is redundant and incorrect.

Note, by the way, that the has-a arrow implies the existence of the inst var *all by itself*. The class diagram should *not* contain a duplicate copy of the inst var in its second compartment. That would be redundant, and is considered an error (see Figure 6.6).

"Has-a" associations in code

Obviously instance variables are how "has-a" associations are manifested in a Java program. For `Pizza` and `Topping`, we'd see:

```
class Pizza {
    ...
    Topping topping;
    ...
}
```

and for our bidirectional `Driver`/`DeliveryCar`, we'd see both

```
class Driver {
    ...
    DeliveryCar car;
    ...
}
```

and

```
class DeliveryCar {
    ...
    Driver currentDriver;
    ...
}
```

These examples both assume that a `Pizza` has only *one* `Topping`, *etc.* If this isn't so, we'd use some kind of container class instead:

```
class Pizza {
    ...
    ArrayList toppings;
    ...
}
```

More on that later.

Aggregation associations

Continuing on towards the "stronger" end of the association continuum, an **aggregation** implies exclusive ownership of the object(s) in question. In other words, if `A` aggregates `B`, not only does it mean that `A` has an instance variable of type `B`, but that *no other `A` object also has that `B`.*

This is frequently misinterpreted, so let me expand on that. The "exclusivity" thing is a statement about *objects*, not classes. If `A` aggregates `B`, that does *not* mean that no other class can have an instance variable of type `B`. Rather, it means that if a particular `B` object is pointed to by an `A` object, no *other* `A` object also points to that `B`.

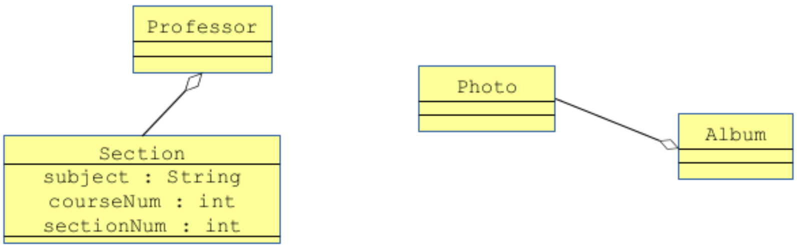


Figure 6.7: Examples of aggregation associations.

Examples appear in Figure 6.7. Note carefully: the diamond appears *on the "aggregator" side* of the arrow; *i.e.*, adjacent to the class that will have the instance variable. (I remember getting this backwards at first.)

In the first example, for a Banner-like college enrollment management system, each `Professor` will teach some number of `Sections` in a given semester. If Professor Jones is assigned to teach section 03 of BIOL 121, then no *other* professor is also assigned to that section. That's what the white diamond communicates.

In the second example, from a Facebook-like social networking site, users can arrange their `Photos` into `Albums`. As indicated on this diagram, a given `Photo` is *not* intended to simultaneously belong to more than one `Album`. (If we wanted to relax that constraint, and permit photos to belong to multiple albums at once, we would get rid of our white diamond and use a plain-old "has-a" arrow instead.)

Aggregations in code

Aggregation is intended to imply some sort of collection or ownership relationship between the two classes. However, in terms of the Java code that you initially write, *there is no immediate difference between an aggregation and a regular "has-a."* In both cases, you'll make an inst var of the appropriate type in the appropriate place. The code difference between aggregation and has-a won't come out until later, when the class methods are being implemented. That white diamond is more of a long-term signal to the programmer about how two classes are generally intended to operate together, rather than being a cue to write the first bit of code differently than you otherwise would.

Composition associations

The last association type we'll cover, and the most tightly-binding between classes, is called **composition**. It's a lot like aggregation (even the diamond syntax is the same, except it's black) but with one difference. With composition, not only does an `A` object have exclusive ownership over its `B` object(s), but there's a **lifespan dependency** as well: if the `A` ever disappears, its constituent `B`'s should also cease to exist.

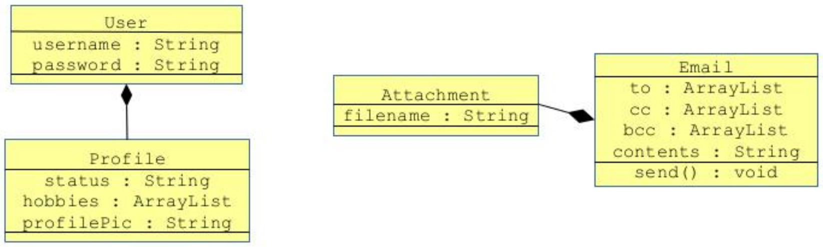


Figure 6.8: Examples of composition associations.

Consider the examples in Figure 6.8. In this social networking site, every `User` has a `Profile`. That `User` is the *only* one with that particular profile (hence this is at least aggregation) and what's more, *the `Profile` has no meaningful existence without its `User`.* If the user ever deletes their account, it wouldn't make sense to have a disembodied `Profile` object lying around, so it should automatically disappear as well. This lifespan connection is really the only difference between the white diamond and the black.

On the right-hand side is an example from some kind of email reader application (like Outlook, gmail, or Thunderbird). A user can compose an `Email` with some text and a list of recipients, and then add `Attachments` to it to send images, documents, code, *etc.* But what if the user decides to abandon the message before sending it?

The `Email` object should go away, but its `Attachment`s should too. Hence this is another example of composition.

Compositions in code

Just as with aggregations, there's no simple Java keyword that magically maps to the idea of "composition." Instead, the presence of the black diamond suggests to the programmer the intended function of the classes involved, and she will write the code with this in mind.

Association annotations

As if all this weren't enough, there are also a couple more syntactic things to learn about UML associations. An **annotation** is another mark on part of a diagram that gives more detail about how it is to be understood or implemented. We've already seen two examples of this: the stereotypes we included next to dependency lines are a type of annotation, as are the arrowheads to indicate navigability. We'll learn two more in this section.

Multiplicity

The **multiplicity** of an association indicates *how many* objects are involved in each concrete relationship. It's important to realize that even though multiplicity is shown on a class diagram, it's really a statement about *objects*.

Let's start with the left-most example in Figure 6.9. There we have two classes from a DMV software system, connected with a "has-a" association between `Driver` and `License`, navigable both ways. Note the numeral "**1**" annotation both sides of the arrow. This indicates that *every `Driver` "goes with" just one `License` object*, and *every `License` also goes with just one `Driver`.* This is called a **one-to-one association**, sensibly enough.

In the center example, on the other hand, we have a "$\star$" on the side of the arrow that connects to `Weapon`. In UML, the symbol "$\star$" means **zero or more**. So here's how we interpret this **one-to-many association**: every `Adventurer` has zero or more weapons,

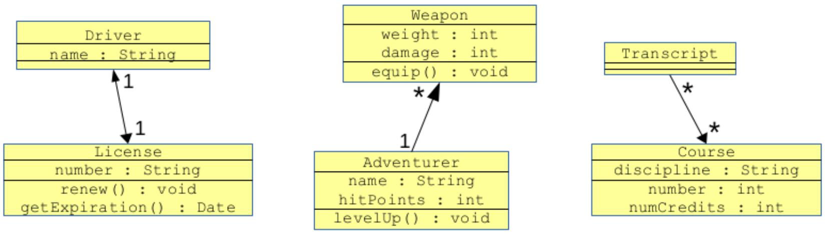


Figure 6.9: Association annotations indicating multiplicity.

while every Weapon is possessed by just one Adventurer. Note that since the direction is only navigable in one direction, this indicates that although an Adventurer is aware of which Weapons she owns, the Weapon objects are *not* aware of which Adventurer owns them. This knowledge (or lack thereof) is perfectly okay, and does not invalidate the meaning of the **1** or the ⋆ in the slightest.

Finally, on the right side, we have a **many-to-many association** between Transcript and Course. This says that every Transcript object is associated with potentially multiple Course objects, while each Course object appears on more than one Transcript. In terms of navigability, Transcripts maintain a record of which Courses they contain, but Course objects don't know which Transcripts they appear on (if any).

You'll occasionally see more elaborate multiplicity notations on class diagrams. The notation "**0..**⋆" means "zero or more"...which is of course exactly what plain old "⋆" means. The only reason for a designer to write "**0..**⋆" is for emphasis: she is stressing to the coding team that an object of the first type may well have *zero* objects of the second type at any given time; this is a real possibility. In contrast, if she writes "**1..**⋆" that means "one or more," which signals the coder "by the way, every object of the first type should *always* be assigned to at least one object of the second type; you should keep that in mind as you code." Even more rarely, you'll see multiplicities like "**5**" ("each object of type A is associated with exactly five objects of type B"), or "**3..8**" ("each object of type A is associated with anywhere from three to eight objects of type B"), *etc.* These are uncommon, especially since as we'll see in the next section, there really isn't any way to code those constraints explicitly in a language like Java.

Multiplicity in code

So what does all this look like in code? Well, first remember that inst vars are only used in the direction(s) along which the association is navigable. For Figure 6.9, this means that only `Driver`, `License`, `Adventurer`, and `Transcript` will have inst vars related to these associations; `Weapon` and `Course` will not. Furthermore, if the multiplicity is a **1**, the inst var will be of the type the arrow is pointing to; if it's a $\star$, it will be *some collection* of that type. Which sort of collection is used – an array, an `ArrayList`, a `Hashtable`[3], a `Set`, *etc.* – is normally up to the programmer, and is decided based on the run-time performance features of that collection type.

So here's some code we might reasonably expect to see from our three examples:

```
class Driver {                  class Adventurer {
    String name;                    String name;
    License license;                int hitPoints;
    ...                             ArrayList<Weapon> weapons;
}                                   ...
                                }
class License {
    String number;              class Transcript {
    Driver owner;                   Course[] courses;
    ...                             ...
}                               }
```

Here the programmer of the `Adventurer` class has chosen to use an `ArrayList` to hold each adventurer's weapons, while the `Transcript` author decided on a simple array. In terms of being faithful to the design, neither choice is right or wrong.

[3]See section 8.5 (p. 127) if you're unfamiliar with the `Hashtable` data type.

Roles

Our last type of association annotation has to do with **roles**. Sometimes, a design will be specific not only about the *existence* of the association between two classes, and about which-knows-about-which, and about how-many-are-involved, but also the intended *meaning* of the relationship. In other words, it may specify what role each of the object types is expected to play with respect to the other. This may sound a bit abstract, but some examples will make it clearer.

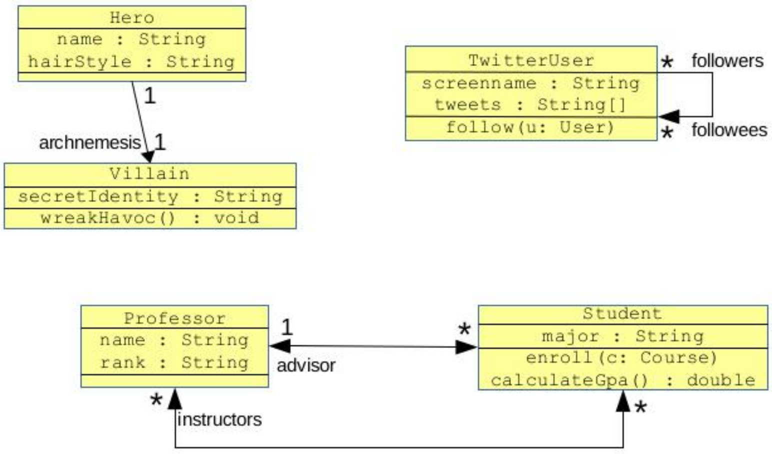


Figure 6.10: Association annotations indicating roles.

The upper-left example in Figure 6.10 shows a piece of a Marvel comic book database application. We have `Hero` and `Villain` classes, and a one-to-one association between them...but what does the association *mean*? If `Hero X` "goes with" `Villain Y`, does that mean that `X` has recently beaten up `Y`? That `X` admires `Y`? That `X` secretly *is* `Y`, unbeknownst to the public?

The word "archnemesis" next to the `Villain`-side of the arrow spells it out. It's called a **role name**. It tells us that in this relationship, the role that the `Villain` plays with respect to the `Hero` is that the former is the archenemy of the latter.

Moving to the right side of the diagram, we have an interesting situation involving only one class: `TwitterUser`. This class apparently has an association to itself! This turns out not to be as weird as it might seem. In fact, if you think about a social network like Twitter, the most meaningful relationships *are* between objects of the same class. And that's the key to de-weirding it in your mind: remember that an association is a statement about *objects*, not classes. We're not saying "`TwitterUser` has a relationship with itself" but rather "each `TwitterUser` is related to zero or more other `TwitterUsers`."

And what do those relations mean, you ask? The role name tells us: one of the users "follows" the other in the Twitter sense. In this diagram, we have role names on both sides of the arrow, although that's probably not strictly necessary. What is interesting here is the navigability of the association: according to the design, a `TwitterUser` object is aware of what other `TwitterUsers` follow him/her, but not which `TwitterUsers` he/she follows. If the design team decided they needed to track that separately, they'd need another arrowhead on the top side of the line.

Finally, the bottom example illustrates two *different* associations between the same two classes. This can happen as well. In this case, there are two distinct roles that `Professors` play with respect to `Students`: as their instructors (each student has several) and as their advisor (each student has one). The role names are imperative here, since otherwise the programming team would be lost as to why there are two relationships and what each one is supposed to mean.

Roles in code

Often, the role name on the diagram is simply used as the instance variable name in the code. For instance, I'd expect to see something like this:

```
class Student {
    String major;
    Professor advisor;
    ArrayList<Professor> instructors;
}
```

since those names were handed to us on a silver platter in the design diagram.

6.3 Visibility

The other parts of UML class diagrams that we'll annotate with extra information will indicate the level of **visibility** that the designer intends the various inst vars, methods, and even classes to possess. Visibility has to do with promoting **encapsulation**, the most important of all OO principles as we learned in Chapter 2.

This is one area where our UML diagrams, ostensibly programming-language-neutral, will betray a very Java-ish flavor. That's because in Java, there are four specific visibility levels for methods and inst vars (two for classes), each with a precise meaning, and we'll have UML syntax to indicate each. The complete list can be found in the tables in Figures 6.11 and 6.12. In both tables, the visibility levels are listed in order from most restrictive to least restrictive.

Java packages

Now's as good a time as any to mention the notion of Java "packages." This was a language innovation intended to provide an organizational mechanism: related classes can be grouped together into a construct called a **package**. This really isn't much more than being able to store `.java` files in different directories to keep them organized, except for one thing: the language itself is aware of which Java classes are members of which packages, and can enforce visibility based on that notion, as we'll see below. For now, here's the basics about packages:

1. Package names can be a single word (like "`combat`") or a dot-separated sequence of words (like "`com.gearbox.halo.simulator.combat`").
2. The dot-separated-sequence variety is kinda sorta meant to convey a hierarchy, from general to specific. In the previous example – from the *Halo* videogame created by Gearbox Software – "`combat`" is a subset of "`simulator`," which is part of the "`halo`" program designed at "`gearbox.com`". (Since "`com`"

is more general than "`gearbox`" – just like "`edu`" is more general than "`umw`" – many package names begin with a domain name written in reverse order like this.)

3. However, even though it looks like a hierarchy, *Java has no notion of subpackages.* In other words, although the "`com.gearbox.halo.simulator`" package looks like it would be a "subpackage" of "`com.gearbox.halo`," in actual fact it is not. It's just a naming convention, and there's no way (for example) to "import everything from `com.gearbox.halo` on down."
4. Every class is in one (and only one) package. This must be specified in *both* of two ways: (1) the first (non-comment) line of the file must be a **package declaration** like "`package com.gearbox.halo.simulator.combat`", and (2) the `.java` file itself must be physically in a directory called "`com/gearbox/halo/simulator/combat`."[4]
5. If a class has no `package` statement, then it is considered to be in "the default package," which just means "the package with no name."

Visibility levels

Okay, back to visibility. Let's look at the syntax and the operational implications of the different visibility levels in Figures 6.11 (for methods and inst vars) and 6.12 (for classes themselves).

visibility level	Java keyword	UML	visible to...
private	`private`	–	the class itself
package	(none)	~	any class from same package
protected	`protected`	#	the same package, or subclass[5]
public	`public`	+	any method anywhere

Figure 6.11: The four Java visibility levels for *methods and inst vars.*

[4]If you forget either one of these two things, or make them incompatible with each other, your code will be officially unreachable by any Java program. (Yes, that was a dumb design decision on Java's part.)

[5]A "subclass" has to do with the topic of **inheritance** in OO, which we will cover in gory detail in Chapters 11 and 12. For now, I just want to make the table complete.

visibility level	Java keyword	UML	visible to...
non-public	(none)	(none)	only classes in same package
public	`public`	+	any class anywhere

Figure 6.12: The two Java visibility levels for *classes.*

Now it's important to understand that unlike multiplicity, visibility modifiers make a statement about *classes*, not *objects.* Also, crucially, visibility is about the very *existence* of the method, inst var, or class, not its *value.* This is very commonly misconstrued, so let me clarify with an example.

Suppose a class diagram included the class in Figure 6.13. Here, for the first time, we see visibility modifiers in action. In particular, the `numHits` and `numAtBats` inst vars are both marked as private, while the `isBetterThan()` method is public.

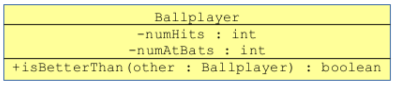


Figure 6.13: A class whose components bear visibility annotations.

Here's the kind of client code we want to make possible with this method:

```
...
Ballplayer jeter = new Ballplayer("Jeter");
Ballplayer arod = new Ballplayer("Rodriguez");
if (jeter.isBetterThan(arod)) {
    System.out.println("Sign Jeter to a zillion dollars!");
}
...
```

Let's inspect the *inside* of the `.isBetterThan()` method (*i.e.*, its implementation). Suppose it reads like this[6]:

[6]Apologies to baseball fans for the gross simplification of reducing an entire player's "goodness" down to his or her batting average. Of course in real life

```
class Ballplayer {
    private int numHits;
    private int numAtBats;
    ...
    public boolean isBetterThan(Ballplayer other) {
        double myBA = ((double)numHits)/numAtBats;
        double otherBA = ((double)other.numHits)/other.numAtBats;
        if (myBA > otherBA) {
            return true;
        } else {
            return false;
        }
    }
    ...
}
```

Now the key line I want to draw your attention to is the *second* line of the method. It reads:

```
double otherBA = ((double)other.numHits)/other.numAtBats;
```

My question to you, dear reader, is this: do you think this line ought to compile without errors, or no? Take a moment to consider your answer.

Many, many students assume this line will *not* compile cleanly. Here's their reasoning: "We're calling `.isBetterThan()` on a particular `Ballplayer` object (say, `jeter`). And we're passing another `Ballplayer` object as a parameter (say, `arod`). Now both `numHits` and `numAtBats` are marked `private` in the class. Therefore, `arod`'s values for these should be protected from, and unavailable to, the `jeter` object. It stands to reason that this will not be allowed. Otherwise, we'd be allowing one object to access another object's private data."

there are all kinds of other stats that come into play here – slugging percentage, base running stats, defensive ability, *etc.* – as well as impossible-to-quantify aspects like teamwork, inspiration, and clubhouse chemistry.

This sounds so eminently reasonable, and yet it is dead wrong. Here's why:

✗ A "private" inst var does <u>not</u> mean that one object's *value* is hidden from another *object*.

✓ A "private" inst var <u>does</u> mean that the very *existence* of one class's inst var is hidden from other *classes*.

In other words, it's not a "data privacy" thing like keeping your information inaccessible to creepy people online. Instead, it's a *code encapsulation thing* that prevents one class from making (and thereafter depending upon) assumptions about another class's design decisions. In terms of the online creep example, here's how I'd explain it:

✗ Making `SSN` (Social Security Number) a private inst var of the `Person` class does <u>not</u> mean that one `Person` object cannot find out another `Person` object's SSN.

✓ Making `SSN` a private inst var of the `Person` class <u>does</u> mean that `Dogs`, `Websites`, `CreditCards`, *etc.* don't even know that people *have* Social Security Numbers.

In yet other words, visibility is about *variables* and *classes*, not *values* and *objects*.

The code above does compile cleanly for one simple reason: it's a method of the `Ballplayer` class. Any method of the `Ballplayer` class can talk about any inst var or method of the `Ballplayer` class, regardless of which particular object is in view.

To complete the example, here's some code which indeed does *not* compile because of those private `numHits` and `numAtBats`:

```
class Team {
    private ArrayList<Ballplayer> roster;
    ...
    public void printRoster() {
        System.out.println("Name          Hits    ABs");
        for (Ballplayer b : roster) {
            System.out.println(b.name + "     " + b.numHits +
                "  " + b.numAtBats);
        }
    }
    ...
}
```

When we try to compile it, the `println()` statement inside the `for` loop barfs with:

```
Team.java:25: error: numHits has private access in Ballplayer
    System.out.println(b.name + "     " + b.numHits +
                                            ^
Team.java:26: error: numABs has private access in Ballplayer
        " " + b.numABs);
               ^
```

as we would expect. It's because the offending code is a `Team` method, not a `Ballplayer` method, and therefore cannot refer to any of `Ballplayer`'s private components (inst vars or methods).

The same mechanic is at play with methods as it is with inst vars: no code in a class can *call* a method unless it has visibility to that method, as specified in the rightmost column of Figure 6.11.

If you're wondering why it would ever make sense to have a private method, the answer is: as a helper method, for other (perhaps public) methods of that class to call internally. Having lots of short methods to perform basic tasks, but not exposing those methods outside the class, is one sign of a good designer.

Which visibility level to choose

Both inst vars and methods can technically have any of the four visibility levels assigned to them from the table in Figure 6.11. Here are the rules (and strong suggestions) to keep in mind:

1. Always make all instance variables `private`[7]. That's the easiest design decision you'll ever make. **Public instance variables unacceptably sacrifice encapsulation.**
2. Always make methods "as private as possible." This promotes encapsulation and reduces dependencies. When in doubt, err on the side of the *higher* entry in Figure 6.11, not the lower. If it turns out you must make it more accessible later on, you can always move its visibility lower on the chart without breaking anything. The reverse is not true.

Lastly, a word about package-level visibility. There may be design decisions you make (namely, certain methods you create on a class) that you don't necessarily want to make publicly accessible to all users of the class, yet which it does make sense to make available to the other classes that are collaborating with that class. Package-level visibility was designed for this purpose. Note that there is no Java keyword for it: it's the default. This is because Gosling & Co. (the designers of Java) were proud of the package concept and wanted to promote its use among Java developers as much as possible. So you have to explicitly type if you want any other choice. I think package-level visibility is a neat feature, but is underutilized.

Class visibility

As shown in Figure 6.12, the notion of visibility also extends to entire classes in Java. But it's simpler: either a class is public, or it's not. If it's public, any class anywhere can refer to it, and if it's "non-public" (yep, that's actually the term) it effectively has package-level visibility (*i.e.*, only other classes in its package can use it.)

Non-public classes thus play the same sort of role as private helper methods do: the public classes use them to help get their job done, but the non-public ones aren't designed to be directly instantiated (or even seen) by the outside world. Their use in practice is somewhat rarer than private methods, but I encourage their use.

[7]Or possibly `protected`, if you intend to inherit from the class. See Chapter 12.

6.4 Putting it all together

All right, let's close this chapter with a small but still full-blown class diagram that illustrates most of the above features. See if you can interpret all of Figure 6.14 correctly.

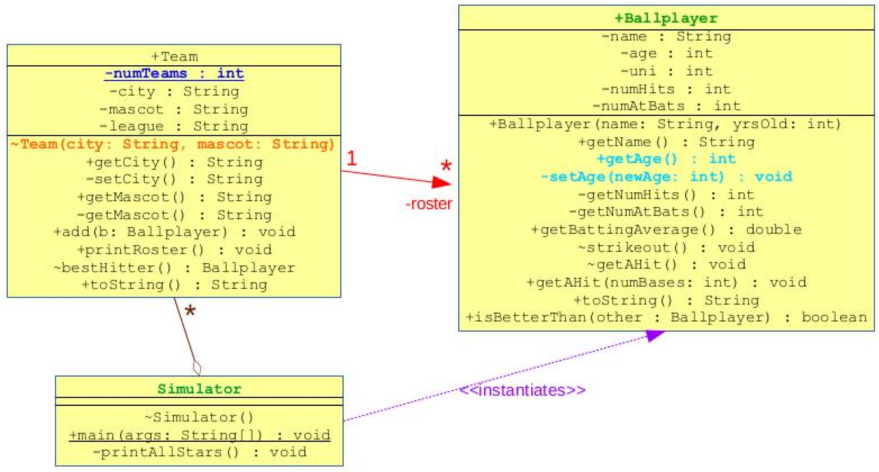


Figure 6.14: A full-blown class diagram. (The color is not part of UML; I only colored certain elements so I could refer to them in the text.)

Here's an incomplete list of things we know from the diagram. Each item's color corresponds to an item in Figure 6.14:

1. The `Ballplayer` class is public, and thus can be used by any class in any other package. The `Simulator` class isn't, though, and can only be referenced by classes in the same package.

2. Although anyone can use a `Team` object, only classes in this package can instantiate one. (And to do so, you must specify a city and a mascot.)

3. Any method that gets its hands on a `Ballplayer` can find out his/her age. But only methods of the `Ballplayer` class itself can change his/her age.

4. Every `Team` object will have a private instance variable[8] called `roster` which holds a collection (perhaps an `ArrayList`) of `Ballplayer` objects. Each of those `Ballplayer`s belongs to only a single `Team` object, but is not aware of which `Team` object that is (*i.e.*, `Ballplayer` objects don't have an inst var of type `Team`).

5. There is a single integer variable `numTeams` which is shared among all objects of type `Team`. It is not visible to any other class.

6. Somewhere in the static `main()` method of the `Simulator` class we would expect to find code like this: "`new Ballplayer( someName, someAge)`".

7. A `Simulator` holds on to some number of `Team` objects, probably in an instance variable, and each of those `Team`s belong only to it.

Did you pick all those things out? If so, you can read a blueprint, and I foresee many beautiful buildings in your future!

[8]Notice that the "–" immediately before the word "roster" is a visibility modifier, indicating that the inst var that results from this association will be private.

Chapter 7

The Singleton pattern

"Singleton" always sounded to me like the name of some small American town, maybe one where nobody ever gets married. But it's actually the name of our first (and easiest) *design pattern*, and the subject of this chapter.

7.1 Design pattern

You know how when you sit down to write some code, there are times when you think, "wait, I've written this before"? Programmer's déjà vu is commonplace, especially because certain tips and tricks end up working in a lot of different settings. For example, we've all seen how to go through an array and add up all its elements, or find its maximum value, or check whether it contains a particular item. You might think of these as "programming patterns." They're bite-sized, go-to solutions that can handle a myriad of common little programming scenarios.

It turns out that the same is true of design. Certain motifs in how classes collaborate with each other crop up again and again in different settings. They're important enough that they've been identified, described, and named.

The people who first promoted the idea of **design patterns** were Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm, who were thereafter nicknamed "The Gang of Four." (You'll hear many references in the software development industry to a "Gang of Four design pattern," sometimes abbreviated "GoF design pattern." This means one of the 21 named patterns that appeared in their hugely influential 1994 book *Design Patterns: Elements of Reusable Object-Oriented Software.* It's a book highly worth obtaining and reading.)

One thing that's great about this is that just by mentioning one of these agreed-upon pattern names – like "Observer," "Iterator," or "Strategy" – every developer worth their salt will instantly conjure up in their mind the mechanics of that particular pattern, and know immediately what kind of problem it's intended to solve. It saves a lot of words trying to describe an idea you know your fellow developer has seen before, if only you could get them to realize what you're talking about.

In this brief chapter, we'll cover the simplest GoF pattern of all: the **Singleton** pattern.

7.2 The Singleton pattern

The Singleton pattern is so simple it almost doesn't even deserve to be called a pattern. But it is. And it's easy to figure out when it applies to your situation: **a Singleton is used when you have a class for which you only ever want to instantiate one object.**

If you think about it, this kind of situation is pretty rare. Clearly any relevant program is going to need to instantiate lots of different `Car` objects, or `Ballplayers`, or `Professors`. There are occasions, though, when your class isn't so much a category as it is a special, one-of-a-kind object. Here are some examples:

- Part of your operating system may have a `PrinterManager` class that controls sending documents to various printers. The code will create many `Printer` objects, and many `Documents`, but only one `PrinterManager` which runs traffic control and routes print jobs to available printers.

- Your website that collects information about classic rock 'n' roll albums may have a `Database` class that represents the underlying data storage. You might get multiple `Connections` to it and instantiate multiple `Query` objects to search it, but there's just one `Database` as a point of contact.
- Most programs have some way of configuring them, usually by tweaking the values of various configuration variables. Your program could have a `Configuration` class from which the other software components can fetch the values of the settings as needed. There needs to be only *one* `Configuration` object, since they will all share access to a common set of settings.

The Singleton pattern does two things: (1) it ensures that only one instantiation is possible, and (2) it provides a global point of access to that one object, so that the rest of the code can get to it.

7.3 Implementation

Figure 7.1 (p. 114) shows what a properly-coded Singleton pattern looks like. It uses the `Configuration` example from above. Let's go through each part carefully.

Let's go through each part carefully. First, we have a ***`static`*** variable called "`theInstance`". Recall that `static` here means "goes with the class as a whole, rather than with each individual object." The reason for this is *the class itself will be holding on to its one-and-only instantiated object.* This is one of the few places we'll be using `static` stuff in this book, because we need to. If `theInstance` were *not* `static`, then the only way to get a hold of `theInstance` would be to have an instance of `Configuration` in the first place...which would defeat the purpose of the pattern.

Note that `theInstance` is also marked `private`. This is partially because of our rule "all inst vars should always be private, period," but also because making this variable accessible outside the class would make the whole pattern collapse. Parts of the code that needed access to the `Configuration` singleton instance would try

```
class Configuration {
    private static Configuration theInstance;

    public static synchronized Configuration instance() {
        if (theInstance == null) {
            theInstance = new Configuration(...);
        }
        return theInstance;
    }

    private Configuration(...) {
        ...
    }

    // The actual methods of the object. For Configuration, this
    //   might include something like:
    public String getParamSetting(String param) {
        ...
    }
}
```

Figure 7.1: A properly-coded Singleton pattern.

to grab `theInstance` and use it, but it might not have even been set to anything yet!

Next, we have the `instance()` method. This method is also `static`, so that it can be called on the *class* rather than on an object. And what does it return? A `Configuration` object...or perhaps I should say, *the* `Configuration` object since there's only ever going to be one.

Unlike `theInstance`, `instance()` is public. (Package-level visibility may also be an appropriate choice, depending on how wide your intended users of this Singleton are.) This is part of the public interface of the class, designed to be called by code external to the class.

The other word on the declaration line – `synchronized` – is probably foreign to you. Its purpose is beyond the scope of this chapter. Very briefly, "`synchronized`" prevents two different **threads** of execution from entering the `.instance()` method at the same time. A

multithreaded program is one that executes more than one flow of control simultaneously, each with its own stack. It turns out that if more than one thread was inside this method at the same time, we might accidentally instantiate *two* (or more) `Configuration` objects. For our single-threaded programs this isn't an issue, but it's good practice to get in the habit of making your Singleton `instance()` methods synchronized.

Now let's dive in to the code for `instance()`. It's very simple, as you'll see: all it does is say "if this is the first time anyone's ever called me, go ahead and instantiate an instance of me, and remember it (in the `theInstance` class variable). Then, return the one-and-only instance of me to the caller, to use to their heart's content."

This is called **lazy instantiation**: the only thing that will trigger the one-and-only `Configuration` object being instantiated is the first time anybody calls `Configuration.instance()`. If nobody ever does call it, then there won't ever be even one instance of this class created. But assuming someone does, a new `Configuration` object will be instantiated *this time only.* From that point on, all the subsequent times `Configuration.instance()` is called, that same object will be returned.

Then we have the constructor. It can do anything that any constructor can do, which varies widely depending on what kind of class this is. (For the `Configuration` example, perhaps it looks at the filesystem for a `.config` file, and if it exists, loads it and remembers all its contents in instance variables.) The important point to emphasize here is that *the constructor must be* ***`private`***). That's because if it weren't private, any old schmo could just write "`new Configuration()`" and get a *second* instance of the class, which is precisely what we want to avoid. Making the constructor `private` means nobody is allowed to instantiate a `Configuration` object...except for the `Configuration` class itself, which we saw in the `instance()` method.

7.4 Using the Singleton

This pattern allows any other part of the code base to do things like:

```
String bg = Configuration.instance().getParamSetting("bgColor");
```

Whenever we call "`Configuration.instance()`" we will get back a `Configuration` object. (Whether we realize it or not, it's the only such object that will ever exist.) There's no need to use use the word `new`; we just say "`Configuration.instance()`" every time we need it.

Other than this scaffolding, the rest of your Singleton class can do anything it wants. It will almost certainly have other (non-static) instance variables, and other methods to carry out its evil deeds. The "Singleton part" is just the `instance()` and `theInstance` members, together with the private constructor.

Singleton is often used in conjunction with the Factory pattern, by the way, which we will look at in a future chapter.

That's it. Told you it was easy! ☺

Chapter 8

Java odds 'n' ends

Before we continue our study of OOA&D proper, let's look at a few Java-specific idiosyncrasies which will be all up in our business soon enough.

8.1 Garbage collection

No, I didn't make that term up just to be funny. **Garbage collection** is actually the official name for a Java feature which was super innovative at the time, but which we now often take for granted.

Consider the code for a `Ball` class given in Figure 8.1. When we run it, Java calls `main()`, which calls `play()`. At the end of `play()`, right before it returns, a memory diagram would look like Figure 8.2. Take a moment to see if you agree with all the details.

At this moment, we have an active stack frame for the `play()` function which contains five variables of various types. And we're getting ready to return the reference variable `basket` back to `main()`, which means that about a nanosecond from now `main()` will be assigning its new `myBall` variable to point to that orange ball.

Okay, now let's do it. We return to `main()`. As soon as we do, the memory diagram looks like Figure 8.3. Take a close look. That second diagram is all correct, but something about it may strike you as a bit weird; namely, there are three objects on the heap *with*

```
class Ball {
    private String color;
    private int airPressure;

    Ball(String color) {
        this.color = color;
        airPressure = 0;
    }

    void bounce() {
        System.out.println("Boing!!");
    }

    static Ball play(int numBalls) {
        ArrayList equipment = new ArrayList();
        Ball b;
        int i;
        for (i=0; i<numBalls; i++) {
            b = new Ball("red");
            b.bounce();
            equipment.add(b);
        }
        Ball basket = new Ball("orange");
        return basket;
    }

    public static void main(String args[]) {
        int x = 3;
        Ball myBall = play(2);
        System.out.println("My ball is " + myBall.color + ".");
    }
}
```

Figure 8.1: A class to illustrate the utility of garbage collection.

nothing on the stack referencing them. The `myBall` variable dutifully points to the orange ball that was returned, but the two red balls, and the `ArrayList` that contained them, are now disembodied from everything else. And in fact, they are effectively *lost* to the program. There's simply no way to reference them.

If you're unsure that there's truly no way, ask yourself this question: "what line of code could we write to (say) change one of the `Ball` object's color from red to blue?" The answer is: there is no possible line of code we could write to do that. To even get off the ground we'd have to start with a *name*, and there is no name we could possibly use to get at either of those red `Ball` objects.

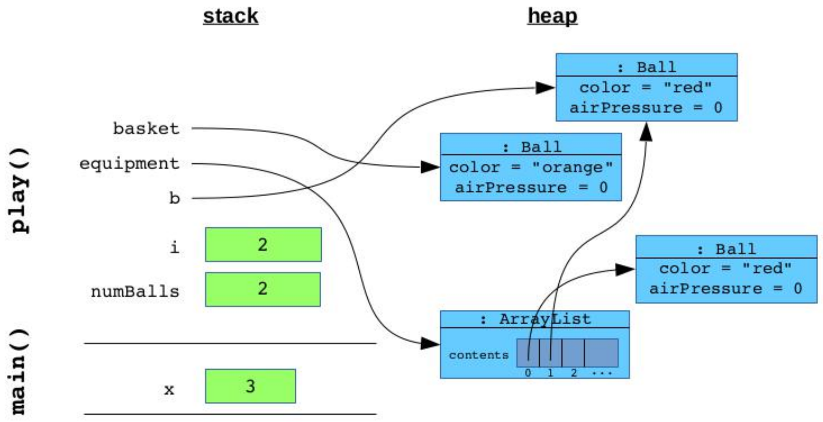


Figure 8.2: A snapshot of memory, taken just before the `play()` function returns back to `main()`.

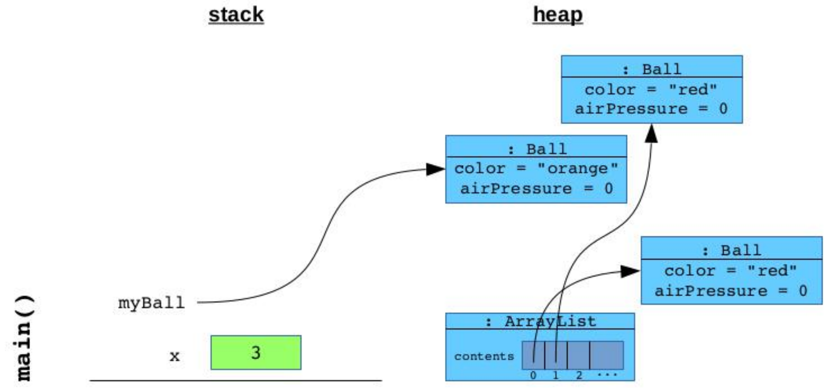


Figure 8.3: The state of memory right *after* the return to `main()`. Notice there are now three unreachable objects on the heap.

Now with C++, a language that preceded Java by decades, this would be a bad situation called (I kid you not) a **memory leak**. The memory the program used to store those now-unreachable `Ball`s is now inaccessible to the program, and what's worse, *C++ doesn't realize that's the case.* So those old objects just sit there, growing stale, occupying system memory that could be used to store other things instead. The program never realizes this, and so never reclaims that space. So the actual amount of memory the program has available to it has effectively shrunk.

In C++, the only remedy for this situation is for the programmer herself to keep track of which objects no longer have any stack references to them, and to explicitly `delete` those objects. This is a delicate task: fail to `delete` what is in fact delete-able and you'll have a memory leak; eagerly `delete` what actually does have other references to it and your program will crash when that memory is reused by something else writing over the top of it. The whole situation is fraught with peril.

Enter Java, in 1995. Java featured **automatic garbage collection** which outsourced the whole responsibility for this from the programmer to a special Java background task called the **garbage collector**. Whenever the garbage collector runs, it intelligently sifts through the contents of memory, looking for junk that can't be legally accessed anymore anyway. Whenever it finds such junk, it *automatically* tells the memory manager that that memory is no longer in use, and can therefore be repurposed the next time the program requests some memory with `new`.

Automatic garbage collection is lit. It means that pictures like Figure 8.3 aren't scary at all. Sure, we have three objects in the heap that can't ever be reached, but the garbage collector will soon run, figure that out, and reacquire that memory so it can be used again. All this without the programmer having to think a thing about it. Memory leaks are in principle a thing of the past.

As with most good things, there are downsides too. One downside to Java's approach is that the garbage collector thread decides to run "any time it darn well pleases." As developers, we don't ever tell Java to get off its butt and take out the trash (although there is a way to suggest this); rather, we just wait for it to run when it periodically thinks it needs to. This isn't normally a problem, but it can be in mission-critial real-time applications with absolute performance deadlines. Think of the software running on a pacemaker, which is embedded in a human heart. The code *must* respond in a certain amount of time in order to trigger the heart to take its next beat! Now if, during our pacemaker program's operation, the garbage collector suddenly decided to run in order to clean up lost memory, it might be a time-consuming operation in its own right. And our program could literally skip a beat (or beat later than it should) while waiting for it to finish. In situations like these, C++'s style of manual control does give us more fine-tuned flexibility over when exactly the reclamation of lost memory occurs. For most applications, though, we don't need that flexibility and Java's way of handling it is much appreciated.

8.2 The `import` statement

You've probably used `import` in almost every Java program you've ever written. Yet I've found most developers don't really understand what it does. Java itself is partially to blame here: the word "import" was a poor choice for this, since nothing is "imported" at all.[1]

Here's a statement that surprises a lot of Java programmers: you can write *any* Java program – even one that uses stuff in the Java API – *without* an `import` statement. The left side of Figure 8.4 is such a program.

No `import` required. Instead, on the left-hand side of the figure, every time we want to refer to the `ArrayList` class, we specified it as `java.util.ArrayList`. That does require us to type out the full name three times, but it turns out to be all Java needs to understand perfectly which `ArrayList` class we want to use.

Now since programmers (myself included) are lazy, and want to avoid typing when possible, the Java gods invented the `import` statement. The *only* thing it does is tell Java "I don't really feel like typing out `java.util.ArrayList` every time. That's a pain. So Java, please know that when I type `ArrayList` in this file, I really mean `java.util.ArrayList`."

The right-side of Figure 8.4 is exactly the same program, but now

[1] Also, the fact that it begins with a lower-case `i` just like C++'s "`#include`" statement reinforces this misconception – C++'s `#include` actually *does* include/import content.

```
class YouDontNeedImport {
  public static void main(String args[])
  {
    java.util.ArrayList celebs =
      new java.util.ArrayList();
    celebs.add("Kim Kardashian");
    celebs.add("Justin Bieber");
    celebs.add("Taylor Swift");
    p(celebs);
  }

  static void p(java.util.ArrayList l) {
    for (int i=0; i<l.size(); i++) {
      System.out.println("People love "
        + l.get(i));
    }
  }
}
```

```
import java.util.ArrayList;

class YouDontNeedImport {
  public static void main(String args[])
  {
    ArrayList celebs =
      new ArrayList();
    celebs.add("Kim Kardashian");
    celebs.add("Justin Bieber");
    celebs.add("Taylor Swift");
    p(celebs);
  }

  static void p(ArrayList l) {
    for (int i=0; i<l.size(); i++) {
      System.out.println("People love "
        + l.get(i));
    }
  }
}
```

Figure 8.4: The same Java program, using explicit inline package names (left), and the `import` statement (right).

using the `import` statement to avoid a little typing.

Whether the savings are worth it in any particular case is up to you. My point here is just to demonstrate that the statement "`import java.util.ArrayList`" does *not* do anything remotely like "go and find the `ArrayList` code in the `java.util` package, and bring it in here so I can use it." Nor is it true that "unless you import that class, you can't use it." Both are common Java myths.

Don't `import *`

By the way, you may have seen the use of "`*`" syntax, like this:

```
import java.util.*;
import com.google.search.engines.*;
import edu.umw.stephen.coolclasses.*;
```

This isn't a good idea. The reason is that it's ambiguous. Suppose in my code I refer to a class called `Scanner`. Which `Scanner` do I mean? Should Java assume I wanted to avoid typing "`java.util.Scanner`" or "`com.google.search.engines.Scanner`" or "`edu.umw.stephen.coolclasses.Scanner`"? It has no way of knowing. So if more than one of those packages defines a class with the same name (which is very possible), Java will at best not compile, and at worst give unexpected runtime behavior.

For this reason, using "*" in `import` statements is only acceptable when writing quick and dirty one-off code, not for anything that will stick around longer than your current coding session.

8.3 Java "generics"

Speaking of misleading terms, let me give you another one.

There are two ways to use a container class from `java.util` (like `ArrayList`, `Set`, or `PriorityQueue`). One way is to just create one without any syntactic fuss:

```
import java.util.ArrayList;
...
ArrayList stuff = new ArrayList();
```

and then put stuff in it:

```
stuff.add("Laundry");
stuff.add("Lunchbox");
stuff.add(new Car("Mazda", "MX-5"));
```

This is called a **heterogeneous** collection, because the things it contains are of different types (`String`s and `Car`s, this case).

Java allows this perfectly well, for reasons we'll understand in detail later. Here's the rub, though: when we get something *out* of the collection, we have to **cast** (or **typecast**, or **downcast**) it to the correct type before doing anything specific with it. This code, for example, does *not* compile:

```
// Doesn't work:
System.out.println("First item has " +
    stuff.get(0).length() + " letters.");
```

The reason is that we're calling `.length()` on whatever object happens to be at position `0` of the list, and Java can't know for sure whether that object will end up being a `String`, a `Car`, or something else. In particular, it can't know that that object will even *have* a `.length()` method. So Java makes us do this:

```
// Works, due to explicit cast:
System.out.println("First item has " +
    ((String) stuff.get(0)).length() + " letters.");
```

This says, "Java, I give you my word. I promise the thing at position `0` *will* be a `String`, and I'll stake my reputation on it. Please force it to be treated as a `String`, and if it turns out I'm lying, you have permission to embarrass me at runtime with a program crash."

Having to do that is a rather high price to pay for the flexibility of being able to store any old thing in `stuff`. That's why back in 2004, Java introduced the idea of **generics**, which allow you to restrict a collection to having only items of a single type. Here's how you do it:

```
import java.util.ArrayList;
...
ArrayList<String> stuff = new ArrayList<String>();
stuff.add("Laundry");
stuff.add("Lunchbox");
stuff.add(new Car("Mazda", "MX-5"));    // Compile error
```

Our `stuff` is no longer a plain old `ArrayList`, but specifically an `ArrayList<String>` (pronounced "array list of strings.") This tells Java, "please don't let me put anything into this `ArrayList` *except* `String`s. I'm asking for this restriction because it's for my own good."

When we do this, trying to insert a `Car` (or anything else) will bomb at compile time, as it should. And now, we don't have to typecast anything we get out of it: since Java knows it would only have put `String`s in, it knows that it will only get `String`s out. Therefore, "`stuff.get(0).length()`" works just fine, no cast necessary.

This is a good feature, and you should pretty much always use it. My only complaint is that I think the word "generic" is exactly the wrong word for it. They almost couldn't have chosen a worse term. Saying you have "a generic `ArrayList`" sounds like it ought to mean the *first* style (above), where the type wasn't specifically mentioned and therefore any object was fair game to put in it. But in actual fact, "a generic `ArrayList`" means "a specific, decidedly *non*-generic `ArrayList` that is declared to only hold values of a certain type." Go figure.

8.4 "Wrapper" classes

This is a good time to explain the usage of Java's **wrapper** classes. Recall that there are two kinds of Java variables: those that store primitive types (like `int`) and those that refer to objects (like `Car`). The biggest practical difference between these two, as we've seen, is where they're stored: primitive types go on the stack (and hence are pass-by-value) whereas the objects pointed to by reference variables go on the heap (and hence are pass-by-reference).

Another difference relates to the container classes like `ArrayList` that we've just been discussing. All those `java.util` goodies, it turns out, can store any type of object...but it must be exactly that: an *object.* In particular, they can't store primitive types.

So this means we can't do this:

```
import java.util.ArrayList;
...
ArrayList<int> uniformNumbers = new ArrayList<int>();   // NOPE
```

because there's no such thing as "an `ArrayList` of `int`s." That sucks because `int`s, `double`s, and `boolean`s are things we'd like to make `ArrayList`s of all the time.

Fortunately, there's an easy way around this. Each primitive type has a "wrapper class" in Java: it is the mold for objects so simple that they don't do anything except hold a piece of data: an `int` (or `char`, or `double`, *etc.*) We could do this, for instance:

```
    Integer michaelJordan = new Integer(23);
    Integer derekJeter = new Integer(2);
    Boolean gameOfThronesRocks = new Boolean(true);
```

and produce the memory diagram in Figure 8.5. Each object is nothing more than a shell that "wraps" a primitive piece of data.

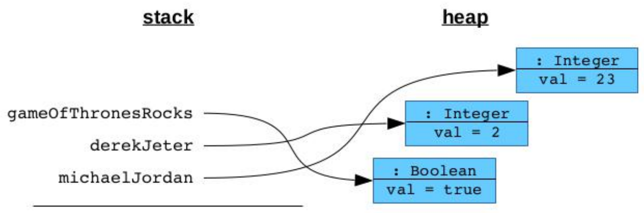


Figure 8.5: Wrapper objects live on the heap.

So far this isn't very exciting. But one reason we need it is so we can store primitive types in container objects like `ArrayListss`. Here's all we need to do to create our list of uniform numbers:

```
  import java.util.ArrayList;
  ...
  ArrayList<Integer> uniformNumbers = new ArrayList<Integer>();
```

and it works, since the elements of the `ArrayList` are declared to be objects, as required. (Really teensy-tiny objects that don't really do anything, but objects nonetheless.)

It's nice that after this, Java lets us work with primitive variables rather than the wrapper classes:

```
    uniformNumbers.add(23);
    uniformNumbers.add(2);
    uniformNumbers.add(20);
    int mikeSchmidt = uniformNumbers.get(2);
```

so we really only notice the wrapper part at instantiation time. This isn't the only time we need to use wrappers, but it's the only one we need in the immediate future.

8.5 The `Hashtable` data structure

One very, very common container type that we'll use is `java.util.Hashtable`. (If you've used a **dictionary** in Python, this is essentially the same thing.) A `Hashtable` holds a container of **key-value pairs**. Each key-value pair represents a *named* piece of data – the key is the name, and the value is the data. So unlike an `ArrayList`, where the data elements are numbered, and thus accessed by a numerical index, a `Hashtable` uses the keys to specify which piece of data is required.

An example will make this all clear. Suppose we want to keep track of superheroes and the names of their secret identities, so that if the government decides to legislate against super powers, we can hunt down all the potential perpetrators. We'll do so in a `Hashtable` called `alterEgos`:

```
import java.util.Hashtable;
...
Hashtable<String,String> alterEgos = new Hashtable<String,String>();
alterEgos.put("Superman","Clark Kent");
alterEgos.put("Batman","Bruce Wayne");
alterEgos.put("Elastigirl","Helen Incredible");
```

This instantiation syntax may make you bug-eyed. You'll see that we include not one but *two* types inside the "< ... >" markers; in this case, both are `String`s. These two specify *the type of the keys, and the type of the values.* Since our keys are text (superhero names) and our values are also text (the names of the alter egos) it makes sense to make this a `String`-to-`String` hash table.

The `.put()` method is used to add a new key-value pair to the table. It is also used to *change* the value that goes with a particular key. That works because in a hash table, *every key goes with just one value.* If we ran a line of code like this:

```
alterEgos.put("Batman","Rich Dude");
```

then the `String` "`Bruce Wayne`" would be permanently removed from memory, and replaced by "`Rich Dude`".

To retrieve the value that goes with a particular key, we use `.get()`:

```
String elastigirlTrueIdentity = alterEgos.get("Elastigirl");
System.out.println("Pssst...Superman is really: " +
    alterEgos.get("Superman"));
```

This looks much like the way we obtain items from an `ArrayList`, except that with `ArrayList.get()`, we passed an integer, and for `Hashtable.get()`, we pass a key (whatever type that may be).

A very common use of `Hashtable`s is to store objects based on some kind of name or identifier. For example:

```
import java.util.Hashtable;
...
Hashtable<Integer,Customer> customers = new Hashtable<Integer,Customer>();
...
customers.put(7533, new Customer("Joe Blow", "New Plymouth, ID"));
customers.put(6717, new Customer("Jill Hill", "New York, NY"));
...
Customer custNum6717 = customers.get(6717);
```

Here we're storing `Customer` objects by their customer ID numbers, for easy retrieval by that later.

In this book, I'll draw `Hashtable` objects as shown in Figure 8.6. Its `contents` table is full of references: each box in the left column points to a key (in this case, an `Integer`) and the box to its immediate right points to the corresponding value (a `Customer`). Notice that there is no well-defined order to the entries in the table – in this case, even though `Joe Blow` was the first one inserted, he occupies the second row of the table. This is to emphasize that when you add a key-value pair to a `Hashtable`, it doesn't remember anything about when you added it, only that you *did* add it.

Iterating through `Hashtables`

One last thing about `Hashtable`s: you can iterate through them as you can iterate through any other collection (like an `ArrayList`). But it's kind of tricky. Remember that there isn't any inherent

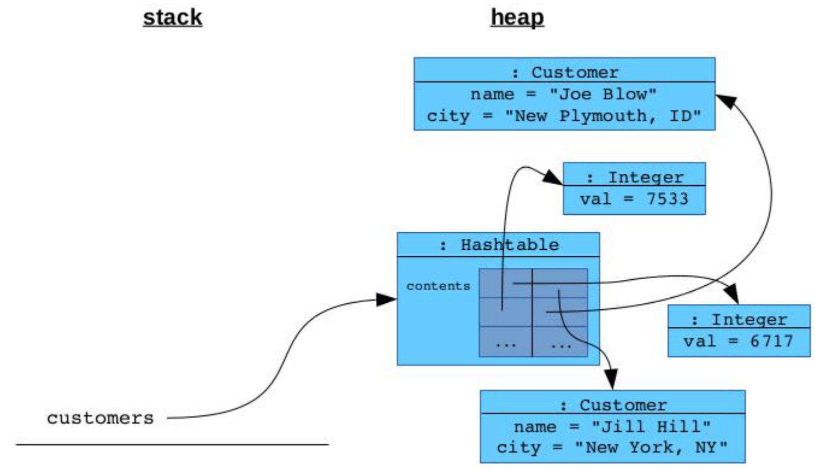


Figure 8.6: A `Hashtable` of `Customer` objects, stored by their customer IDs.

"order" to the key-value pairs, so you can't say "give me the first one, then the second, then ..., all the way to the end."

The way you achieve this is by using an `Enumeration` object, also from the `java.util` package. This is actually an example of the **Iterator** design pattern, which we'll see in a later chapter. For now, mostly just memorize the approach.

You call `.keys()` on the `Hashtable` which returns an `Enumeration` of the "key halves" of the key/value pairs. Think of an `Enumeration` as just "a way to iterate through a group of things, one by one." In this case, the "things" are the `Hashtable`'s keys. Every time you call `.nextElement()` on the `Enumeration` object, it gives you the next key and advances the "cursor" to point further on down the line. When you call `.hasMoreElements()` on the `Enumeration`, it tells you whether or not you can continue further. These two methods are just what you need to write a `while` loop to cycle through all the keys; and each time you get a key, you can use the original `Hashtable` to retrieve the corresponding value. To wit:

```
import java.util.Hashtable;
import java.util.Enumeration;
...
Hashtable<String,String> alterEgos = new Hashtable<String,String>();
...
Enumeration<String> superheroNames = alterEgos.keys();
while (superheroNames.hasMoreElements()) {
    String superheroName = superheroNames.nextElement();
    System.out.println(superheroName + " is really " +
        alterEgos.get(superheroName));
}
```

So the *key* is retrieved directly from the `.nextElement()` method, but to get the *value* we have to go back to the `Hashtable` itself and call `.get()` with the key.

Also, remember that the order in which you're given the key-value pairs here is *not* necessarily the order in which you inserted them. That order is irretrievable after the fact – if you need to keep track of it, you'll need a separate data structure (perhaps an `ArrayList` of the keys, in order of insertion) to remember it.

8.6 Command-line arguments

Every Java program you've ever written has this line in it:

```
    public static void main(String args[]) {
```

Have you ever wondered what that "`args`" thing is for?

Like all programming languages, Java lets you access the **command-line arguments** that the user typed after the program name when she ran the program. Command-line arguments are nothing new for a budding Linux user. For instance, this sort of command is second-nature to you by now:

```
$ cp  Program1.java  ~/backup
```

It consists of a command name ("`cp`") plus two command-line arguments, which are: "`Program1.java`", and "`~/backup`."

Now it turns out that *every* Linux program is capable of taking command-line arguments, including Java programs. Suppose I type this:

```
$ java  Simulator  UMW  Marymount
```

As you know, this is a command to run a Java program whose `main()` method is in `Simulator.java`. Apparently, the user is trying to simulate a basketball game between two opponents, and is using command-line arguments to specify which opponents.

The only question that remains is: how does the Java code get access to those strings that were typed on the command line when it was run? The answer is `args`. When `main()` is called, *Java passes the command-line arguments to it as an array of Strings.* It's that simple. Inside `main()`, you can treat the variable "`args`" in the same way as any other array. For instance, if your code says:

```
class Simulator {
    public static void main(String args[]) {
        System.out.println("There were " + args.length +
            " command-line args.");
    }
}
```

then the output, when run with the command above, will be:

```
There were 2 command-line args.
```

And if your code says:

```
class Simulator {
    public static void main(String args[]) {
        System.out.println("First arg was: " + args[0] + ".");
        System.out.println("Second arg was: " + args[1] + ".");
    }
}
```

then the output, when run with the command above, will be:

```
    First arg was: UMW.
    Second arg was: Marymount.
```

This is powerful because it allows users to run your program with different inputs and options *without* having to recompile it (or even have access to a compiler at all).

8.7 Sameness vs. identicality

Our last odd/end has to do with how Java determines whether two objects are "equal" to each other. It turns out that there are two different definitions of equality, which must be kept firmly separate in your mind: **sameness** and **identicality**.

Consider the memory diagram in Figure 8.8. Here we have four named variables: `p1`, `p2`, `p3`, and `p4`. Now riddle me this, Batman: at the moment this snapshot was taken, which of these variables do you consider to be *equal* to each other?

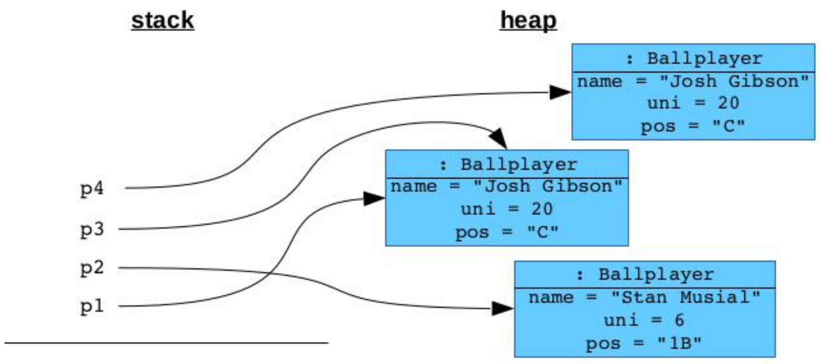


Figure 8.7: What does "equal" mean?

I think we can agree that `p2` does not "equal" any of the others. That leaves the other three. Are `p1` and `p3` "equal?" Are `p3` and `p4` "equal?"

It all depends on what your definition of "equal" is, of course. And here's how Java does it: it says that `p1` and `p3` are **the same**, while

```
    if (p1 == p3) {
        System.out.println("This WILL print.");
    }
    if (p3 == p4) {
        System.out.println("This will NOT print.");
    }
    if (p1.equals(p3)) {
        System.out.println("This WILL print.");
    }
    if (p3.equals(p4)) {
        System.out.println("This WILL probably print (but see below).");
    }
```

Figure 8.8: Testing for sameness vs. identicality.

`p3` and `p4` are **identical**. "The same" means that the two reference variables refer to *the very same* object in memory. They are literally pointing to the same memory address, and so they are pointing to the same "copy" of the object. This is easy to see by imagining changes to that object – if the left-most Josh Gibson changed uniform numbers when he was traded from the Homestead Grays to the Pittsburgh Crawfords, both `p1` and `p3` would automatically see that change. They point to the same object, so they refer to the same uniform number variable: if it changes, they both change.

But `p3` and `p4` are not the same – they're merely identical. They refer to two *different* objects which just happen to have the same internal state. Changing one would not affect the other. At the moment, they're duplicate copies, but copies nevertheless.

(By the way, realize that sameness *implies* identicality. The objects `p1` and `p3` refer to the same, and therefore they are *also* identical.)

Testing for sameness and identicality

Sometimes a programmer cares about one of these conditions, and sometimes the other. Java has two syntaxes to distinguish between the two tests.

To test for sameness, use "==". To test for identicality, use "`.equals()`".

Figure 8.8 gives an example. In the first two lines of that code, we are testing for sameness. So even though the objects referred to by `p3` and `p4` are spittin' images of each other, identical in every conceivable way, they are nevertheless *not* "==" to each other. But they might well be `.equals()` to each other...if we take the special step described next.

Overriding `.equals()`

It turns out that the last print statement of the preceding code statement will actually *not* get printed *unless we inform Java about what "identical" actually means for this class.* By default, Java will fall back to just using the sameness test when `.equals()` is used. The way to tell it how to test for identicality is to *override* the `.equals()` method for the `Ballplayer` class. Here's how:

```
class Ballplayer {
    private String name;
    private int uni;
    private String pos;

    ...
    public boolean equals(Ballplayer b) {
        if (this.name.equals(b.name) &&
            this.uni == b.uni &&
            this.pos.equals(b.pos)) {
            return true;
        } else {
            return false;
        }
    }
    ...
}
```

We've told Java that for `Ballplayers`, "identical" means objects that have the same `name`, `uni`form number, and `pos`ition. We could have done anything else we wanted; for instance:

```
class Ballplayer {
    private String name;
    private int uni;
    private String pos;

    ...
    public boolean equals(Ballplayer b) {
        if (this.name.charAt(0) == b.name.charAt(0)) {
            return true;
        } else {
            return false;
        }
    }
    ...
}
```

Now any two `Ballplayer`s whose name started with the same letter would be considered "identical" – `Josh Gibson` would be considered identical to `Joe Dimaggio`. This is a weird idea, and not normally encouraged. But sometimes a situation does come up where it makes sense to say that one object `.equals()` another even if only *some* of their information matches.

Beware **comparing `Strings`! (Use `.equals()`)**

Notice how in the very definition of `Ballplayer.equals()` we called the `.equals()` method for `String`s. You may have expected it to say:

```
    public boolean equals(Ballplayer b) {
        // WRONG
        if (this.name == b.name &&
            this.uni == b.uni &&
            this.pos == b.pos) {
            return true;
        } else {
            return false;
        }
    }
```

instead. After all, that's less typing, right?

But my choice here was deliberate, and when comparing `String`s in particular *you must always use* ***`.equals()`*** like this. The reason is esoteric, and has to do with how Java tries to conserve memory by re-using space for different `String` objects with identical contents. The upshot of this is that "==" *sometimes* works the way you expect, and other times does not. One minute you'll find that `"Satchell" == "Satchell"` and then a moment later you'll discover that `"Buck" != "Buck"`. It'll seem random, and it basically is. But if you always use `.equals()` to compare `String`s, it'll always return true when their contents are exactly the same, and you won't have any painful late-night debugging sessions (for that reason, at least).

Chapter 9

UML sequence diagrams

Class diagrams are the bread-and-butter of UML. They depict the static features of software systems: the classes, methods, and associations that connect them. Complementary to class diagrams are another type of UML artifact[1] called **sequence diagrams**. They show the *dynamic* interrelations between objects as a system's code executes.

Each sequence diagram depicts one scenario, or flow through the system. Unlike a class diagram, which is sort of "always true" and shows all the permanent and unvarying features of the program in question, every sequence diagram shows its own path: its own thread of execution in a particular, hypothetical scenario. After all, nearly every time you run a program, something different happens, either because the user makes different choices, network and system latencies cause various tasks to end at different times, or a random number generator is involved. A sequence diagram selects just *one* possible outcome and highlights it start-to-finish so that an example of how the classes are intended to interact is unveiled.

I think of a sequence diagram as a "sounding" in the nautical sense.

[1]By the way, the term **artifact** in software engineering means "a document, diagram, computer program, or some other tangible deliverable that results from carrying out a development activity." The measurable progress a development team makes consists of the various artifacts they produce along the way.

In ancient times, ship captains who suspected they were approaching land would test how deep the water was by probing it with a sounding line. Modern ships do the equivalent with sonar. A sounding is an exploratory investigation down one possible path through the water's depths to see what's below. One sounding doesn't tell you everything about the whole region's topography, but it tells you a great deal about the specific area you're in. And if you perform several soundings, you can combine the clues you obtain from each one to build a mental picture of a wider section of the ocean floor. A programmer can do the same by learning from several different sequence diagrams, each of which tells a different story.

Sequence diagrams are designed to be perused in conjunction with their corresponding class diagrams. I always tell students: "when you look at a sequence diagram, only look at it with one eyeball; keep the other eyeball on the class diagram." As we'll see, both diagrams have to be "in sync" with each other, since information presented on one must be compatible with what's on the other.

9.1 Going backwards

Reverse-engineering a sequence diagram from code

Flip back to Figure 6.14 from chapter 6 (p. 108). This is our baseball simulator example. Keep your finger in this page for reference as you follow along in this section.

We're going to begin our "sounding" through this program by starting in the `Simulator` class's `.printAllStars()` method. Its job is to print out the names of the teams and their all-time greatest hitters, like this:

```
NY Yankees - Babe Ruth (3)
St. Louis Cardinals - Stan Musial (6)
NY Giants - Willie Mays (24)
Milwaukee Braves - Hank Aaron (44)
Boston Red Sox - Ted Williams (9)
...
```

The story told by a sequence diagram has to begin somewhere; this is sometimes, but not always, in `main()`. Here we pick up the action in `.printAllStars()`. Let's say the code for it looked like this:

```
class Simulator {
    ...
    public void printAllStars() {
        int numTeams = teams.size();
        for (int i=0; i<numTeams; i++) {
            Team nextTeam = teams.get(i);
            System.out.println(nextTeam + " - " +
                nextTeam.getBestHitter());
        }
    }
}
```

Normally, we *start* with a UML design and then write the code to implement it; but here, we're going to reverse-engineer the UML sequence diagram *from* the code. This is because so far you've seen a lot more code than sequence diagrams in your life, and I think you'll better understand how sequence diagrams work if I show it to you this way first. (In the next section we'll go the other way.)

Sequence diagram features

Now look at the enormous diagram in Figure 9.1. To get your bearings, note these two important aspects of sequence diagrams:

- The ***objects*** (and occasionally classes) that participate in this scenario ***appear across the top*** of the diagram. There is no inherent meaning to the order in which they appear, but often objects that are involved earlier in the code path are on the left side.
- The dashed line that extends down the page from each box "goes with" that object. Horizontal arrows that originate from (or point to) that line are methods that object calls (or that are called on it).
- ***Time goes down.*** In other words, as the code path executes in time, we move progressively further down the page.

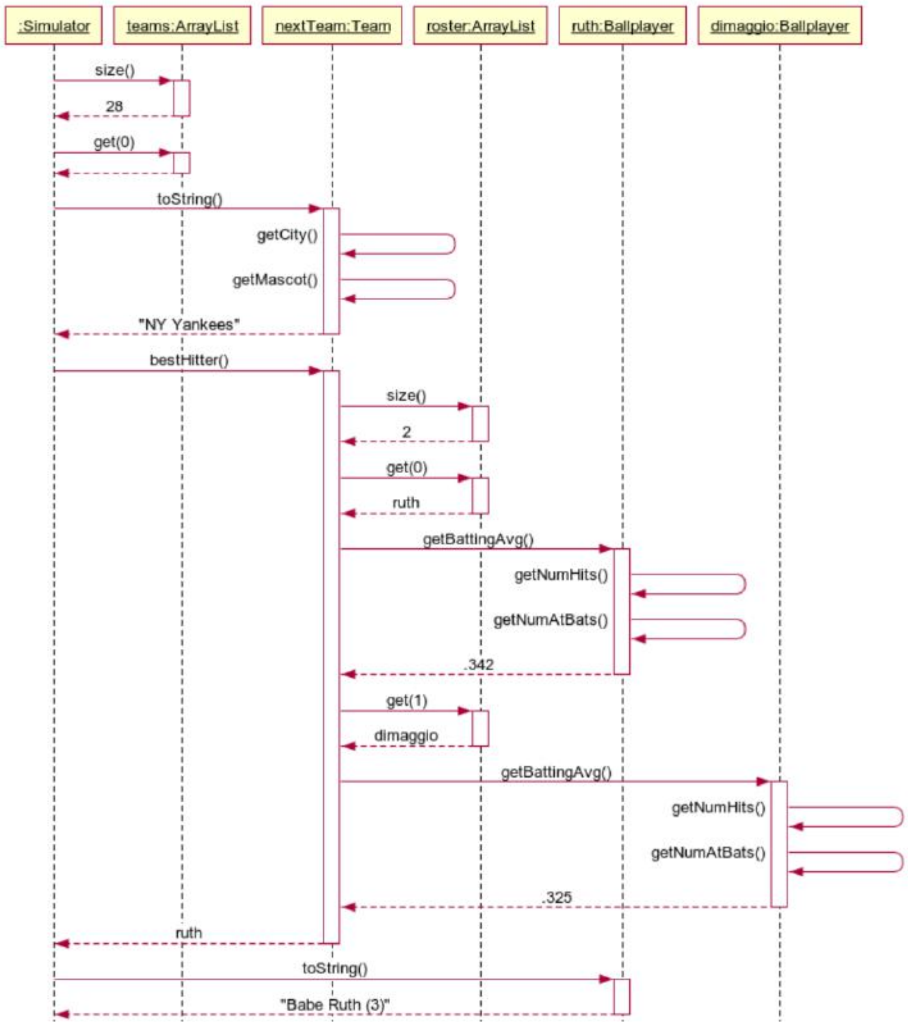


Figure 9.1: A sequence diagram: Baseball example.

What the arrows mean

So we begin by looking at the upper-left corner, where the top-most arrow goes from the "`:Simulator`" box to the "`teams:ArrayList`" box, and is labeled "`size()`". As I'm sure you can guess, this corresponds to the line of code: "`int numTeams = teams.size();`" which is the first thing we do in `.printAllStars()`.

☞ Here's how you interpret *any* arrow on a sequence diagram:

1. Every horizontal arrow is a method call. (Important: method calls are the *only* part of the code that can be shown on a sequence diagram.)
2. The vertical dashed line where the arrow *starts* is the object that makes that call. (In other words, if an arrow starts from a ":Person" box, then somewhere in a method of the `Person` class we can expect to find this method being called.)
3. The vertical dashed line where the arrow *ends* is the object on which the method is *being* called. (In other words, if an arrow ends on a ":Book" box, then we're calling the method on that `Book` object.)
4. The writing on the arrow is the name of the method being called, plus any parameters.

So, that first arrow in Figure 9.1 means:

☞ "Somewhere in a method of the `Simulator` class, we're calling `.size()` on an `ArrayList` object named `teams`."

(Before you go on, read that sentence, and stare at that arrow, several times and make sure you're absolutely certain of every detail. This one key idea is the secret to understanding sequence diagrams.)

Notice that the arrow terminates on the top of a skinny box that extends a centimeter or so down the `teams:ArrayList`'s vertical dashed line. This skinny box represents *the time during which the program is "in" the `.size()` method.* It's not actually intended to specify a duration of time (as in "the `.size()` method will take 1.4 milliseconds to complete") but rather the-fact-that-it's-being-executed at all. In a moment, we'll see that these skinny boxes can be (much) taller if we want our sequence diagram to show detail about what happens *inside* that method.

Following the flow

Okay. Let's continue down the diagram and see the rest of the action unfolding. Keep your finger on Figure 9.1 as we go so you don't lose your place.

1. The dashed vertical line pointing left with a "28" written on it is the *return value.* In this particular scenario, apparently there are 28 teams (*i.e.*, 28 entries in the `ArrayList`).
2. The next call is to `.get()` the first entry in the list. Observe how the line says literally "`.get(0)`" whereas the code has "`.get(i)`", with `i` being a loop variable. This is perfectly fine. It means that as the code executes, the call to `ArrayList.get()` is effectively passed the value `0` as an argument the first time it's called, which is of course true. *The fact that a loop is required is implicit.* The designer, who created this sequence diagram, isn't spelling out details like "use a loop here" for the programmer. Instead, she's illustrating the intended pattern of method calls between objects – the programmer will infer the need for loops, local variables, *etc.*
3. The dashed "return value" line is blank. That's okay too: it means the designer didn't bother to specify a name for it. (Get used to design diagrams containing differing levels of information in different places.)
4. Then `.toString()` is called on the returned `Team` object. Why? Because we're `System.out.println()`'ing it, and as you'll recall from p. 52, the `.toString()` method is automatically invoked for any object that tries to be "printed." So even though our code doesn't explicitly say "`.toString()`" in it, the method is nevertheless called, and is thus dutifully shown on the sequence diagram.
5. Now pay close attention. Notice that the `.toString()` arrow isn't followed by a dashed return value arrow right away.

Instead, the skinny box extends down the page an inch or more. This is because *we're showing what's happening inside .toString()*. In this case, there are two bendy arrows going from the `nextTeam:Team` line *back to itself*. These mean that the methods `.getCity()` and `.getMascot()` are being called by the `Team` object *on itself*. This may disorient you at first, but of course there's nothing really strange about an object calling a method on itself. After all, our `Team.toString()` method looks like this:

```
public class Team {
    ...
    public String toString() {
        return this.getCity() + " " + this.getMascot();
    }
    ...
    private String getCity() { return this.city; }
    private String getMascot() { return this.mascot; }
}
```

Calling a method on "`this`" is exactly what's indicated by those bendy arrows.

6. Now, finally, we get our dashed arrow back to "`:Simulator`", with a return value of "`NY Yankees`". The control flow thus transfers from `.toString()` back to `.printAllStars()`.

7. Next up is the `.bestHitter()` method, called later on in that same `.println()` call. This commences an even longer skinny box, because `.bestHitter()` has a lot to do. Here it is:

```
    ...
    private Ballplayer bestHitter() {
        int numPlayers = roster.size();
        Ballplayer best = roster.get(0);
        for (int i=0; i<numPlayers; i++) {
            Ballplayer b = roster.get(i);
            if (b.getBattingAvg() > best.getBattingAvg()) {
                best = b;
            }
        }
        return best;
    }
}
```

You can see that after getting the size of the roster (in this case, only two players since this is a long enough example as it is!) we get each `Ballplayer` in turn, ask for his batting average, and compare it to our "best so far" in a typical find-the-max-element type of loop.

All of this is faithfully represented in the sequence diagram. First, the `Team` object calls `.size()` (and gets "2" back). Then it calls `.get(0)` (and gets back a player; let's say Babe Ruth), and then calls `.getBattingAvg()` on it (getting the number .342, a jaw-dropping lifetime average, especially for a power hitter). A moment later, it does the same with the second player (say, Joe Dimaggio) and gets his average (still amazing, but "only" .325) for comparison.

8. More detail is shown inside the `Ballplayer.getBattingAvg()` calls. That code looks like this:

```
public class Ballplayer {
    ...
    public double getBattingAvg() {
        return ((double) this.getNumHits()) / this.getNumAtBats();
    }
    private String getNumHits() { return this.numHits; }
    private String getNumAtBats() { return this.numABs; }
    ...
}
```

   and makes two "self-calls" to get this `Ballplayer`'s two relevant statistics. Those calls are again shown as bendy loops.

9. Finally, the `Team` returns its best hitter (Babe Ruth in this scenario) back to the `Simulator`, which again calls `.toString()` implicitly, this time on the `Ballplayer` object. That method returns the name of the ballplayer and his uniform number, formatted nicely, for `Simulator.printAllStars()` to print.

Coming up for air

That was a long journey through the weeds, because that sequence diagram contained a boatload of stuff. In fact, one of the big takeaways here is that *a sequence diagram contains a ton of information about how to write the code.*

A sequence diagram omits programming-specific details like whether to create local variables and what to call them; whether you need a loop and what type of loop you might choose; what the exact formula is for a computation, or the logic to test for a condition; *etc.* But it does present you with a silver platter that says which objects of which types are intended to call which methods on which other objects, and in what sequence.

In practice, I'd estimate that this works out to be about 70% or so of the decisions the programmer would otherwise have to make. What a windfall!

Before we move to our second example, let me tell you the two most common errors I see among students trying to interpret (or create) sequence diagrams:

1. **Misinterpreting what the arrowhead-side of the arrow means.** Each arrow points to a line which represents *the object on which the method is being called.* There's a great way to sanity check this: make sure that the class (for whatever type of object the arrow is pointing to) actually has a method of that name!

   Figure 9.2 shows some common mistakes. Neither of the top two sequence diagram fragments can possibly be correct, because they show an `.add()` method being called on a `Review` object. Now I ask you: look at the class diagram – do you see an `.add()` method on the `Review` class? Nope. That means that right away, without even thinking any further, you can rule out the top two sequence diagram attempts in that figure.

   The bottom two versions pass this test, because in those diagram `.add()` is being called on a ***`Movie`*** object, which does indeed have an `.add()` method.

2. **Misinterpreting what the non-arrowhead-side of the arrow means.** Each arrow originates from a line which represents *the object whose code is making the method call* (on a different object, normally).

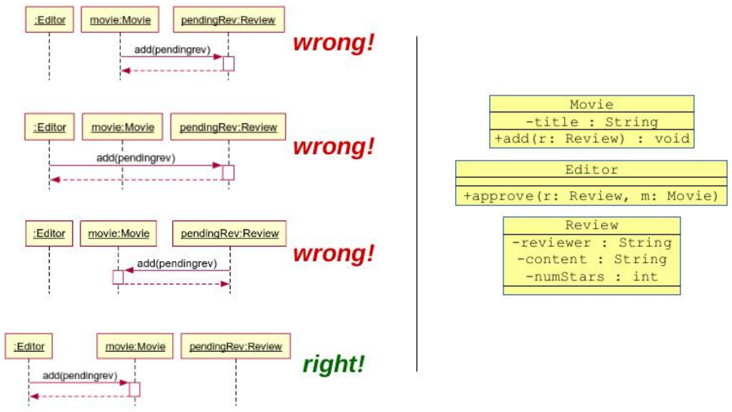


Figure 9.2: Many wrong ways to draw the sequence diagram arrow...and one right way.

To sanity check this, we can't look at the class diagram alone. We have to think about what the code looks like. Suppose this were the case:

```
class Editor {
    ...
    public void approve(Review r, Movie m) {
        ...
        m.add(r);
    }
}
```

Even before we saw this code snippet, we already knew that `.add()` would be called on a `Movie` object (and passed a `Review` object as an argument) because that's in line with the class diagram. But what we learn now is *where* that line of code exists. Is it written in a method of the `Review` class? Or the `Movie` class? Nope – it's in the `.approve()` method *of the* ***Editor*** *class.* That tells us that the *bottom* of the four sequence diagrams in Figure 9.2 is the correct one. The third one shows a ***Review*** making the method call, but if that were the case, the "`m.add(r)`" line would be somewhere in `Review.java`.

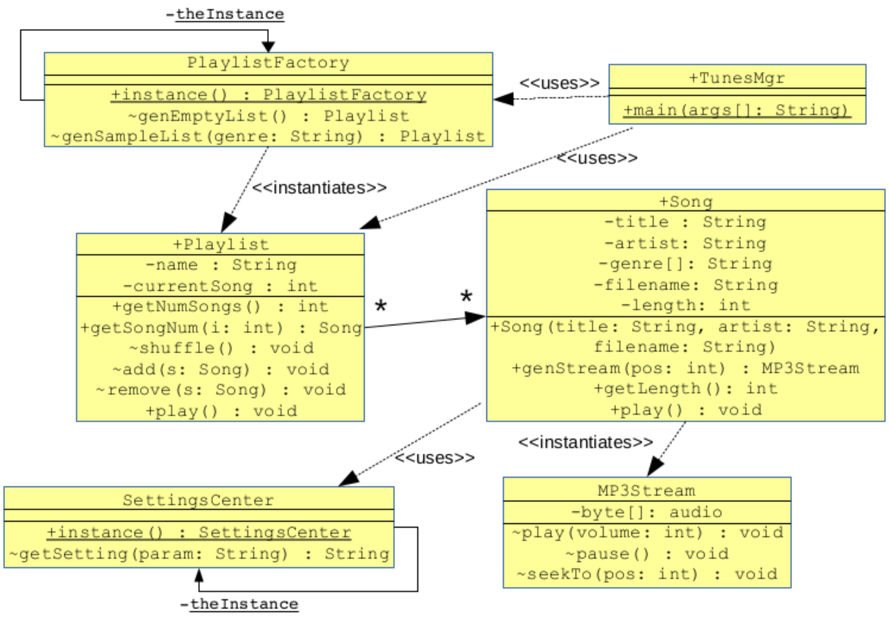


Figure 9.3: The playlist program's class diagram.

9.2 Going forwards

Using a sequence diagram to guide the implementation

Now normally we won't be drawing a sequence diagram after we've written the code, although that does actually happen sometimes, like when we want to illustrate the behavior of a system to a new member of our programming team. (It's usually easier for them to see the diagram at a glance than it is for them to wade through a bunch of code and try to make sense of it.)

We'll now *start* with a design (both class and sequence diagram), and see what we can infer about what the code to implement it should look like. Figures 9.3 and 9.4 give the design, which you are encouraged to study in detail.

Here are some things we can read right off the sequence diagram (starting at the top):

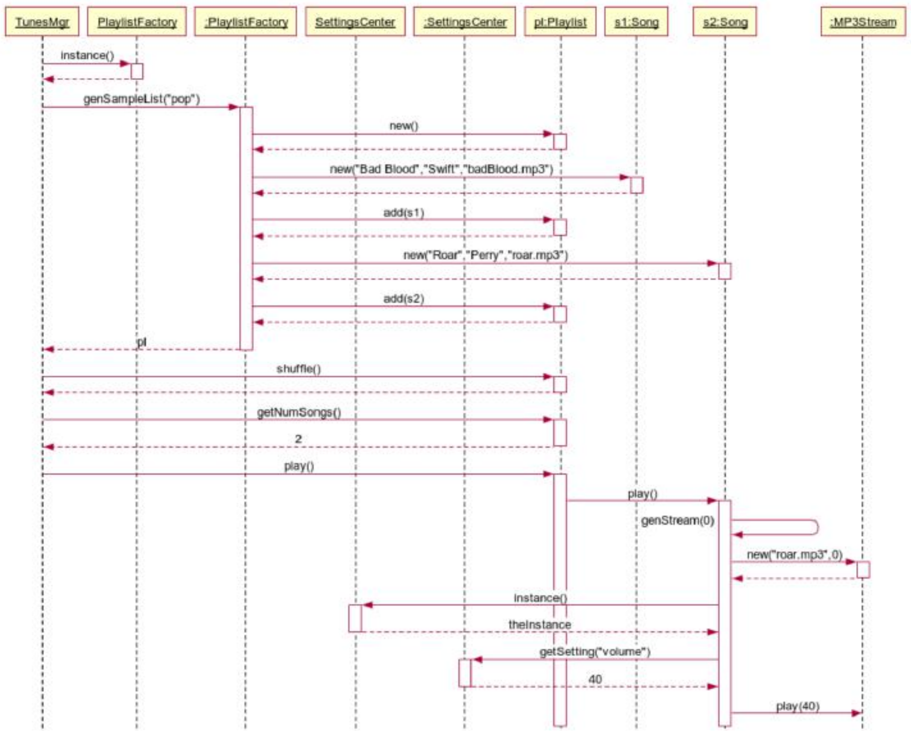


Figure 9.4: One of the playlist program's sequence diagrams.

1. Something in the `TunesMgr` class will call `.instance()` on the `PlaylistFactory` class. (Notice that the second box from the left is missing a colon, so it must represent the *class* `PlaylistFactory` rather than an object of that type.) Since we're calling `.instance()` on a class rather than an object, it had better be `static`; and when we look at the class diagram, happily it is.

   Furthermore, we can easily guess *which* `TunesMgr` method is making that method call, since there's only one listed for it: `main()`. After calling `.instance()` on the class, it looks like it calls `.genSampleList("pop")` on the object. Putting it all together, we can surmise that this code should appear in our

`TunesMgr.java` file:

```
class TunesMgr {
    public static void main(String args[]) {
        PlaylistFactory pf = PlaylistFactory.instance();
        Playlist p = pf.genSampleList("pop");
        ...to be continued...
    }
}
```

All we had to figure out for ourselves was to store the return values in variables, which seemed like a good idea.

2. Switching scenes to the `PlaylistFactory`, we get a good sense of the innards of its `.genSampleList()` method. The first couple of arrows coming out of `:PlaylistFactory` say "new", which as you may guess indicates an *instantiation* and a constructor invocation. In the first case, we're instantiating a new `Playlist` object. Apparently the constructor for `Playlist` doesn't take any arguments (or the sequence diagram author didn't bother specifying any). The `Song` object we instantiate next, on the other hand, takes a slew of arguments: a song title, artist, and filename.

   After creating this new `Song`, we `.add()` it to the `Playlist`. We then do the same for another `Song`. In sum, here's the kind of thing that `.getSampleList()` must contain:

```
class PlaylistFactory {
    ...
    Playlist genSampleList(String genre) {
        ...
        Playlist pl = new Playlist();
        Song s1 = new Song("Bad Blood", "Swift", "badBlood.mp3");
        pl.add(s1);
        Song s2 = new Song("Roar", "Perry", "roar.mp3");
        pl.add(s2);
                ...
        return pl;
    }
    ...
}
```

   Now lest we be too hasty, let's step back for a moment. The above code might well not *literally* be in `.getSampleList()`;

after all, it refers to specific, hardcoded songs. It also doesn't take into account the value of `genre`; presumably, if we passed "`rap`" or "`classical`" as the argument, we'd get a different song selection. So I'm not actually saying that you can read the code off the diagram without thinking. What the sequence diagram tells us, though, is good information about *what kinds of things will happen* in each method. One could imagine the real `.getSampleList()` reading song titles from a file or from an Internet source depending on the genre, for instance. Even in that case, though, all the sequence diagram essentials would be the same: we'd be instantiating a new `Playlist` and `Songs`, adding the `Songs` to the list, *etc.*

3. "Back to `main()`!" the sequence diagram announces. After returning the `Playlist` (which Figure 9.4 suggests we call "`pl`") our `main()` method commences calling three methods on it. We can thus further flesh out our main method as follows:

```
class TunesMgr {
    public static void main(String args[]) {
        PlaylistFactory pf = PlaylistFactory.instance();
        Playlist p = pf.genSampleList("pop");

        p.shuffle();
        int num = p.getNumSongs();
        p.play();
    }
}
```

   Nothing explicitly told us to save the `.getNumSongs()` return value in a variable, but we did it anyway. We're also not told what specifically we would do with that information – perhaps display it for the user, or estimate the duration of the playlist based on it. At any rate, for now we'll just save it and move on.

4. Finally, calling `.play()` on the `Playlist` turns around and calls `.play()` on each of its `Songs`, not surprisingly. We can thus deduce something like:

```
public class Song {
    ...
    private ArrayList<Song> songs;
    ...
    public void play() {
        for (Song s: songs) {
            s.play();
        }
    }
    ...
}
```

5. As for ***Song's*** `.play()` method[2], it apparently involves calling its own `.genStream()` method, which in turn instantiates an `MP3Stream` object to do the actual playing. After this, it gets the Singleton `.instance()` from the `SettingsCenter` (note that arrows can go right-to-left on sequence diagrams in addition to left-to-right) and gets the user's `"volume"` setting. Finally, it calls `.play()` on its `MP3Stream` to play at the correct volume. Putting it all together, we infer this sort of code:

```
public class Song {
    ...
    public void play() {
        // ("0" means "start at the beginning of the song")
        MP3Stream s = this.genStream(0);
        int vol = SettingsCenter.instance().getSetting("volume");
        s.play(vol);
    }
    ...
    public MP3Stream genStream(int pos) {
        ...
        return new MP3Stream(this.filename, pos);
    }
}
```

Here we choose to chain together the calls to `.instance()` and `.getSetting()` rather than saving the `SettingsCenter` instance to a variable. This is programmer's discretion.

[2]Interestingly, note that there are *three* different methods called `.play()` in this design, on each of three different classes: `Playlist`, `Play`, and `MP3Stream`.

Many challenges and questions still remain: how exactly do we go about generating sample playlists based on genre? What algorithm will `.shuffle()` use? Do we play all the songs in a row, back to back, or do we insert a few seconds of silence in between? How exactly does the `MP3Stream` read the bytes off the disk and send them to the audio speakers? *Etc.* These are important decisions, every bit as important as what kind of wood and nails to use for each wall we frame.

But the class diagrams and sequence diagrams have given us a tour of the whole house as it was envisioned by the architect. We now have a framework into which all the little decisions can be fit. And that's the first step towards an elegant and maintainable program.

Chapter 10

Persistence and hydration

Often, we want some of our objects to maintain their existence between executions of the program. They're intended to be durable and lasting, and so we need a way to record their details in some kind of permanent record so they can be resurrected later.

Examples abound. Consider a social media app, where new users can register, login, post messages, friend each other, *etc.* We'd likely store information about all this in various `User`, `Post`, `Profile`, and `FriendRequest` objects. But it sure would be a bummer if all that data disappeared any time the server needed to be restarted!

Or consider a drawing application, which we can use to create figures like lines, rectangles, circles, and so forth, to create the kinds of diagrams contained in this book. Our `Drawing`, `Line`, and `Rectangle` objects, whose instances held information about position, size, and color, would be next to useless if they weren't able to store themselves permanently somehow, to be reloaded in a later execution of the app. It would be like a word processor without a "save" function.

What we want is a way to **persist** (or save) an object, and **hydrate** (or restore) it later. Persistence is taking an ephemeral, in-memory object and writing it out to some form of permanent storage – a file in the filesystem, a database, a network drive, or anything else that will stay put even when the program ends. Hydration is the reverse process: resurrecting that stored version of the entity's state into a living, breathing object once again.

10.1 Java object serialization

There are several ways to implement such operations. One, which I don't recommend, is built into Java and is called "serialization." The `java.io` package has two classes – `ObjectOutputStream` and `ObjectInputStream` – for this purpose. The idea is that just as you can write primitive types like integers and strings to a stream with familiar file I/O operations, you can also read and write bona fide objects themselves.[1] Just open an `ObjectOutputStream` to a file and call `.writeObject()`, and the object you pass, in its entirety, will be saved in the file. `ObjectInputStream.readObject()` performs the reverse process.

This all sounds like a great idea, but there's some serious practical difficulties to consider. For one, `ObjectOutputStream` stores objects in a **binary format** rather than a **text format**. This means that the files it produces can't be easily analyzed – they're compact, but opaque, sequences of 1's and 0's. If you try to open such a file in a text editor like `vim` to inspect what objects and values it contains, you'll get gobbledy-gook. The upshot is that it's nearly impossible to debug your program or even figure out what was saved.

Just as problematic is the fact that serialization is not **forwards-compatible**. When you write an object to a stream in this way, the data that is persisted is hard-wired to the particular version of the class the object is a member of. This isn't a problem if your code is completely, permanently stable. But during a development cycle, you're constantly changing the nature of many classes, including their instance variables, names, and types. As soon as you change a class in any significant way, *bam!!* all previously stored copies of objects are now unreadable. When you try to hydrate them with `ObjectInputStream`, it'll break saying, "whoa, the class of this stored object was an old version of the class – it doesn't match the

[1]All objects that are saved and restored in this way must be from classes that are declared to "implement" the `java.io.Serializable` "interface." We'll talk about implementing interfaces in Section 12.4.

current `.java` file." And that data is, for all intents and purposes, lost.

I've learned the hard way that when you persist objects, you want to store them in a format which is transparent and forwards-compatible. You need to be able to inspect exactly what was stored, and have that data be readable even if you change the class's definition later on.

10.2 Using a database

An excellent solution for persistence and hydration is to use a Database Management System (DBMS) like MySQL, PostgreSQL, or MongoDB. These products specialize in this very task. Some, like those with "SQL" in the name, create **relational databases** which hold rectangular tables of data records, sort of like gigantic spreadsheets. Others are called **non-relational** or **NoSQL databases** and can store data in a more flexible, non-rectangular format. Programs in Java (or any other language) can create **connections** to either kind of database, and read/write to them with standard commands.

The reason I won't go further into this option is because it's really a separate topic from OOA&D, and merits a whole course in its own right (at UMW, it's CPSC 350). Once you learn that material, it's really the best way to do the whole persistence/hydration thing.

10.3 Using plain text files

In this book, I'll just explain how to do caveman-style persistence and hydration: using plain text files. This will allow you to leverage your basic file I/O knowledge from previous programming courses to achieve our purpose of making objects persistent.

The first thing we need to decide is how to organize our storage. Should each object have its own file? Should all objects of the same class be stored in one file? Or should *everything* we want to persist be stored in one file?

There are arguments for all of these, and it really comes down to the needs of the application. If we have a file-based save/restore paradigm, like our drawing editor example from earlier, it probably makes sense to store everything in a single file. When the user chooses "save," we write their entire drawing to the filename of their choice, and when they choose "open," we read it back.

In a social network scenario, on the other hand, we probably want more granularity. As soon as a user posts a new message or changes their status, we want to create or re-write *one* file (not our entire memory footprint) with the updated information. This way we can write in little snippets as we go, creating a bunch of files in a directory that collectively represent the entire state of the application. When the system is taken down and rebooted later on, it reads all those files to reconstruct its previous state.

Example: a resort reservation system

Here, let's assume that we want the first scenario: a single file that contains all relevant information to the application. We'll create part of a simple hotel reservation system, which stores and retrieves information about various resort destinations. It consists of just two classes: `Resort`, objects of which represent individual hotels; and `ReservationSystem`, the main program.

For simplicity, I'm only going to show the parts of the program that are related to persistence and hydration. (Clearly there are lots of other things the code would need to do, like display lists of matching hotels in response to searches, and actually make reservations.)

The key idea is this. Each class whose objects we want to save needs two things:

- For persistence, a `.persist()` method that writes the object out to the `PrintWriter`[2] passed.

[2] `PrintWriter` is a class in `java.io` that has `.println()` methods for various data types. I like it for that reason. Otherwise, it's essentially the same as any other kind of `Writer`, like a `FileWriter`.

- For hydration, a constructor that takes a `Scanner`[3] and creates a new object based on the information queued up on that `Scanner`.

Let's do the `Resort` class first. The key question is: what do we want the persisted form of a `Resort` object to look like? Any way of writing all the necessary information in a way that we can unambiguously get it back will do. I vote for this:

```
Westword Spa and Surf
312-555-1234
5 stars
$$$
This luxurious beach property
is a pleasure for all who visit.
Dogs and cats welcome too!
.
```

The textual representation of a `Resort` consists of the following parts:

1. A single line of text giving the resort's name.
2. A single line of text giving the resort's telephone number.
3. A single line of text with a digit from 1 to 5, followed by a space and the String "`stars`" (or "`star`").
4. A single line of text with dollar signs, the number of which indicates the expense of the resort (on a scale of \$ to \$\$\$\$).
5. One or more lines of text giving a description of the resort.
6. A line containing only a period.

As you'll remember from your previous programming courses, that last item (#6) is called a **delimiter** since it "delimits" (marks the end of) the resort entry.

Now, for the Java class. Figure 10.1 *almost* has what we want. Look at it carefully.

[3] A `Scanner` is a class in `java.util` that can parse primitive types from a stream of text.

```
class Resort {
  private String name, desc, phone;
  private int rating, cost;

  Resort(Scanner s) {
      name = s.nextLine();
      phone = s.nextLine();
      rating = s.nextInt();
      s.nextLine();    // read and discard the " stars" part.
      cost = s.nextLine().length();
      desc = "";
      String next = s.nextLine();
      while (!next.equals(".")) {
          desc = desc + next + "\n";
          next = s.nextLine();
      }
  }

  void persist(PrintWriter pw) {
      pw.println(name);
      pw.println(phone);
      pw.println(rating + " " + (rating == 1 ? "star" : "stars"));
      for (int i=0; i<cost; i++) {
          pw.print("$");
      }
      pw.println();  // newline for the $$$
      pw.println(desc);
      pw.println(".");
  }
}
```

Figure 10.1: Our first cut at the `Resort` class.

`Resort`: hydration

In Figure 10.1, the constructor is used for hydration. Perhaps the trickiest thing about it is that we *pass it a `Scanner` object.* Even if you've worked with `Scanner` before, this may be a new idea to you: passing one around between objects so that different objects can read different parts of a file. But that's precisely what we're doing here. Remember that with file I/O, you have a **cursor** open to a file, which is always "at" a particular location within the file. Reading from the `Scanner` (perhaps using one of the `.next*()` methods like

`.nextInt()` or `.nextLine()`) **advances** the cursor through the file to the next bit.

So when a client calls "`new Resort(someScanner)`", they're saying "this `Scanner`'s cursor is currently positioned right at the beginning of some resort information. So please hydrate one `Resort` object for me by reading just that much of the file."

The constructor proceeds to do just that. It reads the name, phone number, rating, cost, and description, in that order, ending when it reaches (and reads past) the period ("`.`") delimiter. I'll let you glance through the code and see if you agree with my implementation here. There are a few interesting nuances, like:

- When reading the rating ("`4 stars`") we call `.nextInt()` to read the number part, and then `.nextLine()` to read past and discard the rest of the line.
- When reading the cost ("`$$$$`") we take the length of the string (the number of dollar signs) and store that number.
- When reading the possibly-multi-line description, we continually keep a lookout for the delimiter. As long as the lines we read are *not* the period on a line by itself, we append them to the end of our `desc` inst var, along with the newline character ("`\n`") that `.nextLine()` discards.

All just fiddly stuff.

`Resort`: persistence

The inverse process of our constructor is our `.persist()` method, which works similarly: given a `PrintWriter` – to which other information has quite possibly already been written – write out the text representation of this `Resort` object.

Just a few things of note here:

- If you haven't seen the wacky syntax at the end of the "`rating`" line, it's worth knowing. It's called a "conditional expression" or a "question-mark-colon operator" and is a compact way of sticking a short `if`/`else` onto one line. It means "if the boolean expression before the question mark is true, replace this whole expression with the thing before the colon; otherwise, replace it with the thing after the colon." Less than legible to beginners, but a nice shortcut for experts.

- A simple `for` loop converts the integer `cost` into a string of dollar signs. We also have an explicit `.println()` after that loop so that we include the line feed after those dollar signs.
- We explicitly write the delimiter ("`.`") at the end of the description, so that when read back by the hydration process, the constructor knows where the description stops.

Clearly the persistence and hydration code for a class need to be kept in lock-step with each other. Adding a new field to the former, for instance, necessitates us reading that field in the latter.

Detecting the end of a sequence

I said above that Figure 10.1 is *almost* what we want. We're going to make one change to it. When a client calls "`new Resort(someScanner)`", it's expecting the constructor to hydrate a `Resort` object and hand it over. But consider: what if there are no more `Resort` objects in the file?

Let's zoom out a second and appreciate the context in which the `Resort` constructor will be called. We've made the design decision to store all our application's information in a single file. This means that our save file is going to have lots of resorts in it, back-to-back-to-back, each separated from the following one by the period delimiter.

This in turn means that our code in `ReservationSystem` is going to be calling "`new Resort(someScanner)`" *in a loop* to hydrate all the objects. This will work great until the point at which we hit the end of the list. What then?

In that case, the `Resort` constructor can't be allowed to finish, because *it doesn't make sense for an object to be constructed at all.* So what should the constructor do in that case? Throw an `Exception`, of course, as we learned in Chapter 5. This indicates to the client that the instantiation *failed* – there is no `Resort` object to be constructed, and the client should do what's appropriate in that event (in this case, simply stop reading from the file).

The exception we throw can either be a "plain old `Exception`," or a special type of exception that we create for this purpose. The latter approach requires the technique of *inheritance*, which we won't cover until the next chapter. But I'll give you a preview. By simply creating a one-line `NoMoreResortsException.java` file:

```
class NoMoreResortsException extends Exception {}
```

voilà, we can now throw `NoMoreResortsExceptions` in addition to plain old `Exceptions`. This increases code readability (it's obvious from the class name what this kind of exception indicates) and also allows us to distinguish this type of exception from other kinds of things that might go wrong (for instance, a bad filename or other problem accessing the filesystem).

The only other question is: how does the constructor know when it's reached the end of the sequence of resorts? Depending on how we choose to structure the file, it might be when the end of the file is reached, or when some other delimiter is encountered. Let's do the second case, and say that at the end of the resorts list, our file will contain a line with these five characters: "`-END-`". When we hit this, we'll know there are no more resorts to be hydrated.

Figure 10.2 has the modified `Resort` constructor. It's exactly the same except that after reading what it supposes is the resort's name, the constructor sanity-checks that against the string "`-END-`". If they're equal, it realizes that it didn't read a resort's name after all, but instead *it hit the end of the list.* In response, it abandons the rest of the constructor, *refuses to instantiate an object*, and throws the exception back to the caller.

`ReservationSystem`: hydration

And now, what does our overarching "read and write the whole file" code look like? Again, it depends on the structure of the file itself. Let's say it's formatted as in Figure 10.3. This file is comprised of the following parts:

```
Resort(Scanner s) throws NoMoreResortsException {
    name = s.nextLine();
    if (name.equals("-END-")) {
        throw new NoMoreResortsException();
    }
    phone = s.nextLine();
    rating = s.nextInt();
    s.nextLine();    // read and discard the " stars" part.
    cost = s.nextLine().length();
    desc = "";
    String next = s.nextLine();
    while (!next.equals(".")) {
        desc = desc + next + "\n";
        next = s.nextLine();
    }
}
```

Figure 10.2: The modified `Resort` constructor.

1. A line with the resort chain name.
2. The year this data is applicable to, followed by some other text (" `season`").
3. Another line of preamble (the hyphens), which we just have to read past.
4. A sequence of resort entries, each of which is delimited with a period.
5. The line "`-END-`" to indicate the end of the sequence.
6. A copyright notice, which we just have to read past.

Given this structure, our `ReservationSystem`'s constructor should work as in Figure 10.4 (p. 164). Check out that `try`/`catch`/`while` loop construct very carefully. It's short, but it's also a doozy. Inside the body of the `try` block, we have what appears to be an *infinite* loop: after all, `while(true)` means "forever." So we instantiate a `Resort` object and add it to our `ArrayList`, do it again, do it again, do it again...

But it's not really infinite and here's why: eventually, the `Resort` constructor is going to discover that there aren't any more resort entries in the file. When we ask it to hydrate the fourth `Resort` from Figure 10.3, the constructor won't return: instead, it'll throw

```
Holiday Inn resorts
2019 season
-------------------------
Westword Spa and Surf
312-555-1234
5 stars
$$$
This luxurious beach property is a pleasure
for all who visit. Dogs and cats welcome too!
.
Roadkill Motel
306-555-4444
1 star
$$
The 1970's-style armchairs aren't the problem:
the holes in the walls and the odorific properties
are. Traveler beware!
.
ABC Inn
123-456-7890
3 stars
$$
Nothin' fancy, nothin' broken. Your basic motel
for the budget (but hygiene-conscious) traveler.
.
-END-
Copyright (C) 2019
```

Figure 10.3: The file format for our resort reservation system.

a `NoMoreResortsException`. When that happens, we pop out of the `try` block and into the `catch` block, *which is outside the body of the loop.* So we're officially done with the loop at that point, and carry on to read the copyright information at the end and we're done.

And what does the `catch` block do? *Nothing.* That may seem pretty weird, but a moment's thought will convince you otherwise. What should we do when we reach the end of the resorts? *Simply carry on.* Even though it's an `Exception`, it's not really an "error." It just means, "oh, you wanted another `Resort` object, but actually there aren't any. Carry on with your other business, Mr. Client."

To be crystal clear, here's what happens in the `try`/`while` construct when we read Figure 10.3:

```
public class ReservationSystem {
    private int year;
    private String chainName;
    private ArrayList<Resort> resorts;

    ReservationSystem(String filename) throws Exception {
        this.resorts = new ArrayList<Resort>();
        Scanner s = new Scanner(new FileReader(filename));
        this.chainName = s.nextLine();
        year = s.nextInt();
        s.nextLine();    // Read past " season"
        s.nextLine();    // Read past line of hyphens

        // Read all the resorts.
        try {
            while (true) {
                Resort r = new Resort(s);
                resorts.add(r);
            }
        } catch (NoResortException e) {
        }

        s.nextLine();    // Read past copyright
    }
}
```

Figure 10.4: The `ReservationSystem` class's constructor.

1. It hydrates a `Resort` object, and adds it to the `resorts` `ArrayList`.
2. It hydrates a `Resort` object, and adds it to the `resorts` `ArrayList`.
3. It hydrates a `Resort` object, and adds it to the `resorts` `ArrayList`.
4. It tries to hydrate a `Resort` object...but catches the exception instead, pops out of the `try` block (and therefore also the `while` loop) entirely, and goes to the `catch` block.
5. The catch block does nothing.
6. It carries on reading the rest of the file.

Note carefully that *no exception is thrown from the constructor of the `ReservationSystem` class in this case!* Rather, the exceptions thrown by the ***Resort*** constructor are caught and dealt with by the `ReservationSystem` constructor. The only time the `ReservationSystem` constructor would throw an `Exception` is if something went wrong when trying to read the file (in which case an `Exception`, rather than a `NoMoreResortsException`, would be generated).

`ReservationSystem`: persistence

Finally, the `.persist()` method of the `ReservationSystem` class:

```
    void persist(PrintWriter pw) {
        pw.println(chainName);
        pw.println("" + year + " season");
        pw.println("----------------------------------");
        for (Resort r : resorts) {
            r.persist(pw);
        }
        pw.println("Copyright (C) " + year);
        pw.close();
    }
```

It's quite simple. The main part is simply calling the `.persist()` method on each `Resort` object, so that they can persist themselves. The stuff before and after is just for the boilerplate.

Chapter 11

Inheritance (1 of 2)

If one had to name object-oriented programming's most "killer feature," a good case could be made for **inheritance**. This specific technique for code reuse and modular flexibility underlies much of the "magic" that happens in well-architected OO programs, including most of the design patterns we'll consider in later chapters. Developers need to know it, and know it well.

Interestingly, inheritance is used for two distinct (and essentially unrelated) reasons to achieve two very different kinds of results. I call these "**top-down inheritance**" and "**bottom-up inheritance**," for reasons I'll explain; the more standard terms are **interface inheritance** and **implementation inheritance**, respectively.

Curiously, when I was a young'un taking object-oriented programming in college, we learned *only* about the latter of these, and I was mystified early in my career when I saw the former in action and had no idea what the code was doing. Only then did I realize that although bottom-up inheritance is indeed useful, top-down inheritance is the real game changer. We'll cover both in this chapter.

11.1 "Bottom-up" (implementation) inheritance

As you know, the Java API contains a class called `ArrayList`. The class diagram below shows an abbreviated version of it. Already there's one possibly unfamiliar element to you here: the literal word "`Object`". It might seem odd to learn that there is a *class* called *`Object`*, but that is indeed the case. And in fact this very class will come back later in the chapter and play a major role in Java's version of inheritance. For now, just consider that we are working with the *non*-generic `ArrayList` type – *i.e.* the user will not declare "`ArrayList<String>`" but plain-ol' "`ArrayList`" – so that the things that can be stored in it are "*any* type of `Object`." That's why we're using the most general possible word here as the argument of `.add()`, `.remove()`, *etc.*

Now suppose we were writing a program that needed to manage a bunch of list-like data, and that `ArrayList` was just the ticket...except that it was missing one or more important features. For example, maybe in addition to inserting, removing, counting, *etc.*, we also needed the ability to *count the number of unique elements* in a list. The client code we'd like to be able to write is in Figure 11.2.

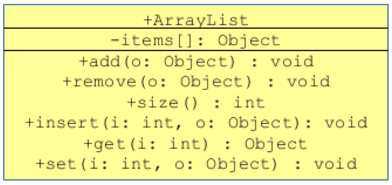

+ArrayList
-items[]: Object
+add(o: Object) : void
+remove(o: Object) : void
+size() : int
+insert(i: int, o: Object): void
+get(i: int) : Object
+set(i: int, o: Object) : void

Figure 11.1: An abbreviated `ArrayList` class.

```
public static void main(String args[]) {
   ArrayList n = new ArrayList();
   n.add("Harry");
   n.add("Ron");
   n.add("Hermione");
   n.add("Harry");
   n.add("Harry");
   n.add("Dumbledore");

   System.out.println(n.get(3));          // prints "Harry"
   System.out.println(n.size());          // prints 6

   // *We want this to print 4, but no such method:
   System.out.println(n.countUnique());
}
```

Figure 11.2: Some client code we'd *like* to be able to write in our hypothetical program.

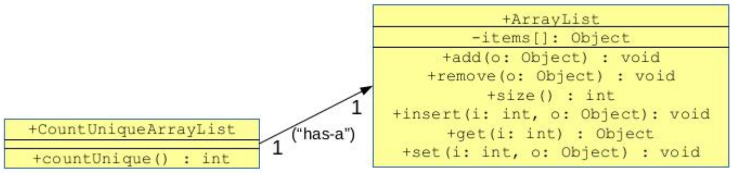


Figure 11.3: A first approach to enhancing a regular `ArrayList`.....

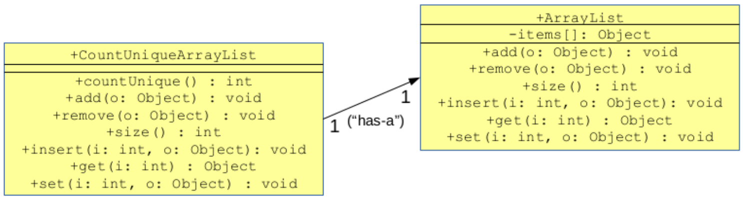


Figure 11.4: .....but this necessitates duplicating all the original methods.

This client code already works except for the last line, which of course contains a method we just made up. It's sad that `ArrayList` meets all our needs except this one teensy one.

Several ways to get around this limitation come to mind. We could use a **has-a** association between `ArrayList` and a new class of our devising, "`CountUniqueArrayList`." Each `CountUniqueArrayList` would have an `ArrayList` "under the hood" which it would use to actually store the data. Figure 11.3 gives the idea.

After writing the `.countUnique()` code, the last line of Figure 11.2 would work like a charm. Trouble is...the other lines wouldn't work anymore. ☹ Obviously we have to be able to do "normal" `ArrayList` things with our class as well as calling our new special method. So we'd have to duplicate all the other `ArrayList` methods on our new class, and have them "pass through" the arguments to the underlying `ArrayList` that it holds. The result is the unwieldy, repetitive monstrosity in Figure 11.4, which is obviously not a good solution. Here's what each of our "pass-through" methods would look like:

```
public class CountUniqueArrayList {
    private ArrayList al;

    public void add(Object o) {
        al.add(o);
    }
    public int size() {
        return al.size();
    }
    ...etc...
```

At the very least, this is clumsy, duplicative, and error-prone. But it could be even worse, if the `ArrayList` class evolves. Suppose the Java API expands to include a `.shuffle()` method on `ArrayList`, which randomly jumbles the contents of the list? Every `ArrayList` in every line of Java code in the world could instantly take advantage of that. But our stunted `CountUniqueArrayList` could not: it would have to be changed to add a `.shuffle()` pass-through method before it could do what any other `ArrayList` could automatically do.

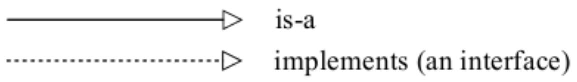


Figure 11.5: Diagrammatic elements for inheritance. (Compare with Figure 6.2, p. 87.)

The solution is to use inheritance. *Instead of "**has-a**," an inheritance association means "**is-a**."* (See Figure 11.5.) Instead of each `CountUniqueArrayList` *having* an `ArrayList`, we're declaring that a `CountUniqueArrayList` in fact ***is*** an `ArrayList`. This gives it all the rights and privileges of any `ArrayList` including all of its methods and instance variables. All the code in Figure 11.2 *instantly just flat works.* The UML equivalent is shown in Figure 11.6. Note carefully that we use an open-triangle arrowhead to designate inheritance, and that there are no other navigability, multiplicity, or role indicators.

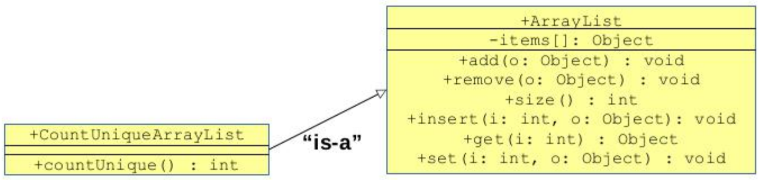


Figure 11.6: Bottom-up inheritance in action. (Note the open-triangle arrowhead.)

This may seem like cheating. Surely if we want to call a method on an object, we're entitled to see that method in the UML diagram for that object's class? Clearly, `.add()`, `.get()`, and `.size()` do not appear in `CountUniqueArrayList`'s box. But the magic of inheritance makes it work anyway. The rule is that you can call a method on an object if that method is defined on the object's class...or on any **superclass**. `ArrayList` is said to be the "superclass" of `CountUniqueArrayList`. And that brings us to a slew of equivalent terminology.

All of these expressions mean exactly the same thing:

A is-a B
A inherits from B
A specializes B
A is a subclass of B
A is a subtype of B
A is a derived class of B
B generalizes A
B is the superclass of A
B is the supertype of A
B is the base class of A

You know something's an important concept when there are a zillion equivalent terms for it. And so it is with inheritance.

Sometimes we say "A is a class, and B is its superclass." Other times we say "B is a class, and A is a subclass." These aren't contradictory statements – it's like saying I'm both a son (of my mom and dad) and also a father (of my three kids). You can totally be a son and a

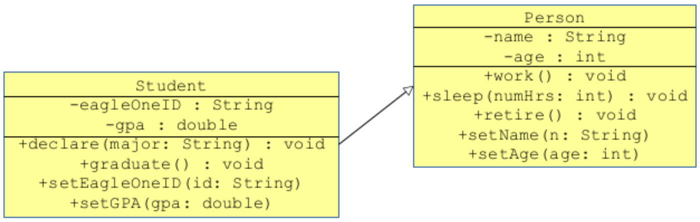


Figure 11.7: A `Student` is a special kind of `Person`, but a genuine `Person` nonetheless.

father at the same time. And a class can be both a superclass and a subclass.

By the way, if you have sharp eyes, you'll have noticed I said "or *any* superclass" a few paragraphs ago. That's because if A inherits from B, B can in turn inherit from some other class, which can itself inherit, *etc.* All the classes and their related sub/superclasses form what's known as an **inheritance hierarchy**.

Under the hood

You might wonder how this magic works behind the scenes. It's actually pretty simple. When we instantiate a `CountUniqueArrayList`, we not only allocate memory for all the `CountUniqueArrayList`-specific parts (if any), but also for its superclass parts.

Let's take a different example so that we know what the "parts" actually are.[1] Figure 11.7 shows two classes in an inheritance relationship: a `Student` **is-a** `Person`. While a `Person` in general has a `name` and and `age`, the special type of person called a "`Student`" also has an `eagleOneID` and a `gpa`.

[1]It may or may not surprise you that I honestly don't know what instance variables the `java.util.ArrayList` class has, or what they're named. This is a perfect example of the wonders of encapsulation: millions of people across the globe use `ArrayLists` every day, and do not know or care how they function internally!

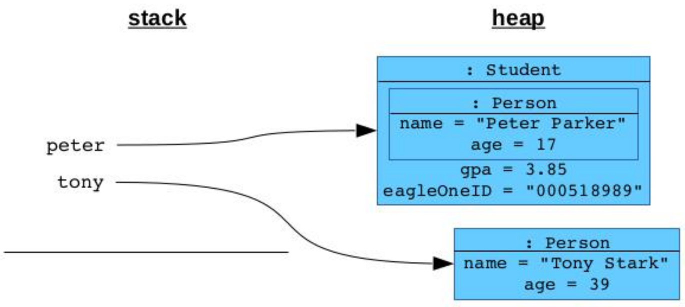


Figure 11.8: A `Student`, and an ordinary `Person`, in the heap.

Now when we instantiate each of these classes:

```
    public static void main(String args[]) {
        Person tony = new Person();
        tony.setName("Tony Stark");
        tony.setAge(39);

        Student peter = new Student();
        peter.setName("Peter Parker");
        peter.setAge(17);
        peter.setGPA(3.85);
        peter.setEagleOneID("000518989");
    }
```

the objects of each type look like Figure 11.8. See how the `Student` object has *both* the required `Student` and `Person` instance variables, since it is both a `Student` and a `Person`. To Java, the fact that the `Person` "stuff" is in a separate little chamber inside the `Student` box is just a detail.

The reason I call this technique "bottom-up inheritance" is that in order to use the special features of your new class, *you need to know you have an instance of the subclass.* In the code above, we couldn't call `.setGPA()` on an ordinary `Person`: it would have to be a `Student`. We couldn't call `.countUnique()` on just any Joe `ArrayList` – only `CountUniqueArrayLists` have that special method. Hence, the code that uses the classes views the hierarchy

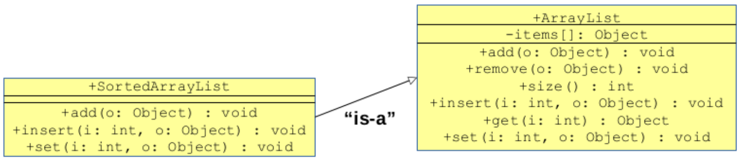

+SortedArrayList
+add(o: Object) : void
+insert(i: int, o: Object) : void
+set(i: int, o: Object) : void
"is-a"
+ArrayList
-items[]: Object
+add(o: Object) : void
+remove(o: Object) : void
+size() : int
+insert(i: int, o: Object) : void
+get(i: int) : Object
+set(i: int, o: Object) : void

Figure 11.9: Top-down inheritance in action.

"from the bottom-up"; *i.e.*, from the perspective of the subclass. In fact, if all we instantiate are `Students` (not `Persons`), then our `main()` method doesn't even need to know there *is* a superclass. To `main()`, it's all about `Students`, and the fact that you can do ordinary person-ish things to a `Student` – like set its `age`, or ask it to `.work()` or `.sleep()` – seem just like other aspects of `Students`.

The conventional term for this, "implementation inheritance," comes from the fact that the reason we're inheriting is *to steal the implementation.* Someone has already gone to the trouble of writing an `ArrayList` class – or a `Person` class – and we don't want to reinvent the wheel. So we make use of that implementation (the code in the methods) and just add whatever else we want to the mix.

11.2 "Top-down" (interface) inheritance

Now top-down inheritance is where the real action is.

Let's go back to our `ArrayList` example, and this time I'm going to write a different subclass, called "`SortedArrayList`." Figure 11.9 shows this arrangement. The open-triangle arrow and the word "is-a" are just the same. The one thing that might strike you as odd, though, is that the methods on the subclass are *also all on the superclass.* Hey, we already had an `.add()`, an `.insert()`, and a `.set()`: what good is our new class that just duplicates this? As it turns out, a *lot.*

To explain why, first let me articulate my motive for creating the `SortedArrayList` type in the first place. I might have a program that needs to store various bits of data in `ArrayLists` – a common enough task – but it needs some of those lists to *always remain in* *sorted order.* Exactly what "in order" means depends on the data type, but we could expect for `Integer`s it would be numerical order, for `String`s alphabetical order, *etc.*

It's easy to imagine a program that would need this feature. Perhaps it needs to print the various lists in some kind of reliable sequence, or to perform fast lookup via a binary search. Anyway, the point is: if I create a `SortedArrayList`, I'm doing so because I want a guarantee that no matter what I do to that list, I can always get the stuff out quickly, and sorted.

Now if you think it through, you'll realize this is a different kind of situation than we had with `CountUniqueArrayList`. Previously, we had a *new feature* we wanted to add to an existing class – an `ArrayList` could do many things, but not count its number of distinct elements, and so we tacked that feature on to the top of it. But now, we don't want a new feature so much as *different behavior for the original features.* We don't want to add any new methods, but have *the existing methods act differently.* And thus is the essence of top-down inheritance.

Before we see it in action, let's think about the implementation. You'll notice that in Figure 11.9 only *some* of the methods appear in the subclass. These are called **overridden** methods: we say that `SortedArrayList`'s `.add()` "overrides" the base class's `.add()`. Now can you figure out why those particular three methods are the ones we chose to override?

If you're sharp, you'll realize that those three methods are the only ones which, if we called the ordinary `ArrayList` version, would threaten to jeopardize the sorted nature of the list:

- If we have a sorted list, and `.add()` an item to it, our new expanded list might be unsorted if `.add()` just tacks the new item on to the end. Hence, **we must override `.add()`** with a version that adds the new element *in the correct place.*
- If I have a sorted list, and `.remove()` an element from it, the shorter list will still be sorted. So the superclass's `.remove()` doesn't mess anything up, and we can stick with it. No need

to add a version to `SortedArrayList` at all.

- Whether the list's elements are sorted or not, `.size()` acts the same for Pete's sake, so we hardly need to override that one.

- On the other hand, if we `.insert()` an element at a specific location, we'll mostly likely disturb the ordered-ness, so **we must override `.insert()`** as well, so that it puts the new element only where it truly belongs.

- The `.get()` method is an easy call: retrieving element #9 out of a list doesn't have any different behavior if the elements are sorted or not, so we leave that one out.

- Lastly, though, **we must override `.set()`** since changing one element's value could throw the ordered-ness out of kilter, requiring resorting.

If you followed all that, you'll realize that the choice of which methods to override in the subclass wasn't an arbitrary one. It was dictated directly by the behavior we wanted our subclass to guarantee and preserve. Methods whose default implementations (in the superclass) wouldn't work for our new type (in this case, those that threatened to jeopardize the order) must be replaced with versions appropriate to the subclass.

The word "override" is a good one, and it conveys almost exactly what it means, although don't make the mistake of thinking that the ordinary `ArrayList`'s `.add()` method is completely obliterated by what we've done. *Au contraire*, for plain-Jane `ArrayList`s all over the world, that original `.add()` code will still run. Only if the object in question is one of our special subtype – only if it's a `SortedArrayList` in addition to being a plan `ArrayList` – will our new method be substituted. Put another way, we're not "overriding the `.add()` method for *everybody*," just for objects of our new special type.

Test your intuition

Okay, now to drill the concept all the way home. I want to ask you a question. Before reading on, consider the code below and ask yourself "what will its output be?" (Commit to an answer before you continue.)

```
public static void main(String args[]) {
    SortedArrayList sal = new SortedArrayList();
    sal.add("Thor");
    sal.add("Bruce Banner");
    sal.add("Captain America");

    getMad(sal);

    for (int i=0; i<sal.size(); i++) {
        System.out.println("Hero #" + i + " is " + sal.get(i));
    }
}

private static void getMad(ArrayList al) {
    al.set(0,"Hulk");
}
```

To figure this out, we first observe that the code is instantiating our *new* special kind of `ArrayList`: a `SortedArrayList`. Therefore, we know that when `sal.add()` is called, *it's our new* ***`.add()`*** that will get executed. (Later in `main()`, we call `sal.size()`, and this will of course trigger the ordinary `.size()` since we didn't override that method.)

Now the big question, and the point of this exercise, is to consider what happens *inside* the `getMad()` function. Note very carefully that `getMad()` takes an argument of type ***`ArrayList`***, not `SortedArrayList`. So `getMad()`, you might say, is itself unaware that `SortedArrayList`s even exist, let alone that it's about to be given one.

So I ask you: will `ArrayList`'s ordinary `.set()` method be called, or will it be `SortedArrayList`'s new, ensure-the-list-stays-sorted `.set()` that will take over?

The critical answer is: *the subclass's method will be called, even though the function itself doesn't know it's dealing with a subclass.* `SortedArrayList.set()` will be called in this case, which will swap `Hulk` with `Captain America` to keep the list alphabetically sorted, giving this output:

```
    Hero #0 is Captain America
    Hero #1 is Hulk
    Hero #2 is Thor
```

This surprises lots and lots of folks. It certainly surprised me when I learned it (considerably after graduating college). I originally reasoned as follows: "`getMad()` was written to take an ordinary `ArrayList`, and hence its code was designed with only `ArrayLists` in mind. Surely this means that our three-hero-list, when `getMad()` receives it, will be treated just as any normal `ArrayList` would be. All that special overriding method stuff only happens when we *know* we're dealing with a `SortedArrayList` in particular."

The exact opposite is true. And it turns out that's how we want it to be. Consider this very example: what good is a `SortedArrayList` if it doesn't stay sorted? That was the whole point of the subclass! Yet we'd be threatening to violate this very principle if we ever passed it into contexts which didn't know it needed to be sorted, and hence unwittingly jumbled it up. We must guarantee that *every time* `.add()`, `.insert()`, or `.set()` is called on it, our new functionality is triggered, whether or not the user of the object even knew that.

"Masquerading" and "smuggling"

Now why do I call this "top-down inheritance?" The reason is that unlike with bottom-up inheritance, you can use objects of your new subclass *without knowing they're of that subclass, or that there even is a subclass.* As a user of the classes – like `main()`, above – you're looking at the inheritance hierarchy from the top down.

A couple of other descriptive words I like to use for this are "masquerading" and "smuggling." Here, the `SortedArrayList sal` is masquerading as an `ArrayList` – pretending to be one for the sake of the `getMad()` function. (And of course it's not actually "pretending" because a `SortedArrayList` truly **is-a** `ArrayList`.) From `main()`'s point of view, we smuggled a `SortedArrayList` into the `getMad()` function, in cognito. Little did `getMad()` know that it wasn't even dealing with an ordinary object of the base class. It was fooled by the disguise.

Why this matters

The reason this idea is so powerful is that it allows programmers to decouple the *what* from the *how*.

A chunk of code that only knows about the superclass (in our example, `ArrayList`) can dictate *what* to do with it. "First I'll `.add()` these three elements, then I'll `.remove()` one, then get the `.size()`, *etc.*"

In response to these method calls, the object itself – perhaps of a specialized subclass – decides *how* to carry out each one. An ordinary `ArrayList` tacks the new element onto the end when it's told to `.add()` one, whereas a `SortedArrayList` decides to stick it in the appropriate place to preserve the order. The original code chunk can be blissfully ignorant of how the details of `.add()`, `.insert()`, *etc.* work for any particular type of `ArrayList`.

If you're a videogamer, think of it in Smash Bros. terms. Every character in the game looks different, has different attack stats, different animations for punching and falling, a different "up smash" and "side special," and so forth. But the main game engine code that coordinates the interaction between characters on a stage doesn't have to worry about all those details. It can simply say, "hey, P1 (character #1), your player just input a dash-left. Display the appropriate animation please." The object for P1, who may happen to be Zelda, then displays her determinedly zooming to the left with her hair flowing behind her.

The game then says, "hey, P4, you just got punched. First, tell me your weight class so I know how far the knockback should be." If character #4 is Peach, her object responds, "I'm in the medium weight class." The game then says, "thanks. Now display your 'hit stun' animation from your current position up to coordinates 562, 431." The Peach object then shows her character flying through the air with her umbrella in a tizzy.

For even moderately complex programs, this ability to compartmentalize these two jobs is crucial – otherwise you end up with a 500-line function that's a mass of spaghetti code. Software engineers call this decoupling "**separation of concerns**," and it is among the most important principles in all of software development.

11.3 "Cool! Can we do both?"

A common question at this point is whether top-down and bottom-up inheritance can be combined in a single class. The answer is yes! If you create a subclass `A` of another class `B`, you could have some methods of `A` override the existing methods of `B`, and you could also have some brand new methods in `A` that weren't present in `B`.

Any code that deals with `B` objects will automatically work for `A` objects also, and your new method implementations will be called when it does. And you can write code that calls your brand new methods, as long as that code *knows it's dealing with an `A`*.

It's not super common to combine the techniques, but I've seen it done.

11.4 A word of warning

I'll finish this chapter with an observation from my years coding. Inheritance tends to be both an *underused* feature and an *overused* feature.

What I mean is that programmers (even experienced ones, sadly) sometimes fail to recognize situations in which inheritance would be appropriate, and their code becomes less elegant and more brittle as a result. Even more worrying, I've seen more than one "inheritance-happy" programmer in my career use it where it's *not* called for. And this turns out to be even worse.

Although there are shades of grey, the basic rule for knowing when inheritance is appropriate can actually be boiled down to a single principle:

☞ **Remember that "is-a" means "is-a."** ☜

The time *not* to use inheritance is when you see some code that you want to reuse, but you really *don't* have a conceptual "subtype" in mind.

An actual example: suppose your team is building a database system for 5k and 10k race results. There's a `Runner` class with inst vars like `name` and `gender` and `dateOfBirth`. Then you say, "all right: to record a performance in a particular race, we need all that information about the runner, *plus* the bib number, finish time, date and location of the race."

You may be tempted (as a colleague of mine once was) to create a `Performance` class which *inherits* from `Runner`. After all, every `Performance` object would then possess all the necessary attributes – those of the racer, and those of the race. What's not to like?

Well, there are two problems. One is conceptual: could one possibly claim that a `Performance` **is-a** `Runner`?! Obviously not. The very fact that you could call `.getGender()` on a `Performance` object, or pass a `Performance` to a `Race.register()` method defies logic.

The other problem is practical: as these two nonsensically-joined classes evolve, more pressure builds in the system that exposes the design flaw. When a data entry mistake is corrected, for example, changing a `Runner`'s name from `"Stanly"` to `"Stanley"`, only one of Stanley's performances will have its `name` corrected; the other independent copies for his other performances remain out of date. Even worse, suppose the need arises for different (legitimate) subtypes of `Runner`: `AmateurRunner` and `CompetitiveRunner`, say. Now the `Performance` class is really in a bind: it only inherits from the more general type, and would have to proliferate itself awkwardly ("`AmateurPerformance`" and "`CompetitivePerformance`") just to stay in sync.

The lesson here is that your object-oriented model should strive to faithfully reflect conceptual reality; it should not use design features in gimmicky ways to achieve short-term programming wins. Always ask yourself "does this choice of classes really make *sense*?" as your guiding question.

Chapter 12

Inheritance (2 of 2)

I split what was once a long inheritance chapter into two. So you're probably back from a snack break, a nap, or a game of Ultimate. Let's get warmed up again.

A class diagram for a "Zoo" program is in Figure 12.1. Study the code on the following page (Figure 12.2) and predict its output.

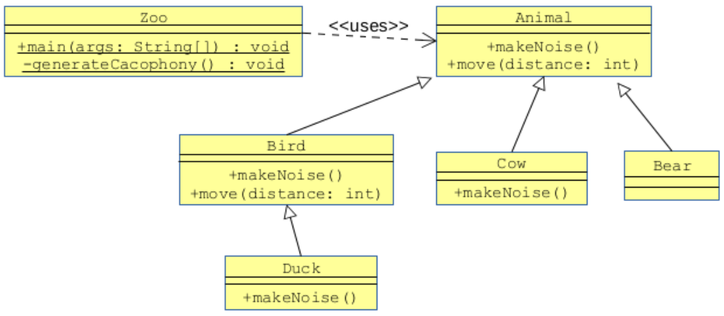


Figure 12.1: A class diagram for the Zoo program.

We have an `Animal` class with a number of subclasses, and one of them even has its *own* subclass. In each case, we override one, both, or neither of the base class's methods.

```
class Animal {
  public void makeNoise() {
    System.out.println("Growl!");
  }
  public void move(int dist) {
    for (int i=0; i<dist; i++) {
      System.out.print("tramp ");
    }
    System.out.println();
  }
}


class Cow extends Animal {
  public void makeNoise() {
    System.out.println("Mooooo");
  }
}
```

```
class Bird extends Animal {
  public void makeNoise() {
    System.out.println("Chirp");
  }
  public void move(int dist) {
    System.out.println("Flap");
  }
}


class Duck extends Bird {
  public void makeNoise() {
    System.out.println("Quack");
  }
}


class Bear extends Animal { }
```

```
class Zoo {
  public static void main(String args[]) {
    ArrayList<Animal> zoo = new ArrayList<Animal>();
    zoo.add(new Animal());
    zoo.add(new Bird());
    zoo.add(new Cow());
    zoo.add(new Bear());
    zoo.add(new Duck());
    generateCacophony(zoo);
  }

  private static void generateCacophony(ArrayList<Animal> animals) {
    for (Animal a: animals) {
      a.makeNoise();
      a.move(3);
      System.out.println();
    }
  }
```

Figure 12.2: The Zoo program.

The output of the program, as you can easily verify, is:

```
Growl!
tramp tramp tramp

Chirp
Flap

Mooooo
tramp tramp tramp

Growl!
tramp tramp tramp

Quack
Flap
```

Lots of top-down inheritance here. Notice that the `Bear` class is completely unchanged from its superclass: evidently, the generic animal behavior works fine for bears, at least as far as moving and making noise goes. This is not an error.

You also may have noticed that the `Bird`'s `.move()` method completely ignores its `distance` argument. That, too, is not a problem.

Finally, and most importantly, note that the `Duck` class does not override `.move()`, but its superclass (`Bird`) does, and so a `Duck` will "`Flap`" like ordinary `Birds` do. The rule is: when a method is called on an object, the code in its class is called, unless the class doesn't define that method. In that case, Java looks for the method in its immediate superclass, then its superclass's superclass, *etc.* all the way up the inheritance hierarchy, calling the first one it finds. This makes sense: since a `Duck` **is-a** `Bird`, it is sensible to make it move like a `Bird` rather than like a generic `Animal`.

12.1 Polymorphism

This technique goes by a funny name, by the way: **polymorphism**. It's one of those geeky-sounding words useful for slinging at parties when you want an annoying person to move away from you.

I define polymorphism (specifically, "subtype polymorphism," which is what we're dealing with here) as *transparently treating objects differently based on their type.* The word "transparently" means "without the programmer having to worry about it."

Essentially, polymorphism is another way to think about top-down inheritance. We have a generic set of operations (like "move" and "make noise"), each of which can be personalized in custom ways by specific subclasses, but which clients can simply call at the desired times without having to be aware of those subclasses. "The right thing" simply happens based on the object's type, without the programmer having to resort to a giant chain of if/else statements or some other monstrosity. This is because the language itself automatically dispatches the method call to the correct code.

In terms of the concrete zoo example, polymorphism basically says: "the way you direct any animal to make noise is the same: telling a duck to make noise uses the exact same code as telling a cow to make noise. Yet different things happen in each case, because of how Java enables the subtype-customization to take place."

`instanceof` is evil

It may help you understand this if I contrast it with a *non*-example of polymorphism.

Suppose we have a `Pokemon` class, instances of which represent various fictional fighting critters and their moves and stats. We also have a `Power` class used to represent a particular superpower like "thunder shock" or "hyperbeam." Each special type of power will be its own subclass of `Power`, in true object-oriented inheritance style. And for this simple example, we'll say that each `Pokemon` has just one "primary power" that it can use in combat. Whenever a `Pokemon` object's `.attack()` method is called, it will use its primary power on the foe it is passed.

Figure 12.3 shows the **WRONG** way to code this. The `.attack()` method itself, in the `Pokemon` class, is scrutinizing its `primaryPower` object, figuring out what subclass it is and responding appropriately. You can see the (evil) Java `instanceof` operator in use here.

It is a way of determining at run-time whether or not an object is of a particular class. **Using `instanceof` is almost always bad programming practice** since it violates encapsulation, reduces modularity, and eschews polymorphism. Among other things, this design creates a gigantic (and ever-growing, as more powers are added to the program) `if`/`else` chain to do the work that Java already does automatically. And that huge type-checking mechanism, besides being unwieldy and error-prone, is plopped right inside the `Pokemon` class, which is not the right place for it. (That code concerns various types of *powers*, not *Pokémon*.)

```
public class Pokemon {

    private Power primaryPower;

    public void attack(Pokemon foe) {
        if (primaryPower instanceof Thundershock) {
            // do the "thundershock" stuff to the foe
        }
        else if (primaryPower instanceof RoarOfTime) {
            // do the "roar of time" stuff to the foe
        }
        else if ...
    }
}

public class Power {
    ...
}

public class Thundershock extends Power {
    ...
}

public class RoarOfTime extends Power {
    ...
}
```

Figure 12.3: The **WRONG** way to implement Pokémon powers, using the `instanceof` operator.

Figure 12.4, on the other hand, shows a good, healthy, proper OO design for the Pokémon model. None of that `if`/`else` junk is even necessary, and the `Pokemon`'s `.attack()` method becomes a one-liner, as it should be. Java will automatically call the proper "thunder shock" code, "roar of time" code, "hyperbeam" code, or whatever, without us having to try to do its work for it. And adding a new `Power` is just a matter of creating a new, encapsulated subclass, with its own `.useAgainst()` code. Neither the `Power` superclass nor the `Pokemon` class need be disturbed by this addition. It all works seamlessly.

```
public class Pokemon {

    private Power primaryPower;

    public void attack(Pokemon foe) {
        this.primaryPower.useAgainst(foe);
    }
}

public class Power {
    public void useAgainst(Pokemon foe) {
        // use the basic, default power on the foe
    }
}

public class Thundershock extends Power {
    public void useAgainst(Pokemon foe) {
        // do the "thundershock" stuff to the foe
    }
}

public class RoarOfTime extends Power {
    public void useAgainst(Pokemon foe) {
        // do the "roar of time" stuff to the foe
    }
}
```

Figure 12.4: The **right** way to implement Pokémon powers, using subtype polymorphism.

12.2 Getting abstract

Abstract methods

Okay, back to the Zoo example from p. 183. For each type of animal, it was up to the subclass to decide whether to override a method or not. If the subclass was happy with the base class's behavior (like `Cow` was with `.move()`, or `Bear` was with everything) it could just ignore that method and default to ordinary animal behavior.

What if we had a method, though, where there was no reasonable default behavior to define? Let's add a method `.getNumChromosomes()` which will return the number of DNA molecules in that species. Unlike making noise, which we might plausibly say is "growling, unless further specified for a particular animal," perhaps there isn't any "default number of chromosomes" that makes sense.[1]

Note that we still want to be able to call `.getNumChromosomes()` on *any* `Animal`, no matter what type. So it's no good to just leave the method out of the `Animal` class and only define it on subclasses.

The solution to our dilemma is to make `.getNumChromosomes()` an **abstract method**. "Abstract" means that even though it's defined in a superclass – with a name, parameter list, and return type – there's not actually any *code* for it there! Here's what it looks like:

```
class Animal {     // (not finished yet...)
    abstract public int getNumChromosomes();
    ...
}
```

It looks a little strange, with a semicolon prematurely truncating the method signature. But by doing this, we're declaring that even though there's no method *body* for `.getNumChromosomes()` in the `Animal` class, we still want to be able to call it on an `Animal`. Individual subclasses provide their own implementation; for example:

[1]Consider: a fruit fly has only 8 chromosomes, while a tasmanian devil has 14, a human has 46, a potato 48, a silkworm 56, and a catfish 104. Given these examples, what default value would have any merit?

```
class Cow extends Animal {
    public int getNumChromosomes() {
        return 60;
    }
    ...
}
```

Notice that the subclasses do *not* include the keyword `abstract` on the method.

Abstract classes

Now I know what you're thinking. "This is all fine, but what if we have a plain-old-`Animal` object and call `.getNumChromosomes()` on it?"

```
    Cow c = new Cow();
    int numc = c.getNumChromosomes();   <-- returns 60
    Animal a = new Animal();
    int numa = a.getNumChromosomes();   <-- ???
```

There's no code to run for a generic `Animal`, so what value would we get back?

This is indeed a dilemma, and Java solves it in the only sensible way: it says *you can't make a plain-old-`Animal` anymore.* The rule is: if any method of a class is abstract, **the class itself must be abstract**, which means it is *un-instantiatable.* In other words, the Java compiler forces us to amend our `Animal` class as follows:

```
abstract class Animal {
    public void makeNoise() {
        System.out.println("Growl!");
    }

    public void move(int distance) {
        for (int i=0; i<distance; i++) {
          System.out.print("tramp ");
        }
        System.out.println();
    }

    abstract public int getNumChromosomes();
}
```

If we ever try to write the code "`new Animal()`", Java will (rightly) stop us with a compilation error.

```
Animal.java:5: error: Animal is abstract; cannot be instantiated
        new Animal();
        ^
1 error
```

Note that every instantiatable subclass needs to be non-`abstract`, and therefore *it must have a body for every one of its methods*, defined in either itself or in one of its superclasses. For example, as soon as we add the `abstract .getNumChromosomes()`, our `Cow.java` suddenly won't compile either:

```
Cow.java:4: error: Cow is not abstract and does not override abstract
method getNumChromosomes() in Animal
    class Cow extends Animal {}
    ^
1 error
```

So we'll need to define `.getNumChromosomes()` in `Cow` and `Bear` (or else make them `abstract`). What about `Duck` and `Bird`? It depends on which of these classes (if either) we want to be `abstract`. If we want to be able to "`new Bird()`" and also "`new Duck()`", then at *least* `Bird` will have to provide a `.getNumChromosomes()` method. (And `Duck`'s would be optional. If in fact `Duck`s have a different number of chromosomes than ordinary `Bird`s, we'd want to also make one on the `Duck` class.) If, on the other hand, we're happy for `Bird` to be abstract (and thus un-instantiatable), we could choose to define it *only* on `Duck` and call it a day. It all depends on what makes sense for the domain.

Putting all this abstract stuff together, here are the rules:

1. A class cannot be instantiated if it's declared `abstract`. (Only its subclasses possibly can be.)
2. If there is any `abstract` method on a class, the class itself must be declared `abstract`.
3. If a class `A` inherits from some other class `B`, and `B` has abstract method(s) on it, `A` must provide implementation(s) for all of those methods, or else `A` be declared `abstract` as well.

UML notation

There are two different ways to mark a method or class as `abstract` in UML. One is to put the name of the method or class in italics. That works well if you're using a design tool like ArgoUML or Rationale Rose to create your diagrams. Otherwise, if you're doing it by hand, you add a ("≪abstract≫") stereotype. I've shown both approaches in Figure 12.5.

12.3 That's just super

Calling a superclass's methods with "`super.`"

Occasionally, it will make sense for a method in a class to explicitly call a method in its superclass. For instance, suppose we said, "When asked to make noise, a `Duck` should '`Quack`' but then *also* do whatever ordinary `Birds` usually do." We would write that as follows:

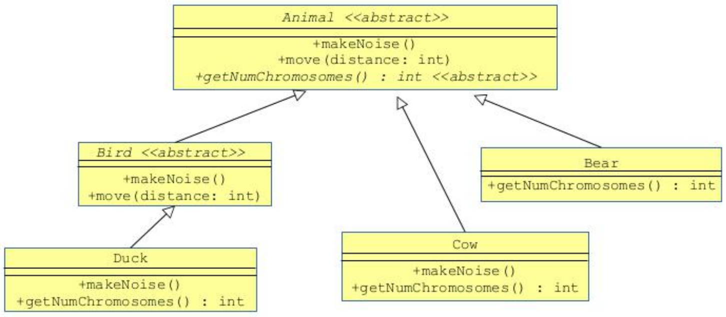


Figure 12.5: Using *italics*, and the ≪abstract≫ stereotype, to indicate abstract classes and methods.

```
class Duck extends Bird {
    public void makeNoise() {
        System.out.println("Quack");
        super.makeNoise();
    }
}
```

The "`super.`" prefix (pronounced "super dot") says to call the version of `.makeNoise()` that's defined in the `Bird` class, not the `Duck` one. (And a good thing, too: otherwise we'd have an infinite loop with `.makeNoise()` repeatedly calling itself!) Written this way, a `Duck` will make this noise:

```
Quack
Chirp
```

As another example, we might define a class `Chipmunk` that moves twice as rapidly as normal animals. If we defined its `.move()` method thus:

```
class Chipmunk extends Animal {
    public void move(int distance) {
        int doubleDist = distance * 2;
        super.move(doubleDist);
    }
}
```

then calling `.move(3)` on a `Chipmunk` object would print this:

```
tramp tramp tramp tramp tramp tramp
```

By the way, students often have a question here, and maybe you do too. They ask whether they can call `super.super.move()` to invoke the "grandparent's method." The answer is **no**, for reasons having to do with encapsulation. It's fine for a `Duck` to know it's a subclass of `Bird`, but it shouldn't know what `Bird`'s superclass is, nor even that `Bird` *has* a superclass. That knowledge (which we would be baking in to our program through a "`super.super.`" call) is outside the realm of the `Duck` class, and therefore should not be known to it. If later, for instance, we refactor our code and make `Bird` its own class instead of inheriting from `Animal`, `Duck` would break if it had a "`super.super.`".

Invoking a superclass's constructor with "`super`" (no dot)

You've known ever since p. 40 that whenever an object is instantiated, one of that class's constructors is called. This is indeed the case, but here's a new wrinkle: when an object is instantiated, *a constructor in its superclass is also called.* This is because in order to "set up" the `Duck` stuff (initialize its inst vars, *etc.*), we also need to set up its `Bird` stuff, and for that matter its `Animal` stuff. Since a `Duck` **is-a** `Bird`, which in turn **is-a** `Animal`, it's imperative that we have a valid `Animal` object before trying to make it a special kind of `Animal` (and similarly, a valid `Bird` before we make a special kind of `Bird`).

This usually happens automatically, but can need some special coaxing if you're passing arguments to constructors. An example is the code snippet in Figure 12.6 (whose class diagram is in Figure 12.7). Here, the only constructor for the `Person` class is one that takes a `String` argument; hence, we must pass it a `String` if we want to instantiate one. In the `Student` constructor (which takes a couple arguments of its own) we call "`super(name)`" which passes the `name` the subclass was given and passes it up to its superclass's constructor. In the `Prof` constructor, we take a `"Dr. "`

on to the front of the name before passing it the superclass. Note that there's no "dot" after these `super()`s; that's how we indicate we want to call the constructor, rather than an ordinary method.

Somewhat weirdly, in Java the call to `super()` must be the *first* line of the constructor. That's because Java says, "a `Student` is a special type of `Person`. So if you want to be a `Student`, you have to first be a fully-baked `Person`. Only then will we get around to the `Student`-specific stuff." (Note this restriction does *not* apply to "`super.`" ("super dot") method calls; they can be put anywhere.)

```
class Person {
    private String name;
    protected Person(String n) {
        this.name = n;
    }
}

class Student extends Person {
    private double gpa;
    private int year;
    Student(String name, int year) {
        super(name);
        this.gpa = 0.0;
        this.year = 1;  // Freshman
    }
}
```

```
class Prof extends Person {
    private String dept;
    Prof(String n, String d) {
        super("Dr. " + n);
        this.dept = d;
    }
}
```

Figure 12.6: The interaction between constructors when inheriting.

One other thing on this `Person`/`Prof`/`Student` example: we have defined the `Person` constructor to have `protected` visibility. As you may recall from Figure 6.11 on p. 102, `protected` visibility (denoted in UML with a hashtag) means that it can only be called by a class in the same package *or a subclass*. It's useful in situations like this that involve inheritance: by making the `Person` constructor `protected`, we're saying that no one outside the package can instantiate a plain-old `Person`: instead they must instantiate a subtype, which can then call the constructor from below. This is related, but not quite the same, as declaring a method `abstract`. (The difference being that `abstract` classes can't be instantiated *anywhere*, even in the same package.)

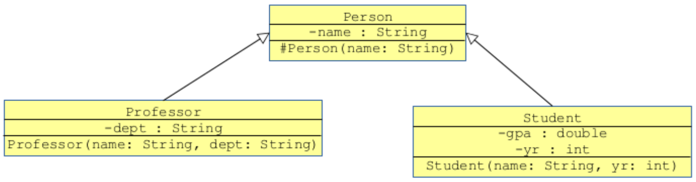


Figure 12.7: The class diagram for Figure 12.6.

12.4 Pure abstraction: Java interfaces

We've now seen several examples of superclasses. Sometimes they have all concrete (opposite of abstract) methods, and a subclass can pick and choose which ones it wants to override. Sometimes, they have one or more `abstract` methods, which the subclass *must* override, or else be un-instantiatable.

What if we push that idea all the way in the abstract direction: a "pure" abstract class, with only abstract methods? Is that any use?

Yes, and probably more than you realize right now. I'll get to the reasons this is so powerful in a page or two, but for now let's just learn the syntax. In Java, instead of a pure abstract class, you can create something called an **interface**, which is essentially the same thing but which gives considerable more flexibility. An interface is a template of sorts, with *no instance variables* and *no method bodies*. Classes that **implement** an interface promise to create a method body for each of the methods in the interface.

In UML (see Figure 12.8), an interface looks almost exactly like a class. The only differences are: (1) there are (sometimes[2]) only two compartments instead of three, and (2) the **stereotype** ≪interface≫ appears in the name box (recall "stereotypes" from p. 88).

We indicate "implements an interface" with a *dashed* line that otherwise looks exactly like an inheritance arrow (recall Figure 11.5

[2]Designers aren't consistent about this, I've found. Some do include all three compartments for an interface, even though the middle one must by definition be empty.

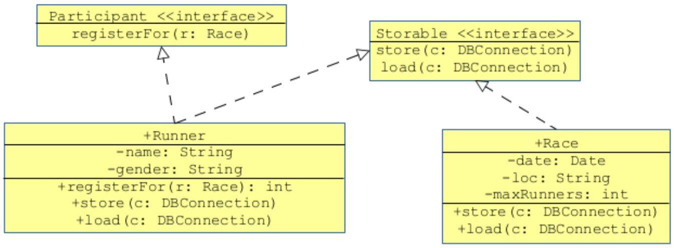


Figure 12.8: Two classes that implement interfaces.

from p. 170).

In Figure 12.8, both of the classes in our racers application need to store and load objects from the database. Hence, both of them implement the `Storable` interface, which defines two methods, one for storing and one for loading. The code for that much would look like:

```
interface Storable {
    void store(DBConn c);
    void load(DBConn c);
}

class Race implements Storable {
    private Date date;
    private String loc;
    private int maxRunners;
    public void store(DBConn c) {
        ...
    }
    public void load(DBConn c) {
        ...
    }
}
```

```
class Runner implements Storable {
    private String name;
    private String gender;
    public void store(DBConn c) {
   ...
    }
    public void load(DBConn c) {
   ...
    }
}
```

Some Java syntax idiosyncrasies: an `interface` has only method signatures (return type, method name, parameter lists) followed by a semicolon, just as with abstract methods. However, the methods are *not* marked as `abstract`, nor `public` nor anything else. Note also that we don't say the class "`extends`" the interface, but rather "`implements`" it.

In the same figure, we have Runner also implementing another interface: Participant. Presumably there are also other types of participants (volunteers, walkers, wheelchair racers, walkers, rock bands that play on the sidewalks) that can .register() for races, and if they want to be able to participate, their classes must implement this one-method interface. We augment the code as follows:

```
interface Participant {
    int registerFor(Race r);
}

class Runner implements Storable, Participant {
    private String name;
    private String gender;
    public void store(DBConn c) {
         ...
    }
    public void load(DBConn c) {
         ...
    }
    int registerFor(Race r) {
        ...
    }
}
```

Take note of the fact that the Runner class definition has *two* interface names in its declaration line, separated by a comma.

And that takes us, at last, to the reason Java interfaces matter: ***because a class can implement more than one of them.***

You might be thinking, "okay, but couldn't we just have inherited from more than one abstract class, and accomplished the same thing without this new 'interface' notion?" The answer is **no**: Java does not allow **multiple inheritance**, which is when a class has more than one superclass. The reasons are somewhat lengthy to explain, and will be omitted here, but suffice to say that languages that choose to support multiple inheritance (C++ is among them) have some knotty problems to solve. For instance, what if class A inherits from both B and C, and both B and C have their own .doThis() method. What code, then, would this line trigger?

```
A a = new A();
a.doThis();       <-- ? Whose "doThis()"?
```

How can it choose between `B`'s and `C`'s perfectly acceptable implementations of `.doThis()`? Or should it run both of them, and if so in what order?

James Gosling and the other designers of Java took a look at this problem and decided that multiple inheritance wasn't worth the hassle. I happen to think they were right, and for one simple reason: although programmers often want to use multiple *top-down* inheritance, they rarely want to use multiple *bottom-up* inheritance. And it's the latter that gives rise to all the paradoxes and strange corner cases. As long as a class simply wants to masquerade as one of several different things (as opposed to wanting to steal method implementations and inst vars from more than one superclass), "multiple inheritance" turns out not to be a big deal. One of Java's big wins was allowing multiple interfaces to be implemented, not multiple classes to be inherited from, thereby sidestepping the whole issue.

So the bottom line is that in Java, a class can *inherit* from only one class, but it can *implement* as many interfaces as it likes. If you do both, you declare yourself like this:

```
class Runner extends Athlete implements Participant, Storable {
    ...
}
```

Put another way, you only have one inheritance relationship to "spend," so use it wisely. Only inherit from something if it truly is an "**is-a**" relationship, and there's meaningful code you get to reuse by so doing.

Example: `java.lang.Comparable`

Here's an example of the kind of thing that interfaces make easy. Recall our `ReservationSystem` example from section 10.3 (on p. 156).

We had a class called Resort that represented high-falutin' vacation destinations. It had inst vars like name, desc, phone, rating (from 1 to 5 stars) and cost (from \$ to \$\$\$\$).

We also sketched a ReservationSystem class that would travel agents to browse available rooms, book hotels for customers, *etc.* We'll use a **has-a** relationship for that (see Figure 12.9).

Now one of the things ReservationSystem is likely to want to do is *sort* the Resort objects in its list. There's a function in java.util.Collections class called sort() that does just that, in fact. We'd like to write code like this:

```
class ReservationSystem {
    private ArrayList<Resort> resorts;
    ...
    private void printAvailableResorts() {

        // Determine availability...

        java.util.Collections.sort(resorts);

        // Print results...
    }
}
```

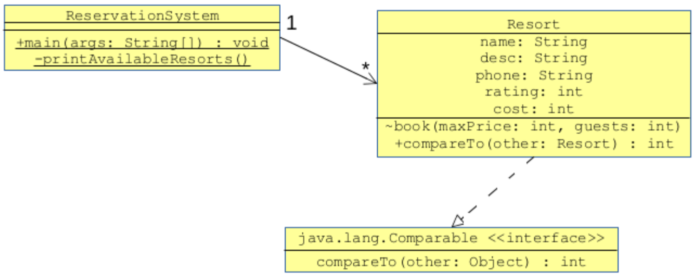

ReservationSystem
+main(args: String[]) : void
-printAvailableResorts()
1
*
Resort
name: String
desc: String
phone: String
rating: int
cost: int
~book(maxPrice: int, guests: int)
+compareTo(other: Resort) : int
java.lang.Comparable <<interface>>
compareTo(other: Object) : int

Figure 12.9: Using the Comparable interface to enable sorting.

The trouble is, even though Collections has an efficient, bug-free sorting algorithm all coded up for us to use, it doesn't know *what* *"sorted" means for `Resorts`.* For a list of integers, we'd expect "sorted" to mean "in numerical order." For `Strings`, we'd expect alphabetical order. But what about `Resorts`?

It turns out that `Collections.sort()` works for any collection of objects that implement the `Comparable` interface from `java.lang`. That interface has just one method, `.compareTo()`, which compares the object it's called on with another object. The specification says it should return a negative number if `this` should come before the argument in sorted order, and a positive number if the argument should come first.

Let's say we want to sort our `Resorts` alphabetically by name. Then we could just write:

```
class Resort implements Comparable<Resort> {
    private String name, desc, phone;
    private int rating, cost;
    ...
    public int compareTo(Resort r) {
        return this.name.compareTo(r.name);
    }
}
```

It's a weird-looking one-liner, but what this `.compareTo()` implementation does is pass the buck to its `name` field. It says, in effect, "if you want to know whether I come before or after the resort `r`, just look at whether my `name` comes before or after `r`'s `name` alphabetically."

Or, perhaps I want to sort by decreasing rating (5 stars first, 1 star last). Then I could write:

```
class Resort implements Comparable<Resort> {
    private String name, desc, phone;
    private int rating, cost;
    ...
    public int compareTo(Resort r) {
        return r.rating - this.rating;
    }
}
```

Now, if I'm a 4-star hotel and `r` is a 2-star hotel, `r.rating - this.rating` will be a negative number, putting me first.

You could detect equal ratings and break the tie by the cost, or sort by phone number, reverse length of `description`, or anything else you dream up. The magic here is that we decoupled the sorting algorithm from the details of how objects are compared. The `Collections.sort()` method is perfectly happy as long as we keep our promise to implement `Comparable` and provide a way to compare two `Resort` objects.

12.5 `java.lang.Object`

One other difference between C++ and Java is that Java has a **single-rooted inheritance hierarchy**. This means that in Java, if you trace *any* class's ancestry up from superclass to superclass, you eventually reach a common point. That point is called the `Object` class from the `java.lang` package. *All* classes, regardless of who wrote them, are ultimately subtypes of `Object`.

It doesn't seem that way when we look at the code, since we saw "`class Bear extends Animal`" yet "`class Animal`...(extends nothing, presumably)". But if you don't explicitly list a subtype via `extends`, Java implicitly substitutes `Object` as the superclass.

This has a couple of nice benefits. For one, there are methods on the `Object` class that all objects are therefore guaranteed to inherit; among them are `.equals()` and `.toString()`. For another, it is possible to treat *all* objects indistinguishably in certain contexts. We saw this with the pre-generic collection classes like `ArrayList`. The reason we can have a plain `ArrayList` hold absolutely anything – say, two `Strings`, three `Ballplayers` and a `Duck` – is that the list is treating them all as the ultimate supertype (`Object`), in true top-down inheritance fashion. In C++ there isn't really any way to do that, since if you have two classes that don't explicitly inherit from a common type, then as Steely Dan says, they simply "got nothing in common."

Chapter 13

The Factory pattern

Now that we've covered inheritance, we're in a position to understand the next simple-yet-ubiquitous design pattern, called **Factory**. It's a pretty easy one to grasp. Simply put, a factory is *a class whose purpose is to instantiate objects.*

Up to now, we've used the `new` operator directly in order to instantiate. If we want a new `Ballplayer` object, we `new` one up:

```
Ballplayer joe = new Ballplayer("Dimaggio","OF");
```

We're now going to outsource this instantiation process to a special class called a `BallplayerFactory`. It'll seem like unnecessary wiring at first, but there are advantages that come to light when the instantiation process is more complicated. Our new line of code will be this:

```
Ballplayer joe =
    BallplayerFactory.instance().create("Dimaggio","OF");
```

The sharp-eyed reader will see the `.instance()` and wonder if this is using the **Singleton** pattern. The answer is yes! In fact, *factories are nearly always singletons.* This is simply because although our baseball simulator will create lots of `Ballplayer`s, it will only need one `BallplayerFactory`.[1]

[1]As with all design patterns, you should use standard nomenclature. If the

Patterns interleave and play off one another all the time. Here, we have a factory class that uses the classic Singleton pattern:

```
class BallplayerFactory {

    private static BallplayerFactory theInstance;

    public static synchronized BallplayerFactory instance() {
        if (theInstance == null) {
            theInstance = new BallplayerFactory();
        }
        return theInstance;
    }

    private BallplayerFactory() {
    }

    public Ballplayer create(String name, String position) {
        return new Ballplayer(name, position);
    }
}
```

Badda-boom.

Now I know what you're thinking. You're thinking "wow, that's a whole lot of code to do nothing but "`new` up a `Ballplayer`," which we were doing before with one line of code and a lot less work. What's the point of all this?"

The answer is that oftentimes *the code which needs to instantiate an object isn't aware of what specific subtype of object it needs.* Read that sentence again and let it sink in. This type of decoupling – one part of the code that directs objects to do things without being aware of subclasses, and a different part of the code that implements subtype-specific behavior – is one of the important benefits a properly-designed object-oriented program can bring.

word "factory" seems clunky or contrived to you, I get it, but accept it as part of the OOP culture and use it. Don't try to come up with your own synonym of "factory" (like "creator" or "instantiator") and use that instead, since you'll just confuse your fellow programmers. The word "factory," like the word "singleton," is baked into the software development community's consciousness, and thus serves as an excellent terse-yet-precise communication of purpose.

13.1 Top-down inheritance and the Factory pattern

This is in fact the launch point of top-down inheritance. We've seen that with top-down inheritance, the client code can treat all specific subtypes (like `Cow`, `Duck`, *etc.*) in exactly the same way – in fact, it doesn't even have to be aware that there *are* any subtypes. To the client code, all that exists are `Animals`, and those objects can be directed to move, make noise, or anything else.

That was all great, but one thing we didn't address was "how do those objects come into existence in the first place?" Somewhere the specific subtype *has* to be mentioned in the code, or else there's no way to "`new`" it. How can we be blissfully ignorant of the subtypes if we're the one who has to instantiate them?

The answer is the Factory pattern. With it, we turn over control of the actual instantiation to the factory class, rather than burdening the client code with it.

Suppose that our simulator is more sophisticated than the original toy example from Chapter 4. Different positions have different kinds of stats: pitchers (with "`koDominance`") are completely different than position players (who have `numHits` and `numAtBats` instead). It might make sense to use an inheritance hierarchy here with subclasses of `Ballplayer`, as shown in Figure 13.1.

Our factory can now instantiate *the right kind* of `Ballplayer` when it is asked to:

```
class BallplayerFactory {
    ...

    public Ballplayer create(String name, String position) {
        if (position.equals("P")) {
            return new Pitcher(name);
        } else {
            return new PositionPlayer(name, position);
        }
    }
}
```

```
class Ballplayer {
    protected String name;
    private int uni;
    private int salary;
    private String handedness;  //  "L" or "R"

    public abstract double estimatedMarketValue();
}

class Pitcher extends Ballplayer {
    private double koDominance;
    private int numKos;

    public Pitcher(String name) { this.name = name; }
    public double estimatedMarketValue() {
        return koDominance * 20000000;
    }
}

class PositionPlayer extends Ballplayer {
    private String position;
    private int numHits;
    private int numAtBats;

    public PositionPlayer(String name, String pos) {
        this.name = name;
        this.position = pos;
    }
    public double estimatedMarketValue() {
        return numHits * 100000 - numAtBats * 1000;
    }
}
```

Figure 13.1: A small inheritance hierarchy for baseball players.

Depending on the position ("`P`" for pitcher, "`OF`" for outfielder, *etc.*) the factory instantiates the proper subtype of `Ballplayer` and returns it. The client code doesn't need to know anything about those subtypes. Notice that the return value of `.create()` is the general type `Ballplayer`, not any of the specific types. If we later add additional subclasses, only the factory class has to be updated, not the code that (unknowingly) uses them.

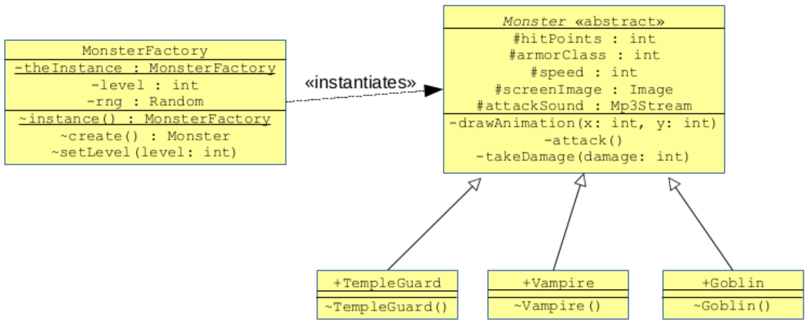


Figure 13.2: The factory pattern in a randomly-generated setting.

13.2 Random type generation

In games and simulations, it's common to instantiate objects according to some random pattern, rather than specifying each type of object deterministically.

Imagine a fantasy-genre videogame in which the player acts as a swordsman or Valkyrie and wanders through dungeon levels looking for monsters to slay. We might have a `MonsterFactory` class to generate monsters as they are encountered. Maybe on the first (and easiest) level of the game, half of the monsters we create are `Goblins` and the other half are `TempleGuards`. And both of these have "easy" statistics – *i.e.*, low values for `hitPoints`, `armorClass`, and `speed`. When the player gets to level two, however, not only will the stats of these basic creatures get buffed, but now an occasional `Vampire` will enter the mix.

The Factory pattern makes this easy. All we need to do is encapsulate the functionality for creating monsters into the `.create()` method, as shown in Figure 13.3. Then, whenever our program needs to create a new bad guy, this line of code suffices:

```
Monster newMonster = MonsterFactory.instance().create();
```

When the player completes a level and is ready for the next challenge, our client code simply calls:

```
int currLevel = MonsterFactory.instance().getLevel();
MonsterFactory.instance().setLevel(currLevel + 1);
```

and away we go.

Hopefully you're getting the gist, here: if the instantiation process for an object is a bit complex, it makes sense to separate it into its own class. This way, you don't have to duplicate the logic in separate places, and you don't have to clutter your client code with a giant `switch` or `if-else` construct right in the middle of instantiating and using an object.

```
class MonsterFactory {
    private int level;
    private java.util.Random rng;

    ...singleton stuff...

    Monster create() {
        Monster m = null;
        switch (level) {
            case 1:
                double randomNumber = rng.nextDouble();
                if (randomNumber < .5) {
                    m = new Goblin();
                } else {
                    m = new TempleGuard();
                }
                m.setHitPoints(rng.nextInt(8));
                m.setArmorClass(rng.nextInt(8));
                m.setSpeed(rng.nextInt(5));
                break;
            case 2:
                double randomNumber = rng.nextDouble();
                if (randomNumber < .4) {
                    m = new Goblin();
                } else if (randomNumber < .8) {
                    m = new TempleGuard();
                } else {
                    m = new Vampire();
                }
                m.setHitPoints(rng.nextInt(8) + 5);
                m.setArmorClass(rng.nextInt(8) + 4);
                m.setSpeed(rng.nextInt(10));
                break;
        }
        return m;
    }
}
```

Figure 13.3: A factory to generate baddies randomly, based on the game's level. (Recall the `java.util.Random` class from Section 4.3 on p. 71.)

Chapter 14

Team software development

At some point in your career (perhaps now), you will begin to work on projects that are too big for any one person to complete in an acceptable amount of time. The solution, of course, is to work on a **team** with other software developers. Working on a team brings up a host of other issues, some technical and some social.

14.1 Team-based version control with `git`

Waaaay back in chapter 1 (p. 17) I briefly introduced the `git` version control system. Hopefully you've been using it all along to do the simple process of "committing" code to your repo in individual snapshots. This is a habit you'll want to continue to ingrain in your cerebral cortex. Commits are the building blocks of any version control system, including `git`; without them, you don't have anything to work with.

Now it's time to learn a little more about `git`, and especially how it works in a team environment.

Figure 14.1 shows the environment you've been using so far: a single developer, with a single **repo**. I use the term **workspace** to mean "the directory (and possibly subdirectories) in which the developer's files actually exist." In Figure 14.1, the developer is Filbert. His workspace is shown as a yellow oval, which matches the ovals from way back in Figure 1.1 (p. 6) that represented ordinary

Linux directories. This yellow oval contains the "bleeding edge" of what Filbert is working on: as soon as he saves any file from `vim`, the file in that directory is instantly updated based on what he just typed, warts and all.

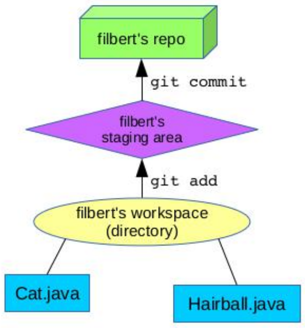


Figure 14.1: A single developer's repo and workspace.

Filbert's personal git repo is shown as a green box. At this point, you should view a repo as a sort of "mysterious" thing that somehow maintains a record of all the previous changes to all the project's files, yet in a way you don't need to understand. Using `git` commands from the command line is the only way we will inspect and command it.[1]

Also shown is a purple diamond called the **staging area**. Normally you don't think too hard about the purple diamond explicitly, but it is there, and the command "`git status`" will only fully make sense if you recognize its use. Essentially, the staging area is for changes that the developer is "about to commit" (or "fixin' to commit," as they sometimes say in the South) but hasn't actually pulled the trigger on yet. When you execute the two commands "`git add` filename" and "`git commit -m "My commit message"`", back to back, the `add` puts a snapshot of the change to the staging area, and `commit` actually records it in the repo. (If you're in the habit of using "`git commit -a`" to commit your files, it effectively does

[1]If you're curious, it is maintained in a hidden directory called "`.git`" – note the initial dot – inside the top-level directory of your workspace. If you `cd` in there, you'll see all kinds of crazy stuff under the hood. Do *not* modify any of it, or you'll probably break your repo and lose all its contents!

both the add and the commit all in one step.)

Beware a common pitfall with "`git commit -a`", though: it only commits changes to *existing* files, not *new* files. If you create a brand new `.java` file in your workspace, representing a new class, you must explicitly "`git add`" it to your repo before committing it. I can't tell you how many times one of my colleagues (or myself *shame*) has broken a build by committing all of their changes *except* for the new files.

Anyway, just to repeat the basic instructions for reproducing Figure 14.1:

1. The command "`mkdir someDirName`" creates the workspace, under whatever directory you're currently in (which can be seen via "`pwd`").
2. The command "`cd someDirectoryName`" actually goes into that directory (remember that `mkdir` alone does *not* change your current directory).
3. The command "`git init .`" creates the green box and the purple diamond.
4. You use "`vim`" to create files like `Hairball.java` and `Cat.java` directly in your workspace (blue rectangles).
5. When you want to snapshot the current version of one of your files, in anticipation of doing a commit, you type "`git add nameOfFile`". This adds the current contents to the purple diamond. You can now proceed editing further, or go straight to the commit.
6. To actually commit, type "`git commit -m "My message"`".[2]

Understanding `git status`

Two extremely common commands for inspecting your workspace are "`git status`" and "`git log`". Let's look at each in turn.

The `git status` command tells you the current state of your workspace as compared with your repo. The output will look like this:

[2]Or, in place of steps 5 *and* 6, you can do the shortcut operation "`git commit -a -m "message"`", but only *if* all the files you're changing were already explicitly "`git add`"ed at some point.

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    modified:   Cat.java

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in directory)
    modified:   Cat.java

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Cat.class
    Hairball.class
    Hairball.java
```

Several things of interest here:

- The "`On branch master`" means you're on the main, default code "trunk." Later in your career, you'll discover that you sometimes want to create **branch**es, which are independently developed sequences of changes for some specific purpose. They are normally brought back together and merged into the trunk at a later time.

- The output lists `Cat.java` as a "change to be committed." This means that an updated version of `Cat.java` is *in your purple diamond.* (Refer back to Figure 14.1.) If we were to follow up this command with a "`git commit`", that version of `Cat.java` would be committed to the repo for posterity.

- Somewhat weirdly, the output shows `Cat.java` as being in the "changes *not* staged for commit" section as well! What the heck is going on here – is `Cat.java` going to be committed, or not?

  The answer to that question is "both." When a file shows up in both lists, it means there have been several *different* changes to it, only *some* of which were present the last time a "`git add`" was performed, and which are therefore in the purple diamond. Other changes to this same file were made *after* the "`git add`", and so they are in the yellow oval only. In a moment, I'll teach you how to find out which changes are in which category. For now, just grasp the fact that the same file can be in both sections of the "`git status`" output.

- The "untracked" files are the files in your workspace that `git` doesn't yet know about. Looking at this list is a good way to avoid making the "oops I did a `git commit -a` but forgot to add my new files" problem I mentioned earlier (p. 213). In this case, `Hairball.java` is potentially such a file, and this message reminds us that we may want to "`git add`" it. If we don't, our next commit will *only* have the (first set of) `Cat` changes, not the `Hairball` changes.

- The other entries under "untracked" are compiled `.class` files, not `.java` source files. **Normally, we *want* those to be untracked**, and so it's actually good that they're not set up as part of the commit. Some developers, however (myself included) find this message annoying. The way to fix it is to create a (hidden) file in your project directory called "`.gitignore`". *All files whose names match something in the "`.gitignore`" file will be ignored entirely by `git`*, and hence not be mentioned in a cries-wolf warning message.

  Here's an example `.gitignore` file, which you can edit like any other file in `vim`:

  ```
  .gitignore
  *.class
  ```

  There are two lines. The first is a bit of a mind-blower: it's the name of the `.gitignore` file itself! By including the line "`.gitignore`" in our `.gitignore`, we're telling `git` to not do version control on `.gitignore` itself. (Without that, we'll feel like we're stuck in a Monty Python skit where we create a `.gitignore` to avoid annoying warning messages, only for the `.gitignore` file itself to cause another annoying warning message.)

The second line says "also please ignore any file that ends with `.class`". Now, we know that anything that shows up in the "untracked files" section of `git status` really is something to think hard about.

- Finally, notice the helpful comments that "`git status`" provides: they're great for telling you exactly how to change things if necessary. For instance:
  - If we decide we don't want to check in that first set of `Cat.java` changes after all, we run the command "`git reset HEAD Cat.java`" to remove it from the purple diamond.
  - If we want *all* our changes in `Cat.java` to be committed (the old pre-`git-add` ones and the newer ones), we do "`git add Cat.java`" which will update the purple diamond's copy to match the workspace.
  - If we decide to actually ditch the `Cat.java` changes entirely, the command is "`git checkout -- Cat.java`" to discard them. (Notice the double-hyphen, and also the spaces on either side of it: it's a weird syntax but you have to use it.)

Understanding `git log`

While `git status` is a detailed look at the present, "`git log`" is a detailed look at the *past*. With it, we can see the time, author, and description of every change that's been made during our whole project.

I personally find the default `git log` output pretty wordy. It uses multiple lines per commit, which to me is TMI and fills up my screen too quickly:

```
$ git log
commit 928ab4924f1c5ddd3b9e2a1c7b507b1b60cf745d
Author: Betty Lou <bettylou@umw.edu>
Date:   2020-11-03

    Fix NPE bug caused by multiple hairballs.

commit 7813b199f40051df14b23a418bce37ccb51a986d
Author: Filbert <filbert@umw.edu>
Date:   2020-11-02

    Add support for multiple, simultaneous hairballs.

commit bdb0fa3071a220bfaccb0d687046e73873a6381d
Author: Betty Lou <bettylou@umw.edu>
Date:   2020-10-21

    Make most setters private.

commit e4471910b8d27c819e4e0df39804ab607cd16c5c
Author: Jezebel <jezebel@umw.edu>
Date:   2020-10-15

    Add Cat.java, Hairball.java.
```

Notice that the entries are in reverse-chronological order (most recent at the top), which is what you want to get used to. Each five-line section represents one commit. The big hairy numbers immediately after the word `commit` are called the commit's **hash**. That just means that every commit has a unique number, randomly/automatically generated by `git`, so that you can unambiguously refer to it. (More on why to do this in a moment.)

The `Author` and `Date` elements are self-explanatory, and the rest of the text is the actual message that the developer typed when doing the `git commit`.

It turns out that `git` provides a great deal of fine-grained control over what this output looks like. (Right away, that tells you that people look at git logs a *lot* and need them to be just the right format to quickly cull maximum information from them.) I like mine to be all on one line, and in color. To do this, I execute the line:

```
$ git config format.pretty "(%h) %Cblue%an%Creset: %Cgreen%s %Creset(%ad)"
```

which looks bizarre, but there's a method to its madness. Each one of these control characters (starting with a "%") specifies a certain piece of information or formatting. The result, when I type `git log` is this:

```
$ git log
(928ab49) Betty Lou: Fix NPE caused by multiple hairballs. (2020-11-03)
(7813b19) Filbert: Add support for simultaneous hairballs. (2020-11-02)
(bdb0fa3) Betty Lou: Make most setters private. (2020-10-21)
(e447191) Jezebel: Add Cat.java, Hairball.java. (2020-10-15)
```

Easier on the eyes, IMO. Run the command "`man git config`" to see all the options. Btw, you may be wondering what good it does to only list the first few characters of the commit hash. It turns out that for most commands that use it, you only have to type the first few characters anyway (just enough to guarantee uniqueness) and the short version is waaaay easier to look at.

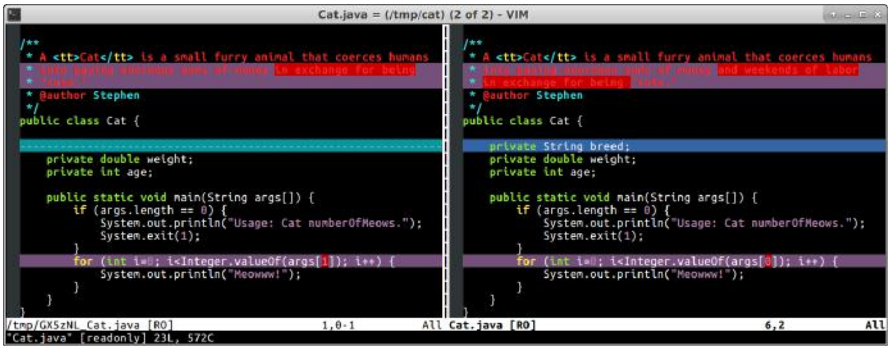
Comparing versions

A very common operation for a developer is to compare two versions (of the code base, or of a single file) to see what changed between them. Colloquially, comparing two versions of something is called "**doing a diff**," named after the Linux "`diff`" command. Here's my favorite way of comparing versions.

The `vimdiff` comparison tool

First, make these changes to your `git` profile[3]:

[3]By the way, any time you want to see *all* of your `git` configuration settings, you can do so with the command "`git config -e --global`". It will bring up all your settings in a text file for you to browse in `vim`. You can even edit this file directly to make changes to settings, although be very careful not to mistype anything. Stuff can go haywire if you do!



```
Cat.java = (/tmp/cat) (2 of 2) - VIM

/**                                                          /**
 * A <tt>Cat</tt> is a small furry animal that coerces humans  * A <tt>Cat</tt> is a small furry animal that coerces humans
 * into paying enormous sums of money in exchange for being   * into paying enormous sums of money and weekends of labor
 * "cute."                                                     * in exchange for being "cute."
 * @author Stephen                                             * @author Stephen
 */                                                            */
public class Cat {                                           public class Cat {

    ---------------------------------------------------------     private String breed;
    private double weight;                                       private double weight;
    private int age;                                             private int age;

    public static void main(String args[]) {                     public static void main(String args[]) {
        if (args.length == 0) {                                      if (args.length == 0) {
            System.out.println("Usage: Cat numberOfMeows.");             System.out.println("Usage: Cat numberOfMeows.");
            System.exit(1);                                              System.exit(1);
        }                                                            }
        for (int i=0; i<Integer.valueOf(args[1]); i++) {             for (int i=0; i<Integer.valueOf(args[0]); i++) {
            System.out.println("Meowww!");                               System.out.println("Meowww!");
        }                                                            }
    }                                                            }
}                                                            }
/tmp/GX5zNL_Cat.java [RO]             1,0-1          All  Cat.java [RO]                    6,2          All
"Cat.java" [readonly] 23L, 572C
```

Figure 14.2: Using "`git difftool`" when configured with "`vimdiff`".

```
$ git config --global merge.tool vimdiff
$ git config --global diff.tool vimdiff
$ git config --global difftool.prompt false
```

This tells `git` to use the "`vimdiff`" tool to do side-by-side comparisons. Then, "`git difftool`" is your principal way of bringing up a comparison. When you run it, you will get a somewhat strange-looking window that seems to be running *two* copies of `vim`: one on the left and one on the right. All your `vim` positioning commands – `h`, `j`, `k`, `l`, the arrow keys, `CTRL-U` and `CTRL-D`, even search with "`/`" – move the cursor around just as in normal `vim`. As you scroll up or down, both panes will scroll together. The changes from one version to another will appear in color so you can easily see what's changed between them. (Note that if there are long sequences of lines that were unchanged, sometimes `vimdiff` "folds them up" so that it's easier to skip over them.)

Figure 14.2 shows what `vimdiff` looks like when you run it. Its vertically-split pane shows two versions of the same file, with the differences between the two in various colors. It is plain from the figure that the differences between the two `Cat.java` versions are:

1. The JavaDoc class comment (see Chapter 18) has an extra phrase.
2. A new "`breed`" inst var has been added.
3. A bug has been fixed by changing the "`args[1]`" in the `for`

loop to "`args[0]`".

Super cool, and easy to see at a glance. Learn to use this often in your debugging and other code base investigations.

Don't attempt to use `vimdiff` to *edit* your file, though. If you're comparing two versions of a file and find a mistake, quit out of `vimdiff` and enter plain-ol' `vim` to fix it. (Incidentally, to quit `vimdiff` you have to enter "`:q`" *twice*, once for each of the panes. Sometimes it's quicker to use the "`:qa`" (for "quit all") sequence instead.)

Specifying the versions to compare

Okay. So that's how `vimdiff` (which is triggered from `git difftool`) works in general. Now, how do you specify what versions of the files you want to compare?

If you just type the command with no other arguments:

```
$ git difftool
```

you'll be comparing *your workspace* (yellow oval) against *your staging area* (purple diamond). This shows you the things that you have changed *since your last `git add`*. This is useful when asking, "I'm about to do a commit...have I forgotten anything I meant to include?"

If you add the "`--staged`" argument:

```
$ git difftool --staged
```

you'll be comparing *your staging area* (purple diamond) with the *repo* (green box). This is useful when asking, "if I do a commit right now, exactly what changes will be recorded?"

Finally, to compare *any two versions*, include the first few letters of the commit hash of each. For instance, this command:

```
$ git difftool bdb0fa3 7813b19
```

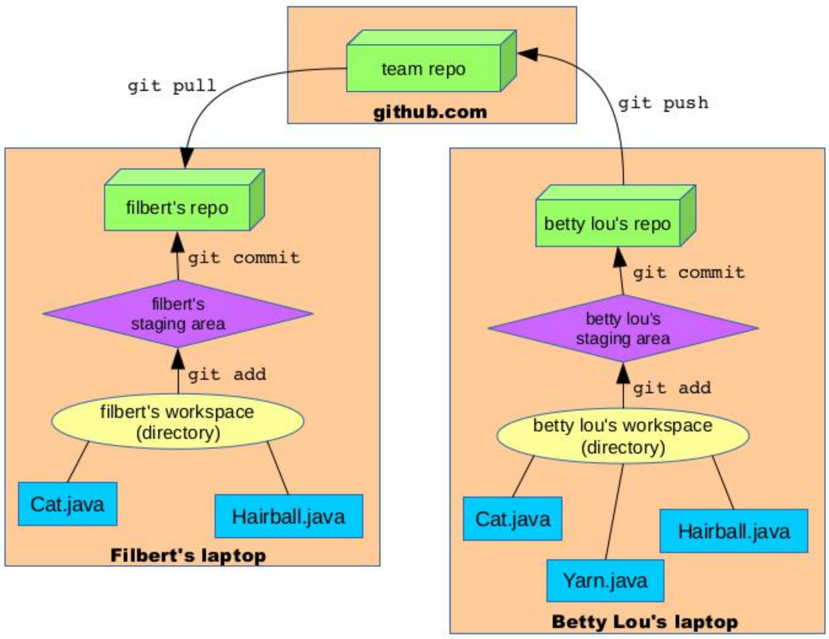


Figure 14.3: A development team's repos and workspaces.

will examine the changes Filbert made when he added multi-hairball support. (Refer to the commit hashes in the `git log` output, above.) The first hash identifies the commit Filbert was working with when he began making changes (*i.e.*, the one he started with), and the second is the hash of the commit he himself made (the one he ended with). This process gives us fine-grained resolution in examining the past and identifying errors.

Using `git` with more than one human

In a team environment, the `git` tool works much the same way, but with an added level of complexity to "join" the developers together. Look at the revised picture in Figure 14.3.

We now have two developers working on this feline program: Filbert and his colleague Betty Lou. The picture is considerably more complex. First, notice that the three large salmon-colored squares represent *different machines* which communicate only over the Internet. Filbert and Betty Lou each have their own laptop to do development on. And in addition, there is a third machine involved: a publicly-available hosting service like BitBucket, SourceForge, or github. Think of this public repo as the team's "home base": despite the fact that at any given moment Filbert and Betty Lou may be writing new code for the project, the latest stable version is always available in the repo.

There are also some new `git` verbs we need to learn about to make this picture work. To wit:

- `git clone`: Normally, the way this whole process kicks off is that someone creates the initial version of the repo on github, and the other team members **clone** copies of it. "Clone" is exactly what it sounds like: it means to make an exact duplicate.

  github makes it easy to do this in a variety of ways. Perhaps the most common is when one developer starts with an embryonic version of the code base, creates his/her own local git repo, and then "pushes" (see below) this to a new github project. It's also possible to start on github with a brand-new blank project. At the time of this writing, a very helpful and obvious green "New" button is present on the github main account screen, with instructions to follow for each of these different starting techniques.[4]

  On the github project page, you'll see a button that says "Clone or download," and which will display a specific URL to your project if you click on it. Then, on your local machine, you can go to the directory you want to be the parent of your project, and type something like:

  ```
  $ git clone https://github.com/stuff/projectName.git projectName
  ```

[4]Services like github and BitBucket are free to use, by the way, as long as you keep your repo public and visible to the world. The idea is that they're trying to promote open source software, so in exchange for sharing your ideas, they're giving you free storage space and tools. Some services even have free *private* repos – after being purchased by Microsoft, github now offers this feature for teams with three or fewer developers.

which will populate your filesystem with a copy of all the repo's files.

- `git push`: As usual, you'll constantly be making your own local commits to your own copy of the repo as you work. You should do this every time you reach a stopping point.

  A new operation in the team environment is the "`git push`". It says "take the updated contents of my own local repo (which have already been committed) and propagate them up to the team repo in github." This is how you share your changes with your teammates.

  Rule of thumb: making a local `commit` should be a common operation. Doing a `push`, on the other hand, is rarer: you only do it when your teammates need your latest code, and when your code is stable enough to warrant making it "the new normal" in the team repo.

  Last thing on `git push`: you normally don't do a `push` until *after* doing a `pull` to make sure you have your teammate's latest code integrated in yours. See next bullet.

- `git pull`: The inverse process of `push` is `pull`. It means, "go to github, fetch whatever changes have been made by my fellow developers, and integrate them into my repo so I have the latest and greatest."

  Now here's where `git` is fancy, and dare I say, seemingly magic. Suppose you've been editing code for the project at the same time your teammates are, and furthermore you're actually editing *the same files.* Doesn't that seem like it would be a nightmare? Doesn't it seem like you would each be making incompatible changes, and that one person's work is ultimately doomed to be wiped out by the other person's changes?

  That fear is indeed true if you're thinking of your code "a file at a time." But `git` is smart enough to consider your code "a *line* at a time." So if Filbert and Betty Lou are both making changes to `Hairball.java`, but they're working on *different parts* of that file, it turns out that git can automatically and intelligently **merge** the changes without you even having to worry about it.

When you do a `git pull` operation, *read the output carefully.* About 95% of the time, it will give you a happy message like this:

```
Auto-merging Hairball.java
Merge made by the 'recursive' strategy.
 Hairball.java | 1 +
 1 file changed, 1 insertion(+)
```

This is git's way of saying, "your teammate was editing the same file(s) as you, but it's chill; I figured out how to put their changes into your copy without messing up anything you were doing." When this first happened to me, I admit I was fearful, and couldn't comprehend how it could really be smart enough to integrate those changes without me checking. But I've since learned to stop worrying and love the bomb, and it really is "all good."

The other 5% of the time, you're not so lucky, and you'll get a sad message that says:

```
Auto-merging Hairball.java
CONFLICT (content): Merge conflict in Hairball.java
Automatic merge failed; fix conflicts and then commit the result.
```

This situation is called a **conflict** and essentially means that you and your teammate were working in the same *part* of the file and made incompatible changes. Perhaps one of you changed line 57 in one way, and the other of you changed line 57 in a different way. Or perhaps one of you changed line 90, while the other completely deleted lines 85-95. In these cases, git can't figure out what you want to do, so it throws it back to you and asks you to manually resolve it.

Resolving it is generally pretty easy. You open up the offending file(s) in `vim`, and look for the markers "<<<<<", "=====", and ">>>>>". Here's the kind of thing you'll see:

```
...
<<<<<<< HEAD
 * <tt>Cats</tt> are wonderful creatures that make good
 * house pets.
=======
 * A <tt>Cat</tt> is a small fuzzy animal that coerces
 * humans into paying enormous sums of money and weekends
 * of labor in exchange for being "cute."
>>>>>>> c6812dcd33b977de0e7f0e9cab1eb1376bcfda88
...
```

Here's how to decipher that. The lines between the "<<<<< HEAD" and the "=====" are what *you* had in your version of the file. In this example, you changed the `Cat` class JavaDoc entirely by rewriting it in a more feline-friendly way. Meanwhile, your teammate (whose code is marked between the "=====" and the ">>>>> `c6812...`") made a smaller change to that comment, changing "furry" to "fuzzy." All that matters now is what you want to do about these changes. It's your job as a developer to restore that part of the `Cat.java` file to be *what your team wants the code to look like moving forward.* Perhaps you keep your change, perhaps you keep your teammate's, perhaps it's a combination of both. But after fixing it up the way it should be (with maybe a phone call or chat session on Slack with your colleague to make sure you're on the same page), you will have removed the "<<<<<" and "=====" and ">>>>>" markers and can "`git add`" and commit your changes. Finally, you can then push the combined changes to the team repo, and everything goes on hunky dory.

Normally, the only time you end up in the 5% conflict case (instead of the 95% auto-merge case) is when you and your teammates aren't communicating well (or enough). Two developers both editing the same lines of code and not realizing that the other person is doing it is usually a sign that you need to work more closely together or be more explicit about who's working on what. A simple email or text often does the trick.

Bottom line: using the `git` verbs `add`, `commit`, `pull`, and `push` properly is your key to ensuring your team has a living, breathing, healthy, working repository of shared code.

Here are the most common pitfalls I see:

- Forgetting to do a `pull` before trying a `push`. git won't allow you to `push` into a repo if you're out-of-date. You must first `pull` the recent changes so you're all in sync, and *then* you can `push` your new changes to it.
- Forgetting to do a local `commit` before trying to `pull`. git won't let you pull changes from another repo if you have car parts all over your own garage. Make sure you check everything in and have a clean local workspace before doing that.[5]

[5]An alternative to a full local commit here is the "`git stash`" command which I've found useful. Doing a "`stash`" is different than a `commit`, since you're not actually marking version control with labeled changes. Instead, you're sort of sweeping stuff under the rug to make your workspace temporarily clean, for the sake of doing an important `pull` operation from your teammates. You can then pull your stuff back out from under your rug and continue. For details, type "`man git stash`" and read carefully.

Chapter 15

Doing design (1 of 2)

This chapter marks a watershed of sorts. Up to this point, we've been doing **analysis** instead of **synthesis**. Analysis is when you look at something that already exists – a design diagram or a code snippet, say – and seek to understand it, usually by breaking it down into its constituent parts. Synthesis, on the other hand, is designing something that doesn't already exist. Instead of scrutinizing a UML diagram, we're creating a UML diagram; instead of examining a method, we're writing a method.

Until now, I've presented you with example after example of classes and methods already written, and diagrams illustrating their various parts. But now it's time to ask the question: "how do we figure out what the right classes and methods *are* in the first place?" It's all well and good for someone to hand us `Ballplayer`, `Team`, and `Simulator` classes. But how did we know to create those particular classes? Why not `Pitch`, `Catch`, and `Hit`? Why not `FirstBaseman`, `Shortstop` and `Outfielder`, or `NationalLeague` and `AmericanLeague`?

Going from the general idea of a program to a list of classes is tricky. It's as much an art as a science. It calls for intuition and imagination more than adherence to a set of rules. Nevertheless, there are principles that guide the selection of good classes, and we'll talk about them in this chapter.

Of all the OO pioneers who weighed in on the question of how to arrive at a good design, the one who had the most influence on me was Rebecca Wirfs-Brock, who invented the technique called **responsibility-driven design**. I'm highly indebted to her, and recommend the original book authored by her and her colleagues.[1]

15.1 "Discovering the design"

I don't know who first coined the phrase "discovering the design" (it certainly wasn't me; it might have been Wirfs-Brock) but when I originally heard it my ears perked up. It sounded strangely paradoxical. "Design" was something one brought to the table and imposed on one's world, right? Not something one found already there. "Design" seemed like a matter of *invention*, not *discovery*; it was surely something you did to a steam engine, not to a planet.

Yet hidden in this phrase is a powerful technique for OO design that attempts to *let the requirements speak for themselves.* One of Rebecca Wirfs-Brock's great ideas was to begin with a written description of a software program in action, and to cull from the language clues as to what the "correct" classes are.

Let me immediately clarify that "correct" does not mean "there is one and only one 'right' set of classes" for a particular program. In fact, there are many such choices, some better than others, some downright awful. What we mean by "the correct classes" is a set of classes (and their corresponding inst vars and methods) that will:

- represent the domain well
- work seamlessly together
- be amenable to adaptation as the system requirements evolve
- distribute the responsibilities evenly among several classes
- neither duplicate nor omit important functionality

You get the picture. A good design is elegant, flexible, maintainable, and robust to change. Many choices of classes will not meet these goals. A few will. "The correct set of classes" means *any* set of classes that will do so reasonably well.

[1]Wirfs-Brock, Wilkerson, Wiener, *Designing Object-Oriented Software.* Prentice Hall, 1991.

Start with a written description of the software, and:

1. Identify all noun phrases.
2. Eliminate obviously bad ones:
    a) probable duplicates
    b) nouns that aren't instantiate-able
    c) things you obviously wouldn't represent
    d) likely *attributes* of a class, not classes themselves
3. The remaining ones are your **candidate classes**. See which of them "feel right." Identify what each one **knows** and can **do**.

Figure 15.1: Procedure for "discovering the design."

15.2 Straight from the horse's mouth

Wirfs-Brock's procedure is paraphrased in Figure 15.1. We have to somehow come up with a written description to kick things off. Often, a **requirements specification**[2] has been authored by someone higher-up on our company's food chain, and can be mined for much gold. Sometimes, we ourselves take a step back and bang out a few paragraphs that describe what users do and experience as they work with the system.

The essential point is that the requirements themselves speak loudly about what classes would be appropriate for the program it describes. Let's see how.

Nouns, and only nouns

If you flash back to *Schoolhouse Rock* or *Sesame Street*, you'll remember your grammatical parts of speech and realize that a **noun** is the right kind of word for a class name. Every object (and therefore the class it's an instance of) is a "person, place, or thing," not an action word, modifier, or anything else.

[2]Sometimes called a "req spec" – pronounced "reck speck."

Further, not just not any old noun will do. Consider the list in Figure 15.2: these are all nouns, but only *one* makes a valid class name. Can you find it?

~ happiness	~ crocodile	~ communism
~ Beyoncé	~ teamwork	~ recreation
~ oxygen	~ width	~ London

Figure 15.2: All nouns...but not all good class names.

I claim the only legit class name in this list is *crocodile*. Here's why. First, some of these entries are "proper nouns" which means they refer to specific instances of things, rather than categories. In English, we almost always use *capital letters* to denote proper nouns, which means when you see "Beyoncé" and "London" you can immediately roll your eyes and move on.

Second, most of the other nouns aren't *instantiate-able.* Here's the litmus test for whether a noun is instantiate-able: can you meaningfully put the word "a" (or "an") before it? And can you meaningfully make it plural and put a number (like thirteen) before it?

Clearly not. All of these phrases are plainly ridiculous:

- four happinesses (?)
- eleven oxygens[3] (?)
- a teamwork (?)
- a communism (?)
- nineteen recreations (?)

Remember, the only thing we ever really do to a class is make instances of it, to which we can do things. If you can't imagine a "`new Communism()`" or "an `ArrayList` of `Happinesses`," it has no business being a class.

[3]One could imagine a chemical analysis program that dealt with oxygen atoms, among other things, and I've heard chemists speak loosely of things like "an extra oxygen" or say "that molecule has five oxygens." I still like `OxygenAtom` much, much better as a class name even here, though.

The closest contender to crocodile is the word *width*. There may be cases where this is a sensible class, but the reason I discard it is that a width is almost certainly a *modifier* of some other object, rather than an object itself. One could imagine `Building`, `Image`, and `Rectangle` objects that all had an instance variable called `width`; it's harder to imagine "a width" as an entity in its own right, with its own properties and operations.

Noun phrases

By the way, it's often the case that instead of a bare noun, we use a **noun phrase** as a class name. A noun phrase is simply a noun with one or more modifiers. "Grizzly bear," "chess tournament," and "public liberal arts college" are examples.

Singular, not plural

Finally, it should hardly be worth stating that all class names must be **singular**, not plural. I don't work in "a buildings," but a *building*; and nobody has "a dogs" as a pet. When we instantiate an object, we're going to say "`Crocodile alice = new Crocodile()`", not "`Crocodiles alice = new Crocodiles()`".

15.3 Carrying out the process

1. Identify all noun phrases. Okay. We begin our semi-automated process of deriving class names by starting with a written description of the program's requirements. Here's a short example:

A bicycle store needs to manage its inventory. Shipments of various models of bicycles are received every week from its suppliers, and customers place individual orders for bikes and other accessories from the store. The store manager must be able to place orders from vendors, maintain contact information so they can be confirmed or canceled, and view lists of the incoming products and their expected arrival dates. The manager also must be able to record multi-item orders from individual customers, accept and record down payments, and track inventory levels to ensure that enough items are ordered to satisfy customer demand.

Rebecca Wirfs-Brock's process from Figure 15.1 calls for sifting through the requirements description and *circling all noun phrases*. Unless it's an exact duplicate of one that previously occurred, be conservative and circle every one. It would be a good exercise for you to do this yourself in the box above, and then compare with my answer:

A bicycle store needs to manage its inventory. Shipments of various models of bicycles are received every week from its suppliers, and customers place individual orders for bikes and other accessories from the store. The store manager must be able to place orders from vendors, maintain contact information so they can be confirmed or canceled, and view lists of the incoming products and their expected arrival dates. The manager also must be able to record multi-item orders from individual customers, accept and record down payments, and track inventory levels to ensure that enough items are ordered to satisfy customer demand.

This is the raw material for the rest of the process. If we make everything singular and lower-case, this leaves us with the following list:

bicycle store	bike	manager
inventory	accessory	multi-item order
shipment	store	individual customer
model	store manager	down payment
bicycle	order	inventory level
week	vendor	item
supplier	contact information	customer demand
customer	list	
individual order	incoming product	
	expected arrival date	

2a. Eliminate probable duplicates. According to Figure 15.1, the next step is to eliminate likely duplicates. Obviously things like "bicycle" and "bike" refer to the same conceptual entity; we're hardly going to have a `Bicycle` class and a separate `Bike` class in our program!

This isn't always 100% straightforward, but it's usually 99% so. Different synonyms and turns of phrase are pretty easy to detect. I think we can be pretty safe boiling this list down into a slightly smaller one, where duplicates are shown:

bicycle store (== store)	accessory
inventory	store manager (== manager)
shipment	contact information
model	list
bicycle (== bike)	incoming product
week	expected arrival date
supplier (== vendor)	down payment
customer (== individual customer)	inventory level
individual order (== order	item
== multi-item order)	customer demand

The choice of which synonym to retain is mostly aesthetic. All other things being equal, I usually choose the shorter one.

2b. Eliminate nouns that aren't instantiate-able. Now we apply our test: "can we put 'a/an' or a number before the noun phrase, and have it make sense?"

Actually almost all of these remaining entries pass that test, with the exception of customer demand, and possibly contact information. While one could indeed envision "four or five customer demands" in other contexts, it's pretty clear from the text that this is being used as an abstract concept, not an individual object. "Contact information" is a closer call, but by inspecting the requirements again, we can see that this is really an attribute of vendor/supplier. We'll therefore strike the idea of a "`ContactInformation`" class. We're now down to:

store	supplier	incoming product
inventory	customer	expected arrival date
shipment	order	down payment
model	accessory	inventory level
bicycle	manager	item
week	list	

2c. Eliminate things you obviously wouldn't represent. When you look at some of these surviving noun phrases, you scratch your head. Would we really have a "`Week`" class? Surely not. Also, although this program will no doubt be used by the manager of a store, does it really make sense to represent the `Manager` as an object? We'll cross out both of these.

2d. Eliminate likely *attributes* of a class. Things are getting a bit more subjective, but some of these remaining nouns definitely seem "too small" to be their own classes. Consider expected arrival date. Surely this is better modeled as a property of an order, rather than as its own individual object. The same could be said for down payment and inventory level. Generally speaking, noun phrases that seem to refer to bits of data that have an obvious "home" in another class ought to be modeled as inst vars, not classes.

So now here's all that remains:

- store
- inventory
- shipment
- model
- bicycle
- supplier
- customer
- order
- accessory
- list
- incoming product
- item

We dub these our **candidate classes**, which essentially means "those noun phrases which each have a very good chance of actually turning into a class in our program." We're not 100% committed to them yet, but they pass muster enough to deserve a deep think.

3a. See which of the remaining ones "feel right." We've come a long way semi-mechanically. Now it's time to allow ourselves the luxury of turning over in our minds each candidate class, "trying it on," so to speak.

It's here that a clear picture of our software system emerges. When I look at the ten candidate classes, here's what comes to mind:

- First, and most importantly, I realize that the word "order" was used in *two different ways* in the requirements description. You may have actually noticed this earlier when we scratched out "expected arrival date" and "down payment" in step 2d. Those were both aspects of an order...but what *sort* of order? If we go back to the Bible (the req spec) we see these two phrases:

  "...customers place individual orders for bikes and other accessories from the store..."

  and

  "...The store manager must be able to place orders from vendors..."

Aha! Different beasts entirely. One is something Mrs. Jamison places with *us*; the other is something *we* place with Schwinn, Inc.

This sort of post-noun-phrase-stage epiphany isn't uncommon. English words are used in a variety of ways, which makes them versatile and suggestive, yet imprecise. Here, we clearly have two different notions of "order": (1) a contract for delivery from one of our big suppliers like Trek or Cannondale (which might include a dozen bikes or more), and (2) a customer's reservation of a particular model/color/style of bike, which he or she is anxiously waiting to take home for its first ride. More succinctly: one kind we buy, and the other kind we sell.

We're going to have to invent at least one noun phrase of our own here, since the req spec author double-dipped on the word "order"; perhaps we'll call the first one a `PurchaseOrder` and the second one a `CustomerRequest`. (In situations like this, I think it's better to *avoid* using the original word altogether, since it was ambiguous to begin with and therefore may encourage confusion down the road.)

- Second, I notice that some of these nouns have overlapping meanings, and I sense that *inheritance* might be applicable. In particular, consider these four noun phrases:

  bicycle   accessory   incoming product   item

  These clearly all refer to things that can be purchased. When we go back to the Bible, we see that the modifier "incoming" on product really refers to the temporary state of a product (*i.e.*, one in transit from a supplier), not a fundamentally distinct kind of thing. So I'm going to be bold, ditch "incoming," and make these two assertions:

1. item == product
2. bicycle and accessory are two different kinds (subclasses) of item

With these changes, our list is now:

- store
- inventory
- shipment
- model
- bicycle (subclass of item)
- supplier
- customer
- purchase order
- customer request
- accessory (subclass of item)
- list
- item

3b. Identify what each one *knows* and can *do*.

As you'll recall from Chapter 2, a well-conceived class combines both **state** and related **behavior**. This is the cornerstone of good object-oriented design.

Hence at this stage, we apply this litmus test to the candidate classes that remain. One tool to facilitate this procedure is creating a set of **CRC cards**, which stands for "Class, Responsibilities, Collaborations." The idea is for your design team to create a literal 3×5 card for each candidate class, put them on a flat workspace so you can move them around and compare and group them, and see whether they seem to fit into a cohesive whole.

The **Class** for each card is just the class name (which can be fluid as you try to zero in on the best name). The **Responsibilities** include the state ("what an object of that type knows") and the behavior ("what an object can do"), which I usually designate in separate sections. Finally, the **Collaborations** are the other classes the class is likely to work closely with to accomplish its job.

If you try to make a CRC card for a class, and have trouble coming up with appropriate contents (particularly the Responsibilities section), that's a red flag that perhaps this isn't a good class after all.

Let's start at the top.

`Shipment`:	
Knows:	which vendor is delivering it
	which purchase order it is fulfilling
	the expected arrival date
	the items in each shipment
Can:	receive and add to inventory
	cancel
	update status
Collaborates:	`PurchaseOrder`, `Item`, `Inventory`, `Supplier`

Our `Shipment` class looks like a good one. It clearly bears the hallmarks of good OO design: it has state (information about what's in the shipment, when it's due, and who it's from) and associated behavior (accept it, cancel it, get an updated status).

You might be thinking, "that's great, Stephen, but how did you know what to write in that box?" I admit it's not a turn-the-crank process. This is part of the design process that's art, not science. Basically, though, you have to try to create a little egocentric world in your mind, the center of which is the class in question.

Here, I said to myself, "let's view the entire world through the lens of a Shipment. First, in the real world, what ought a shipment to '*know*'?" The answer is things that relate directly to that shipment. The items it should contain and the expected date of receipt are good examples. The number of bicycles in the store is not, nor is a customer's contact information. Then, I asked "in the real world, what *actions* pertain to a shipment?" The main one, of course, is to receive that shipment, which involves paying for it and adding the items to the store. We also might ask the supplier for an update if the shipment is past date, or even decide to cancel it and go with another vendor if it's taking too long.

The important thing here isn't to get all the "knows" and "cans" 100% right – things will evolve as our understanding evolves. It's more important to recognize that there *are* clear "knows" and "cans" for this class, which certifies it as a bona fide entity within our object-oriented system.

The "Collaborates:" list consists of those other classes we've identified as likely co-participants in various system functions. A `Shipment` is related to a `PurchaseOrder`, of course, since the former is the fulfillment of the latter. Its also comprised of `Items`, and will need to update the `Inventory` levels when it arrives. It may need to call methods directly on the `Supplier` class in the case where updates or cancellations are necessary. Often the collaborations list just helps us organize our thoughts (and our 3×5 cards) by grouping related classes together.

Let's walk through a couple of other CRC cards.

`Accessory`:	
Knows:	its name
	its manufacturer
	its part #
	its cost
	its compatible bicycle models
Can:	purchase
	find quantity in stock
	add to customer request
Collaborates:	`CustomerRequest`, `Model`, `Inventory`

An `Accessory` – which we've tentatively identified as a subclass of `Item` – has attributes like name and cost, and also knows which bike models (if any) it is compatible with. This prevents a customer from ordering the wrong kind of bike seat or fender for a particular bicycle, for instance. It can also be purchased (duh), either on the spot or as part of a special customer order. It can also provide its quantity information by interfacing with the `Inventory` class. Speaking of which...

`Inventory`:	
Knows:	the quantity of each item in the store
Can:	add items when shipments arrive
	remove items when purchases are made
	find quantity in stock
Collaborates:	`Item` (and subclasses)

I smell a Singleton. Our store is likely going to have a single `Inventory` object which can be used to query and update its item quantities.

By the way, you may have noticed that one of the "Can" items – "find quantity in stock" – was listed on the CRC card for both `Accessory` and `Inventory`. This isn't wrong; probably an `Accessory` object, when asked for its current quantity, will turn around and query the `Inventory` singleton to produce that answer. We've discovered a key shared function between classes.

That's the inventory of the store, and now for the `Store` itself:

`Store`:	
Knows:	its inventory
Can:	?
Collaborates:	`Inventory`

If we didn't realize it before, this is the moment when we discover that our `Store` class is weak sauce. Turns out there really isn't anything for a `Store` object to know or do, other than manage its inventory, which of course is the job of the `Inventory` class. The CRC card process revealed a false infiltrator, and we discard (literally) the `Store`.

You get the idea. I'll leave the other CRC cards as an exercise for the reader. Remember, there are no hard-and-fast right answers to this process: many different designs are possible, and it's only important to get a set of classes that are cohesive, modular, encapsulated, and work well together.

15.4 A longer example

Figure 15.3 on p. 246 gives a second, longer example of a req spec. We'll develop this example in the remainder of this chapter and in the next one. In particular, it involves a non-trivial inheritance hierarchy.

First, test yourself on identifying noun phrases, and see if you find the same ones marked in Figure 15.4.

Our mechanical noun phrase extraction produces this list:

- intro student
- skill
- professor
- Uno simulation program
- simulator
- Uno game
- player
- student
- `Player` class
- card
- color
- wild
- game
- deck
- type
- number
- "wild" card
- time
- hand
- virtual table
- "up" card
- chance
- discarded card
- object
- rule
- rank
- special effect
- direction of play
- turn
- unfortunate player
- round
- point
- "forfeit cost"
- point value
- cumulative score
- winner

Note that this req spec somewhat unusually refers to a specific object-oriented *class* (`Player`) which will of course become one of our actual classes in the end.

2a. Eliminate probable duplicates. After eliminating likely duplicates, our list is shrunk to:

- student
- skill
- professor
- simulator
- game
- player
- card
- color
- "wild" card
- deck
- type
- number
- time
- hand
- virtual table
- "up" card
- chance
- object
- rule
- rank
- special effect
- direction of play
- turn
- point
- "forfeit cost"
- cumulative score

An interesting decision here involved the terms game and round. The former is used in a couple of different senses: Uno itself is a "game," yet the word "game" is also used to mean a single deal of the cards, at the end of which one player goes out. Curiously, there's no noun in the description for "the overall match" which comprises 50,000 games. We may find we need such a class. In any event, I scratched round in favor of game in the list above.

2b. Eliminate nouns that aren't instantiate-able. After getting rid of the non-instantiate-able stuff, we shrink further to:

- student
- professor
- simulator
- game
- player
- card
- color
- "wild" card
- deck
- type
- number
- hand
- virtual table
- "up" card
- rank
- special effect
- direction of play
- turn
- point
- "forfeit cost"
- cumulative score

It's worth drawing attention here to the noun "rule," which I discarded. I find that many students' inclination is to keep rule as a class, whereas I think the description makes it clear that "rules in general, according to which the game is played" is what's intended. And that would steer us away from instantiating some number of `Rule` objects.

2c. Eliminate things you obviously wouldn't represent. The only ones I got rid of on this step were (ironically) student and professor. Nothing personal.

2d. Eliminate likely *attributes* of a class. I think you'll agree that color, type, number, and rank are best suited as attributes of a `Card` class, not as classes in their own right. Too, direction of play – which is simply "clockwise" or "counter-clockwise" – seems like a property of the game. Similar thinking leads to deleting point and cumulative score (attributes of the `Players`) and forfeit cost (an attribute of a `Card`.) We're now left with only:

- simulator
- game
- player
- card
- "wild" card
- deck
- hand
- virtual table
- "up" card
- special effect
- turn

3a. See which of the remaining ones "feel right." This is honestly a pretty darn good list. If I were to nitpick it further, I'd probably say that "up" card will probably turn out to be an instance variable of *type* `Card`, rather than its own class. I'd wager that turn doesn't end up being a full-blown class either, since "whose turn it is" is better served with an inst var.

The "wild" card noun phrase is quite literally going to become a wild card for us, as we'll discover in the next chapter. It conceals what is really going to be a deep inheritance hierarchy, in which the different types of cards are all subclasses of `Card`. This is where the special effects come into play as well – in the end, I didn't model this as its own class, but rather embedded the functionality into the `Card` subclasses. Either way's okay, though.

3b. Identify what each one *knows* and can *do*. I'll sign off this chapter by taking a crack at CRC cards for some of the classes that are going to survive the whole design vetting. These are quality classes that will ensure a solid design that's robust for the present and the future!

`Simulation`:	
Knows:	the name of each player
	the `Player` subclass for each player
	how many total games to simulate
	how many games have been played so far
Can:	play some number of games and compute final scores
Collaborates:	`Game`, `Player` (and subclasses)

`Card`:	
Knows:	its type
	its color (if applicable)
	its number (if applicable)
	its “forfeit cost”
	whether it can be legally played on an up card
	whether the player who played it can call a new color
Can:	perform any special effect appropriate to its type
Collaborates:	`Hand`, `Deck`

`Deck`:	
Knows:	the contents of a standard Uno deck
	which face-down cards it contains, in order
	which face-up cards have been discarded
Can:	draw a new card
	reshuffle (when empty)
Collaborates:	`Card` (and subclasses)

`Game`:	
Knows:	which player’s turn it is
	the current direction of play
Can:	start a simulation of a single Uno game
	advance to the next player
	reverse the direction
	observe the end of the game, and report scores
Collaborates:	`Player` (and subclasses), `Hand`, `Deck`

`Hand`:	
Knows:	which cards are being held
Can:	defer to its controlling `Player` object to choose a card
	defer to its controlling `Player` object to call a color
	add a card (when the player must draw)
	count the “forfeit costs” of all its cards
Collaborates:	`Player` (and subclasses), `Card`

`Player`:	
Knows:	its hand
	all cards that have been played since last shuffle
Can:	select which card to play on the "up card"
	choose a color to call immediately after playing a wild
Collaborates:	`Card`, `Game`

To test his intro student's algorithmic development skills, a professor is developing an Uno!® simulation program. The simulator simulates a number of consecutive Uno games, each of which has four players participating in it. Students write their own Player classes, which are called by the simulator in order to play cards and call colors after wilds are played.

Uno is a game played with a special deck of cards of various types. Most cards have a color (red, blue, yellow, or green) and feature either a number on them (from 0 to 9) or else a special action (like "reverse," "skip," *etc.*) Some cards are "wild" cards, which do not have any particular color, and thus can be played at any time.

When play begins, the deck is shuffled, cards are dealt to each player's hand, and one card is turned face up in the middle of the virtual table, called the "up card." Each player in turn gets a chance to play, by playing a card from their hand on top of the up card. That up card is then replaced by the new up card. If the deck is ever exhausted (*i.e.*, runs out of cards) the discarded cards are reshuffled and placed beside the up card to be drawn anew. The object of the game is to be the first player to "go out" by playing all cards from your hand.

In order to be legally played, a card must match according to certain rules (either the color of the card played, or the rank of the card played, must match the up card.) Some cards have special effects, involving reversing the direction of play, skipping over player's turns, or causing unfortunate players to have to draw additional cards from the deck.

When one player wins a round, he/she gets awarded points based on the cards remaining in other players' hands. Each type of card has "forfeit cost," or point value that determines how much it is worth. As it runs, the simulator maintains the cumulative scores of the players as they each win games, so that at the end of 50,000 games, an overall winner can be declared.

Figure 15.3: The requirements specification for a game simulator.

To test his intro student's algorithmic development skills, a professor is developing an Uno simulation program. The simulator simulates a number of consecutive Uno games, each of which has four players participating in it. Students write their own Player classes, which are called by the simulator in order to play cards and call colors after wilds are played.

Uno is a game played with a special deck of cards of various types. Most cards have a color (red, blue, yellow, or green) and feature either a number on them (from 0 to 9) or else a special action (like "reverse," "skip," *etc.*) Some cards are "wild" cards, which do not have any particular color, and thus can be played at any time.

When play begins, the deck is shuffled, cards are dealt to each player's hand, and one card is turned face up in the middle of the virtual table, called the "up card." Each player in turn gets a chance to play, by playing a card from their hand on top of the up card. That up card is then replaced by the new up card. If the deck is ever exhausted (*i.e.*, runs out of cards) the discarded cards are reshuffled and placed beside the up card to be drawn anew. The object of the game is to be the first player to "go out" by playing all cards from your hand.

In order to be legally played, a card must match according to certain rules (either the color of the card played, or the rank of the card played, must match the up card.) Some cards have special effects, involving reversing the direction of play, skipping over player's turns, or causing unfortunate players to have to draw additional cards from the deck.

When one player wins a round, he/she gets awarded points based on the cards remaining in other players' hands. Each type of card has "forfeit cost," or point value that determines how much it is worth. As it runs, the simulator maintains the cumulative scores of the players as they each win games, so that at the end of 50,000 games, an overall winner can be declared.

Figure 15.4: Noun phrases.

Chapter 16

Doing design (2 of 2)

16.1 The two domains

Before we dive back in and complete our two examples from last chapter, let me make an observation about the classes in an OO program. They tend to come from two different sources. We call these two categories “the **problem domain**” and “the **solution domain**.”

The problem domain provides classes that relate to the *problem* the program is designed to solve. A key give-away of a problem domain class is that the user herself recognizes the term used. She thinks of that entity as central in what the system does/is.

For instance, in an eBay type application, classes like `Bid`, `Auction`, `Item`, `Seller`, and `Buyer` are all from the problem domain. eBay users think about, and talk about, these very concepts when they think about the system, even if “code” never crosses their mind.

The other source of classes is the solution domain, which consists of supportive classes that don’t really represent things about the problem itself, but which are necessary to solve the problem. Suppose an email application had a `SMTPServer` class. This would represent a connection to a piece of hardware that acted as a SMTP (Simple Mail Transfer Protocol) server to deliver electronic mail. Is this class’s functionality necessary to send email, as the email application needs to do? Yes. But does an everyday email user think about "SMTP Servers" being involved? Likely not. The same could be said of classes like "`DatabaseConnection`," "`MessageListener`," and "`LoginPane`." These classes all perform critical supporting functions and therefore are vital to the operation of the system. At the same time, though, we recognize that they are tangential to the main purpose of the system as the user sees it: users of Wikipedia don't think in terms of "database connections," nor email users of "message listeners," nor Spotify users of the "login panes" in their UI. So we relegate those classes to a different realm of sorts; one that contains classes to perform functions, not to represent the domain's reality.

You might wonder which of the two domains is most important to get right. The answer is unquestionably the *problem domain.* Think about it: if Spotify decided to change their underlying storage mechanism, and thus needed to retire their `DatabaseConnection` class, that's not a big deal to their user base. If the new program version is implemented well and doesn't introduce a lot of lagginess or bugs, the user will be unaware that it was even changed. But change something in the *problem* domain, and whoo Nellie, the whole system experiences a change. Imagine if Spotify got rid of their `Song` or `Playlist` classes. The entire application would have to perform differently, with serious consequences for the user.

16.2 Turning CRC cards into UML

When we last left our heroes in chapter 15, they had succeeded in turning an English language description into a set of CRC cards. That's a ton of progress. All they need to do now is complete the trick: turn those CRC cards into a UML class diagram, and then into Java code. And that's just what we'll do in the rest of this chapter.

The bike store example, continued

Reacquaint yourself with the CRC cards on pp. 238–239. These reflect some of the candidate classes from our bike store example.

You've probably already figured out that when turning a CRC card into a class, the "Knows:" section typically gets turned into instance variables, and the "Can:" section becomes methods. It isn't always a straightforward one-to-one mapping, but it's often pretty close.

Let's start with `Shipment` on p. 238. The four items on its "knows" items all call out for instance variables:

- "which vendor is delivering it": type `Supplier`
- "which purchase order it is fulfilling": type `ArrayList<PurchaseOrder>`
- "the expected arrival date": type `Date`[1]
- "the items in each shipment": type `ArrayList<Item>`

In terms of a UML diagram, we would depict the third of these as an entry in the middle class box (see Figure 16.1), and the other three as associations to other classes. Also, the "can" list mentions that we can update the "status" of a `Shipment`, which will probably entail a `String status` inst var as well.

As for its methods, we have getters and setters for status and supplier, and also the ability to `.cancel()` and `.receive()` the shipment. At this point we're sort of guessing as to argument types and return values for each method; it seems to me that both `.cancel()` and `.receive()` can simply be argument-less and return `void`. (We'll amend this assumption later if it turns out to be incorrect.) The finished class is in the upper-left corner of Figure 16.1.

We didn't actually write full CRC cards for all the classes in this design, but that's okay: to complete `Shipment`, we can just sketch in temporary placeholders for classes like `PurchaseOrder` and `Supplier`.

The `Accessory` card from p. 239 has a number of "knows" entries, though when we consider where to put them, we realize that many of them will go in the abstract `Item` class. Only an `ArrayList` of `Bicycle` objects seems appropriate as an inst var for the `Accessory` subclass specifically, and that is what the diagram shows. Since p. 239 tells us that an `Accessory` "knows its compatible bicycle

[1]The `java.util` package has a `Date` class that represents all the necessary aspects of a day in time on planet Earth. This is a better choice than a `String` or a handful of `int`s to do it ourselves.

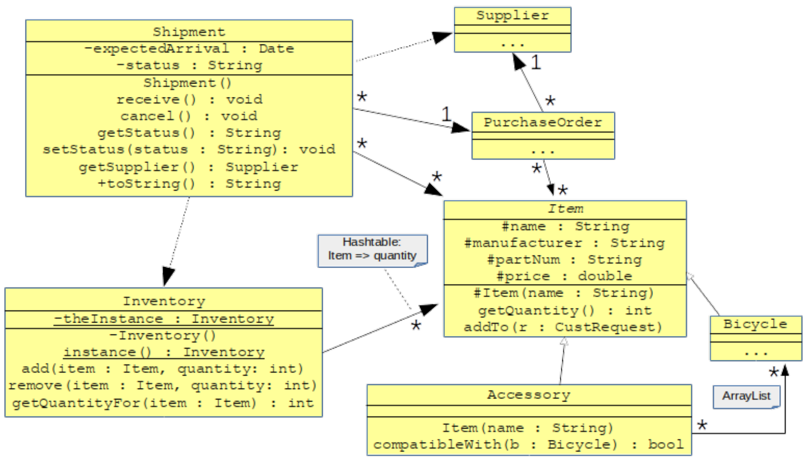


Figure 16.1: A first crack at converting CRC cards from Chapter 15's bicycle example into UML.

models," it seems appropriate for the class to support a method like `.compatibleWith()` that returns a `boolean` indicating whether the accessory in question is compatible with a particular bike.

Finally, our `Inventory` CRC card (also on p. 239) tells us that in addition to the standard Singleton stuff, we need to be able to `.add()` and `.remove()` quantities of items from the `Inventory`, as well as get a current count of how many units of an `Item` we have in stock. One way to implement this would be through a `Hashtable` that maps each `Item` to an in-stock quantity, and that is what Figure 16.1 shows.

I think you'll agree this is a pretty straightforward, though not completely mechanical, process. CRC cards have already identified the lion's share of the program's important static structure, and go a long way towards giving us a UML class diagram from which we can write code.

The Uno!® game example, continued

Now let's work on the Uno!® example from the CRC cards on pp. 243–244. We'll break this up into two UML diagrams, one for the principal classes (Figure 16.2), and the other for the `Card` inheritance hierarchy (Figure 16.4).

The CRC cards didn't explicitly say that `Simulation` would be our `main()` class, but it's as good a choice as any, so that's what's reflected in the diagram (bottom-left). All the "knows" have been given inst vars. The `Simulation` will instantiate lots of `Game` objects: one for each of the 50,000 games in the match, to be precise. Each `Game`'s `.play()` method will simulate a single game to completion, and return an array of the scores to add to each player's cumulative total (held in the `Simulation` class). Btw, we could have made `Simulation` a Singleton, and given it non-static inst vars. Your call.

The "double dispatch" technique

The `Game` CRC card (middle of p.244) tells us that it must maintain the current player and the direction of play at all times. These two bits of information are represented as inst vars in the second compartment of the class. `Game` also has methods on it to `.advanceToNextPlayer()` and to `.reverseDirection()`. These can be called by any other part of the program in order to modify the game's state. Our plan is for different `Card` subclasses to invoke these methods to carry out their special effects: see the `.performCardEffect()` method on the abstract `Card` class in the upper-left corner.

This technique is referred to as **double dispatch**, and it can be disorienting at first. In double dispatch, you call a method on object A, passing it another object B as an argument. Object A's method then, in addition to whatever else it might need to do, will call method(s) on B.

Figure 16.3 shows this in action for the Uno game. In this scenario, the `Game` object determines that the next player is #3, and therefore instructs the third `Hand` to `.play()` a card. **Note that `g` passes `h3` the argument "`this`" in the call to `.play()`.** That's how double dispatch works: `h3`'s `.play()` method now has possession of the `Game` object `g`, which it can call methods on and/or pass around further. In this case it does both: first, `h3` turns around and calls

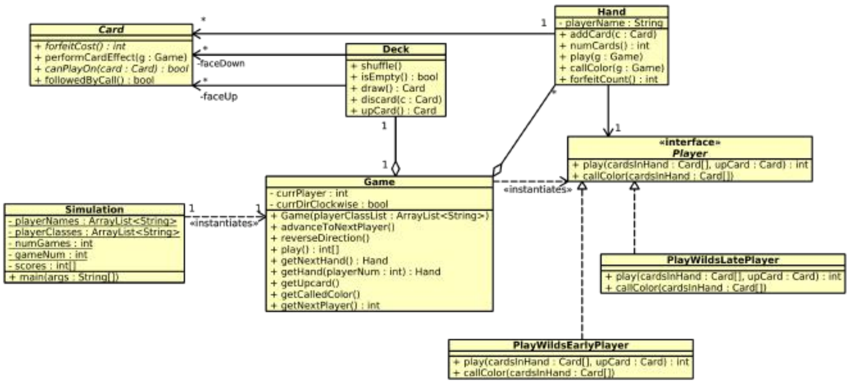


Figure 16.2: A first cut at converting CRC cards from Chapter 15's Uno example into UML.

`.getUpCard()` on `g`, to find out what the up card is. Then, when it passes that `up` card along to `p3`'s `.play()` method, and learns that the `Player` algorithm chooses to play card #7 from her hand (a blue reverse), it calls `.performCardEffect()` on that `ReverseCard`. The `Card` object `c` now also has possession of the `Game` object, and can tell it to do the two things required: reverse the direction of play, and then advance to the next player's turn. Other types of cards would do different things instead, as described in the next section.

Back to Figure 16.2. We see that the `Deck` class – whose CRC card was given on p. 244 – contains `two` collections of `Card` objects, one to hold a sequence of face-down cards and one for the face-up cards. The rest of this class is self-explanatory.

`Hand` objects each hold on to a list of `Card`s, of course, as well as the corresponding player's name for good measure (which wasn't on the CRC card). It also maintains an inst var to a "`Player`": this is the creation by each of Stephen's students that implements the `Player` interface and thus provides an algorithm for choosing cards and colors. Two example players have been shown: one that plays wild cards as soon as it can, and another that holds them until forced to. (Surprisingly, to me anyway, the former outperformed the latter in most simulated games.)

The `Card` class hierarchy

Finally, let's figure out the `Card` class and its subclasses. It's a bit tricky. One might think that a `Card` having a "`number`" is a no-brainer...except that not all cards *have* numbers (like Skips or Draw 2's.) Very well, then, at least all `Card`s have a *color*, you say...except that wild cards don't have that either.

The trick is to recognize when there are commonalities between card types, and to infer the presence of appropriate *abstract* classes. Figure 16.4 gives the idea. All of the associations here are top-down inheritance ("is-a"). In addition to all the concrete `Card` types you'd expect – `DrawTwoCard`, `ReverseCard`, `WildCard`, *etc.* – we have created abstract `ColoredCard` and `ActionCard` classes.

But isn't this overkill, you might ask? It is not, for the following reason: each piece of information is now in only one place. For example, all numbered cards *and* "action cards" have a color, but wilds (of either variety) do not. Therefore, it makes sense to define the `color` inst var in the superclass of all the `Card` types that have colors. It shouldn't be in the `Card` class itself – that's too general, since not all `Cards` have colors. And it shouldn't be in the `NumberedCard` class – that's too specific, since more than just numbered cards have colors. By similar logic, only the `NumberedCard` class should have an `int` inst var.

Furthermore, since all "action cards" have the same forfeit cost (20 points) it is appropriate to define a (non-abstract) `.forfeitCost()` method in the (abstract) `ActionCard` class. That way, `SkipCard`, `ReverseCard`, and `DrawTwoCard` don't need to override it.

Note that `WildCard` is a concrete class, even though it has a subclass. This is perfectly okay (in Java), and necessary since there are indeed ordinary wild cards in the deck. Both types of wilds have the same forfeit cost (50 points) and both require the player to call a color, but the Draw 4 variety obviously has a different effect on the game, and therefore provides its own `.performCardEffect()` method that overrides that of the `WildCard` superclass (and the `Card` superclass).

16.3 Evaluating a design

I end this chapter by giving a few simple guidelines for sanity-checking your design once you've gotten this far. Again, there is no one "right way" to design a program, but there are plenty of wrong ways, and some of them are easy to spot.

Here's my super short "must-do" checklist:

1. Each class represents a **crisp** and **coherent** entity.
2. Each class does **one thing** well.
3. Responsibilities are **distributed** over the entire design.
4. There is evidence of **encapsulation**.

The first item on the list is somewhat intangible, but oh-so-important. It basically means that the meaning and purpose of each class should be natural and easy to describe. If you find yourself struggling to articulate what specific type of entity one of your classes actually represents, rethink it.

The second item involves a very common pitfall for beginning designers: having *too few classes, each of which does too much.*[2] There's a funny name for a class that does too much: it's called a "**god class**" (no joke.) Very often, I see students creating designs that on the surface seem to have several collaborating classes, but which in actuality have all the real functionality in one god class while the others serve merely as data containers.

Strive instead to have each class do only one self-contained job, and to do it well. Remember: the larger your classes are, and the more tasks each one encompasses, the less encapsulated your program is bound to be.

Related to this is the third item, which is that when your program is in operation, most of its important responsibilities should be *shared* between the different classes. This is exposed clearly on sequence diagrams: if you find that most of your arrows are emanating from a single vertical line, that's a bad sign. If you look at the Uno design, you'll see that any significant operation – like a player actually

[2]I have a theory that this is normally due to simple laziness in creating new files, but I've never seen that proven.

taking her turn – involves many classes operating in tandem: `Game`, `Deck`, `Hand`, a `Player` implementation, and some subclass of `Card`. This is A Good Thing.™

Finally, for each class, you want to scrutinize its list of inst vars (both those in the second compartment and those implied by associations) and its methods and make sure they all "fit together" well. They should all make sense for that type, and the data and behavior should go hand in hand. Each class's design decisions should be cleanly insulated from others. Ideally, when you look at your design, you should see a picture like the right-hand side of Figure 2.2 (p. 26), not the left.

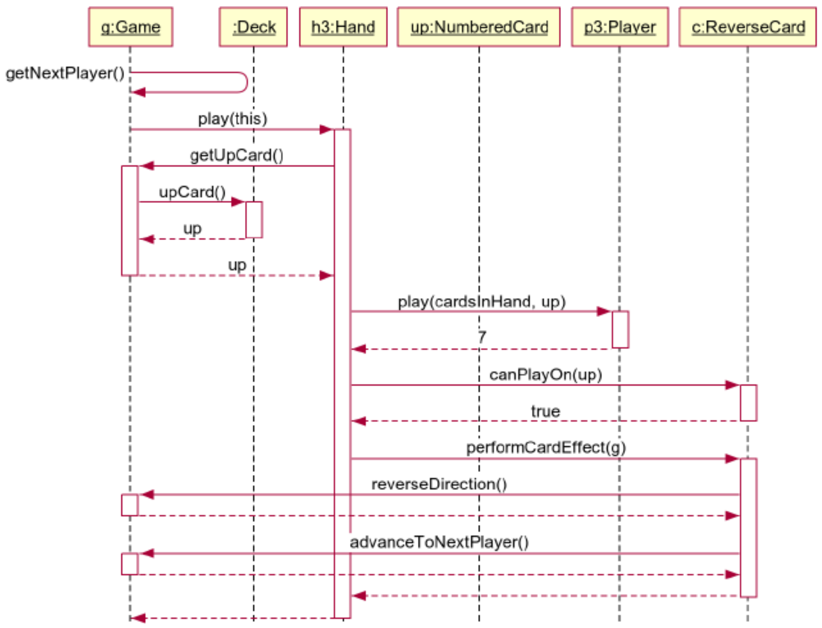


Figure 16.3: An illustration of the double dispatch technique.

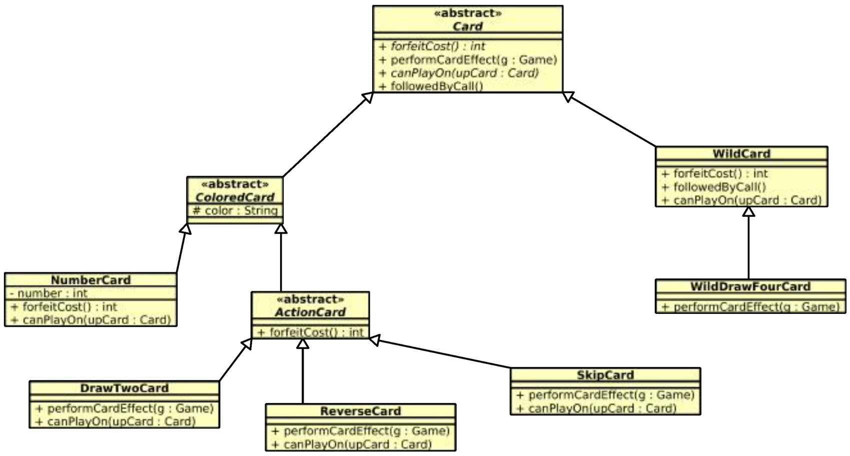

«abstract»
Card
+ forfeitCost() : int
+ performCardEffect(g : Game)
+ canPlayOn(upCard : Card)
+ followedByCall()
«abstract»
ColoredCard
# color : String
WildCard
+ forfeitCost() : int
+ followedByCall()
+ canPlayOn(upCard : Card)
NumberCard
- number : int
+ forfeitCost() : int
+ canPlayOn(upCard : Card)
«abstract»
ActionCard
+ forfeitCost() : int
WildDrawFourCard
+ performCardEffect(g : Game)
DrawTwoCard
+ performCardEffect(g : Game)
+ canPlayOn(upCard : Card)
ReverseCard
+ performCardEffect(g : Game)
+ canPlayOn(upCard : Card)
SkipCard
+ performCardEffect(g : Game)
+ canPlayOn(upCard : Card)

Figure 16.4: The `Card` class hierarchy from Chapter 15's Uno game example.

Chapter 17

Use cases

During my start-up-company days[1], it occurred to me that in order to have a successful product, you need to do two important things:

1. Build the right thing.
2. Build the thing right.

By the first of these items, I mean you have to create a product that actually helps people, that truly meets a need, lessens some pain, scratches an itch, or makes their life better in some way. The second one means you have to engineer it well: to make it efficient and proficient, with an elegant design that is easily maintainable, reasonably bug-free, and adaptable to future technology changes.

Thought experiment: if your project team could only manage to do well at *one* of these activities, which would you choose? (Take a moment and consider before reading further.)

Here's my answer. It is certainly important to design and code things well so that your software product will be robust, flexible, and extensible for the long haul. But if you can only excel at one of the two above items, my advice is to make darn sure it's the first one.

Here's why. If you screw up the second one, you're going to have a bunch of pissed-off customers, and of course nobody wants that.

[1]None of my entrepreneurial endeavors ended in IPOs or lucrative buy-outs, partially because of the lessons of this chapter.

Twitter and Facebook will light up with complaints about how your product is buggy, doesn't do what was advertised, has trouble integrating with other software, keeps missing release dates, and so forth. And yes, that can indeed be a headache.

But if you screw up the first one, you're going to have *no* customers. And believe me, that's a lot worse. Your well-built, snazzy-looking, bug-free little product isn't going to get any airtime on social media because *nobody cares*. It simply isn't something people find worth using, and so all the great engineering in the world isn't going to be of any use.

Incidentally, if there's lots of complaining on the Internet about how your product is buggy, that's actually a really nice situation to have. It means that people are using your code, and that they care enough to gripe. You've (partially) solved a problem that they genuinely want solved, and that means that ultimately, if you can manage to get #2 under control, you're going to have a market and a chance at a big success.

Here's another thought experiment: why, if #1 is the most important thing to get right, do we spend almost the entire Computer Science curriculum teaching students how to do #2 well? Think about it: just about every CPSC course you've taken (and will ever take) involves some aspect of #2.

I think there are several answers to this, but much of it comes down to the fact that #1 is just harder to teach. It's certainly not as predictable as #2 is. Not as much is known about it. For the various aspects of #2, there are quite a few reliable and even quantifiable techniques that the Computer Science community has discovered and which have stood the test of time. If you follow best practices, you're going to end up building your code right. But knowing what program to write in the first place is a different realm entirely, and it requires a lot of intuition about people and their fickleness.

Steve Jobs was a genius at #1, and he had Steve Wozniak as his wingman doing #2. When top engineers came and went at Apple, the company could survive the changes because the principles they

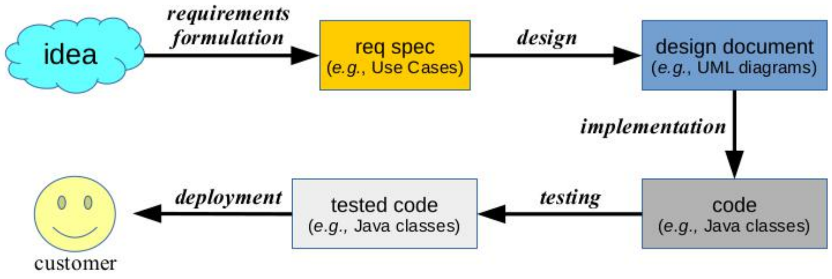


Figure 17.1: A simplified depiction of the software development lifecycle.

were using were transcendent. When Jobs wasn't there, though, you could sure see the difference.

17.1 Capturing requirements

This chapter is a brief look at #1. It's a different phase of the **software development lifecycle** than you're used to focusing on. A simplified picture of this lifecycle is given in Figure 17.1. Each box represents an **artifact**, which as you may remember (p. 137) means the tangible result of some software development activity. The arrows between boxes are labeled with those activities, or "processes." During the time that your development team is working on a particular process (sometimes called "being in a particular **phase**") they are focused on producing the artifact at the end of the arrowhead. That artifact will capture the result of their thinking in a tangible form, where it can then be the input to the next phase.

Any application starts with an **idea**, of course, which is "something that could exist, but doesn't yet." For bite-size projects, one could set to work coding up the idea directly, dispensing with most of this diagram. But for larger projects this doesn't work very well. The first problem is that there is often not enough detail specified about "the idea" yet to know how to proceed. In other words, the design team doesn't yet fully know what the **requirements**[2]

[2]"**Requirement**" is just a loose term for "something the proposed software

are for the system they're being tasked to build. The activity of **requirements formulation** is intended to remedy this situation. There have been various methodologies proposed for coaxing a more detailed description of what the proposed software is supposed to actually *do*, one of which (Use Cases) is the subject of this chapter. Regardless of how they are elucidated, though, the result is some form of **requirements specification** (or "req spec") as we first saw on p.229.

Churning out a **design document** from a req spec is the subject of **design**, of course, which we covered in chapters 15 and 16. The term "**implementation**" is a fancy word for "coding" in a programming language like Java. The **testing** phase typically involves both **unit testing** and **system testing**, which work at different levels of granularity to ensure that each individual software component works according to its specification, and that the system as a whole does. Finally, **deploying** the product to the user base is the culmination of a release cycle, and is usually cause for much celebration (and sometimes, alcohol.)

As I indicated, Figure 17.1 is simplified to the point of being naïve. For one thing, in a real life software process the various phases loop back upon each other: you always learn things in one phase (say, design) that make you go back and revise the work in an earlier phase (say, requirements formulation). Also, it's rare that an entire, fully-functional system gets built in just one execution of this step-by-step chain. Today's software teams **iterate** through this process, or portions thereof, multiple times as they converge on a fully-implemented product. Moreover, larger projects have different groups of people working on different parts of the lifecycle: a "requirements team" with input from marketing, a "design team" led by an architect that focuses on strategic implementation concerns, a "programming team" to actually write the code, "test teams" and "QA (quality assurance) teams" to handle the last few phases, *etc.* Figure 17.1 is really a caricature of the necessary phases, and one

system is *required* to do." Whatever form they may ultimately take, "the requirements" spell out what the design team needs to know in order to build what's expected of them. The requirements are a sort of contract between the management and engineering teams.

simple way they might fit together.

For now, though, the diagram suffices for our needs. My only real motive for showing it to you is to help you mentally place this chapter's content in the proper overall position, namely:

This chapter concerns the ***requirements formulation*** *phase.*

Unlike everything else we've discussed, we're not writing code here, or even figuring out a UML design. We're simply describing **what** our object-oriented program needs to do, when it is ultimately built.

"What" vs. "how"

The dichotomy between "what" and "how" runs deep through human thought, well beyond just software development. Think of a military hierarchy in wartime. The general may decide that **what** needs to happen in a particular campaign is to cut off enemy access to a water source, and focus troops in the western flatter region where the enemy is vulnerable. The Lieutenant Generals who work under him, however, need to take those "whats" and figure out "**how**" to make each of them happen. The Lt. Gen. assigned to the water source task may decide that a quiet amphibious landing upstream from the source, simultaneously with a small group of shock troops at water's edge as a distraction, is just the ticket. So that's **what** this Lt. Gen. decides to do. Each of his or her Colonels then need to unpack those overarching plans to the next level and decide **how** to implement them. And so forth. Every level of the hierarchy is a "what" for the higher-up and a "how" for the underling. The same phenomenon is seen in the Org Chart for a business, a government agency, a sports team, or even a family.

Another pair of terms for this phenomenon are **strategy** and **tactics**. Think of "strategy" as the "what," and "tactics" as the "how." To continue the military analogy: sending a group of aircraft to intercept an incoming bomber squadron might be considered a *strategic* decision. One of those aircraft banking left and then gaining altitude in a flanking maneuver, however, is a *tactical* move. And this dichotomy persists throughout all levels of the plan: something that's a "what" to a major becomes a "how" for his captain to solve, and "what" his captain decides gets fleshed out in "how" his lieutenants decide to operationalize it, *etc.*

By the way, you'll sometimes hear people complain about "**micromanaging**." Here's my definition of the term: when someone "above" you in the hierarchy is supposed to tell you **what** to do, but instead starts telling you **how** to do it, they're micromanaging you. It's a chafing feeling, and can quickly lead to resentment, because your colleague is really overstepping their bounds. They should be outlining what the requirements of the task are, and deferring to you on how specifically to make that happen. On the other extreme are managers who fail to completely specify the "what," leaving you, as the "how" person, with insufficient information about how you might proceed. A well-functioning organization is one where everyone understands and honors these boundaries and is thus able to carry out a complex task composed of many interlocking levels.

17.2 Use Cases

Use Cases are one methodology for capturing[3] requirements. Rather than describing the entire system in a block of text, as in our examples on p. 231 and p. 246, we isolate and identify individual units of functionality that a user of the system has available to her, and describe each one. Each of these units of functionality is called a **use case** (pronounced "YOOS case," not "YUZE case.")

Here are some important terms:

Actor: A role that a person plays when interacting with a system.

Use Case (UC): A function of the system that yields a result of value to an Actor.

UC description: A detailed specification of exactly what happens when the Actor executes the use case, including any important variants.

[3]"Capturing" means "identifying and then precisely describing."

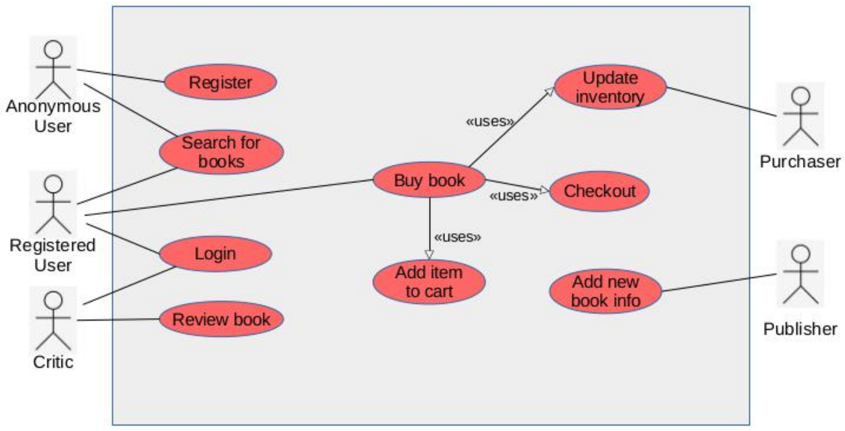


Figure 17.2: A Use Case diagram for an online bookseller.

UC diagram: A mostly-useless picture showing which Actors are intended to perform which UCs. Can be used to impress managers, and makes a good cover page.

You can see that I'm slightly cynical about that last item. Unlike other UML diagrams we've seen (class diagrams and sequence diagrams), a Use Case diagram contains *almost zero information.* That's because the UC descriptions – which are the important part of all this – tell you everything the diagram tells you, and much more.

For the record, though, Figure 17.2 presents a UC diagram for an online bookseller (like Amazon). The large rectangle represents the **system boundary**; in other words, the stuff inside the box is functionality present inside the software system being described. Actors are shown as stick figures, and UCs as ovals. A line from the former to the latter means "a person acting as this role can execute this use case." The "<<uses>>" arrows (with an open-triangle, like inheritance on a UML class diagram) indicate a sort of "subroutine" relationship: in this case, the use case "Buy book" will entail running the "Add item to card," "Checkout," and "Update inventory" use cases.

About actors

There's a couple things worth mentioning about Actors. For one, different actors can sometimes execute the same UC. In our diagram, both the Anonymous User and Registered User actors can "Search for books," and both the Registered User and the Critic can "Login." This is clear from the diagram, and easy enough to understand. The second thing worth mentioning, however, is not explicit on the diagram, but is equally important: **an actor is a *role* that a person may play, not the person herself.**

What I mean is this. Someone might look at Figure 17.2 and say, "wait a minute – mightn't a Critic who writes book reviews also sometimes buy books on the site? Isn't it limiting to disallow Critics from buying books?" The answer is: the same human being may indeed sometimes review books, and sometimes buy books. But *she is acting in different roles when she does so.* When this person writes a book review, she is in the role of a "Critic" actor; when she makes a purchase, she is acting as a "Registered User" actor. So the Actors simply represent the various different capacities in which human beings can act as they use the system. There is certainly nothing preventing a person from embodying different roles at different times.

UC descriptions

I mentioned that UC *diagrams* are almost completely worthless. Their main value-add in my experience is simply to look pretty and make a nice-looking cover sheet for your Use Case model. Make no mistake, the real work of the requirements phase – and the important information it reveals – is found in the **Use Case descriptions**.

Each UC has a written description that narrates exactly how the actor and the system interact when that function is carried out. Often a requirements team will use a Use Case **template**, which is nothing more than a form with required fields to fill in. If you Google for Use Case templates, you'll find a zillion of them, most of them far too complex (IMO). Here's the one I like to use:

Name:
Synopsis:
Actors:
Precondition(s):
Sunny Day flow:
    1.
    2.
    3.
Rainy Day #1 flow:
    1.
    2.
    3.
Rainy Day #2 flow:
    1.
    2.
    3.

Here's what goes in each field:

Name: The UC's name. Use Cases should always be **verb phrases**, never nouns or other parts of speech. They're titled according to *something the actor does.*

Synopsis: A concise (one-or-two sentence) description of the function. This is mostly necessary so that when someone's flipping through a stack of UC descriptions, they can quickly orient themselves and find the one(s) they're looking for.

Actors: The names of the Actors who are intended to use this functionality.

Precondition(s): A short list of assumptions that *must be true* before this UC can even apply. This helps orient the reader as to where in the grand scheme of things this functionality is expected to take place.

Sunny Day flow: Each of a UC's "flows" is a step-by-step narrative of what takes place. The "Sunny Day" flow (also sometimes called the "primary flow") describes what happens *when* *all goes as expected.* There is almost always only *one* Sunny Day flow, because there's typically only one "way" things can go right.

Rainy Day flow(s): The Rainy Day flows explain exceptional conditions or errors; in other words, what the system should do when things *don't* go as expected. There are often several Rainy Day flows, since there's often several different ways things can go wrong.

Figure 17.3 gives an example Use Case description for the "Register" UC mentioned before. Note especially the level of detail provided. The description avoids mentioning User Interface specifics (like where the button is positioned, or what color it is) but it does specify visual details *where they impact the functionality*, such as "dummy characters" in the password. The line between specifying too little and specifying too much is admittedly a bit fuzzy at times, and every development team settles on their own preferred practices.

The general rule is: the UC flows must be specified in enough detail that the design & programming teams know *what* they're supposed to make the system do. Inevitably there will be details that the implementers have to supply themselves, but the goal is to keep this to a minimum. The UC descriptions, in essence, form a **contract** between the requirements and design/programming teams.

Note also the pointers embedded in the flows: "See Rainy Day #1," "Include Login," and "Go to Sunny Day step 3." These mean just what you think they mean. The first and the last of this triad direct the reader to jump to a different step. The second one refers to another Use Case entirely, specifying that when this UC ends, the appropriate next experience for the actor is the Login UC's sunny day flow.

Tips for good Use Case descriptions

The most important general rule for Use Case descriptions, as with all other documentation, is this: **spend time writing useful in-**

Name: Register
Synopsis: An anonymous user creates a unique identity with the system, to be used to identify this user in this and future sessions.
Actors: Anonymous User
Precondition(s): The user is not logged on.
Sunny Day flow:

1. From the main bookseller.com home page, a "Create login" button is visible.
2. The user clicks this button.
3. The user is presented with a "create a user id" page, and is instructed to enter (and re-enter, for accuracy) their e-mail address and password.
4. The user enters their e-mail address and password, re-typing their e-mail address a second time to ensure accuracy. (See Rainy Day #1.) When entering text in the password field, a dummy character appears for each keystroke rather than the actual character pressed.
5. The system presents a "successful registration" page. <Include Login>

Rainy Day #1

1. If the e-mail addresses do not match, the system presents a message indicating this, and prompts again for e-mail address and password. <Go to Sunny Day step 3.>

Figure 17.3: A sample Use Case description.

formation, not doing busy work. A lot of students find writing Use Cases a drag, because they think they need to force themselves to write a bunch of text documenting what's obvious anyway. Don't do that. Documentation is expensive (in terms of person-hours) to write, to read, and to maintain. So don't over-generate it. Make all your documentation crisp, information-rich, and to the point. Make it as long as it needs to be, but no longer.

Here are some other guidelines:

- Never use the name of a class, method, or variable in a UC description. Remember: UCs are for requirements, not design or implementation.
- Avoid phrases like "*etc.*," "for example," and "and so on." Those are almost always indicators that you are postponing making important requirements-level decisions until the design phase. The requirement phase is where you want to nail those down. (I remember a student team who was working on a social network project, and in one of their Use Cases they wrote, "the user's profile is displayed, containing their username, password, hobbies, relationship status, *etc.*" I told them, "the design team – who you'll be passing these requirements on to – doesn't know how to write code for '*etc.*'!")
- Make your Use Case descriptions only as long and as detailed as they need to be, no more. (And this will vary widely between Use Cases.)
- Focus on the user's *intent*. Why is she using this functionality? What benefit does she gain?
- Identify, and be explicit about, what information is passed back and forth between the user and the system.
- Note that Rainy Day flows must specify *what the system is supposed to do* in each exceptional case, not just *that* an exceptional case may happen. (For example, it is not sufficient for a Rainy Day flow to say, "Rainy Day #1: the two passwords the user typed don't match. End of Rainy Day #1." It's true that you have identified a possible error case that might occur. But the purpose of the Rainy Day flow is to tell the implementation team how they should *handle* that case.

17.3 A more complex example

I'll end this chapter with one more Use Case description that isn't quite as obvious as the "Register" example. This one is for a text-based adventure game like the classic *Zork* games of the 1980's. Players type text commands (like "north" or "take pickaxe" or "examine painting") to specify what action they want to take in a virtual world, and read descriptions of the rooms, items, and other things they encounter.

Figure 17.4 gives a ***bad*** Use Case description for a combat scenario. It is bad because it is underspecified: many questions will remain in the minds of the design & implementation teams after reading this puzzling description. I claim that Figure 17.5, on the other hand, adequately tells the reader what the system should do, in sufficient detail so as to be implementable. See if you agree.

Name: Duel with monster

Synopsis: A combat sequence is initiated between the player and one hostile NPC (non-player character) in the current room.

Actor: Player

Sunny Day flow:

1. The player begins the combat by attacking the monster. ***How does the player do this? What command(s) will trigger a combat?***
2. Either the adventurer or the monster is victorious, based on the combatants' levels, strength scores, and weapons. ***How is this decided? How are the various statistics combined to determine a winner? Is there any randomness involved? Is the entire combat resolved in a single step, or are there multiple attacks before a death?***
3. The player can attempt to exit the combat at any time by typing "`flee`." ***The phrase "at any time" implies the action is ongoing, with punctuated intervals. But this is not explained. The word "attempt" suggests that the flee attempt might not be successful. How is this determined?***
4. If the adventurer dies, the game ends. ***What does the user experience here? Is there an option to restart at a previous save point? Is there an exit message? Does the system just crash?***
5. If the monster dies, the system prints an appropriate message and the player scores points for the combat. ***What is the message? How many points? Are they told that they scored a certain number of points, or is this just present in their total the next time they ask for their score?***

Figure 17.4: A **bad** (underspecified) UC description for the combat Use Case, with ***unresolved questions***.

Name: Duel with monster

Synopsis: A combat sequence is initiated between the player and one hostile NPC (non-player character) in the current room.

Actor: Player

Preconditions: The player is currently in a room with an NPC who is "hostile" (as opposed to "friendly.") The NPC is currently still alive – *i.e.*, its "number of wounds" is less than 3. The player is in possession of an item which has an attack event associated with one of its item-specific commands.

Sunny Day flow:

1. The player begins combat by typing "`attack` NPCname".
2. With equal probability, the system randomly decides which party will be successful in the attack: the adventurer, or the NPC.
   - If the player is chosen, one of the following colorful messages is displayed: "`you hit the nameOfNPC!`" or "`the nameOfNPC is wounded!`" The NPC's number of wounds is incremented by 1.
   - If the NPC is chosen, one of the following colorful messages is displayed: "`the nameOfNPC sneaks in an attack!`" or "`pain rushes through your body!`" The adventurer's number of wounds is incremented by 1.
3. If neither the adventurer's nor NPC's number of wounds is equal to 3, the player can attack again (return to step 1) or issue another command instead. In the latter case, the wound counts remain for both adventurer and NPC (they are not reset back to 0).
4. If the adventurer's wound count is equal to 3, the system prints "`thou art slain!`" Go to <Finish game>.
5. If the NPC's wound count is equal to 3, the system prints "`the nameOfNPC is dead!`" A new item called "nameOfNPCCorpse" is now present in the room. The adventurer's score is increased by 20.

Rainy Day #1 flow:

1. The player terminates the combat by entering a command other than an item-specific command associated with an attack event.
2. In this case, play resumes as though the adventurer had never entered combat – any legal command can be entered with its usual effect. The wound counts of both combatants, however, are maintained indefinitely.

Rainy Day #2 flow:

1. The player attempts to surrender to the monster by typing "`surrender`."
2. The attempt to surrender is denied. The system prints the message "`nameOfNPC takes no prisoners!`" and the adventurer dies. (Go to Sunny Day step 4.)

Figure 17.5: A **good** UC description for the combat Use Case.

Chapter 18

Documenting an API

"**API**" – which historically stands for "Application Programming Interface" – is one of the dumber acronyms you'll encounter. And worse, it's commonly used to mean two different things: (1) a set of classes (and their methods) which a programmer could make use of in their own code, and (2) the documentation describing those classes/methods.[1]

In common lingo, people speak of "programming to an API," which means "writing some code which conforms to those documented classes." Every time you've used an `ArrayList` or a `Scanner`, in fact, you have been doing this. Instantiating such objects, calling methods on them, and (importantly) reading the documentation at `https://docs.oracle.com/javase/8/docs/api/` to find out how they operate is all part of leveraging the built-in Java API for your own purposes.

These days, when we talk about using an API, we often mean writing code that connects over the Internet to some publicly-available service or database of information. Nearly every major Internet player these days – Google, Youtube, Instagram, Flickr, eBay, Twitter, Dropbox, Spotify, Amazon, `data.gov`, GeoDB cities, *etc.* – has

[1]By the way, the term "API" isn't used only for object-oriented software. One could write some old-school procedural code (with functions and data structures, rather than encapsulated classes and methods), describe it, and call that an API as well.

a publicly-accessible API. This allows you to write code (in any language) to connect to it and query it for information, perform commands, make purchases, and so forth. Browse `dev.twitter.com` to get an idea of the rich functionality available to anyone with the technical savvy to understand and exploit an API.

It's an interconnected, collaborative world. Developers rarely write all the code themselves anymore on a little isolated island. Instead, they share code for others to use, and take advantage of what's been shared with them. If you can figure out how to effectively do that, you've increased your programming potential a hundredfold.

18.1 The importance of good docs

Now in order to make it possible for other developers to use the code you so painstakingly wrote, it must be **documented** in a way that is clear, complete, and unambiguous. To appreciate the importance of this, I want to lead you in a thought experiment.

First, pretend you're back in the 1990's, a glorious time to be young and alive. In particular, pretend that *GPS and cell phones are not yet commonly available.* (Believe it or not, this was true in the recent past.)

Let's suppose it's Friday night, and you're going to a party at the apartment of your acquaintance Biff. Biff lives up in North Stafford, and you've never been to his place before. Luckily, your close friend Filbert is also going to the party, and he's been to Biff's on many occasions. You're picking him up at 8pm.

Consider the following two scenarios.

Scenario A: You'll pick up Filbert (and possibly one or two others), and drive together to Biff's apartment.

Scenario B: Filbert calls you at the last minute and says that he's getting a ride with somebody else. He gives you *written directions* to Biff's, however, so that you can get to the party on your own.

My question: in which of the above two scenarios are you *more* likely to successfully arrive at the party without getting lost? Or are both cases equally likely?

The careless thinker might at first conclude that the two cases are equally likely. After all, they both depend on Filbert's knowledge of how to get to Biff's. In one case, Filbert's verbalizing the directions as you drive, and in the other case, he's laying them all out for you in advance. But theoretically, as long as Filbert knows how to get there, you'll be successful in both scenarios.

Theoretically. But in the real world, as everyone knows, it usually doesn't work like that. In scenario A, with Filbert in the passenger seat, you have the chance to interactively ask about every intersection and every turn. But in scenario B, *Filbert had to specify everything perfectly in advance.* He had to describe the route with no errors, since there would be no chance to make corrections en route. He had to anticipate every question you might have, since he wouldn't be there to answer them. That's a lot of pressure on Filbert to give good directions.

Consider the following very realistic possibilities:

- Filbert wrote "left" when he meant "right" in step 3 of the directions because he's human.
- Filbert just plain forgot step 5 of the directions because he's human.
- When he wrote, "turn left at the next opportunity," he meant "at the next *intersection*," and assumed that would be obvious to you. However, you (quite naturally) thought he meant "the very next possible left," which was down a side road.
- A road is closed, or there's a traffic jam, and you need to improvise in order to make it to the party on time.
- *Etc.*

You can think of a dozen more. In all these cases, having Filbert with you in the car allows you to clarify ambiguities, fill in omissions, ask questions as they arise, and change course in response to unexpected circumstances. With the written directions, you have none of those options. Put another way, Filbert isn't even at your disposal in Scenario B: your only asset is Filbert's brain dump, as he was conceiving it at 6:13pm.

And by the way: most people are pretty bad at giving directions.

Collaborating with someone you'll never meet

In case the above analogy isn't plain, Scenario A corresponds to a software development team where your teammates are just down the hall. They're just an email or a Slack away. You can ask questions, report bugs, or even request alterations as the need arises. The pressure is off, as far as documentation is concerned. In fact, why even bother trying to document everything exhaustively in advance, if your teammates can ask focused questions in real time?

Scenario B corresponds to you using a public API. The instructions written by a developer you will never meet are *your one and only chance* to comprehend how to use the thing. Those instructions had better be darned good, because there is no chance to ask questions on the road. They'd better clearly and exhaustively contain *everything* you're likely to want to know.

By the way: most people are pretty bad at writing clear and complete documentation. The good news is that it's possible to improve this through discipline, practice, and painstaking effort.

18.2 JavaDoc: mechanics

One of Java's supplementary (but in retrospect, killer) features was the `javadoc` utility shipped with the JDK. The idea behind JavaDoc was to combine two previously incompatible aspects of code documentation. The key question is: where should the documentation be kept?

On the one hand, it seems that the English text describing how to use a software component (like a class, method, or package) ought to be maintained right alongside the code itself, in the source file. This promotes keeping the code and the docs in sync.

On the other hand, there are clearly many advantages to presenting the documentation in a rich, interactive, point-and-click hypertext format. Then the user can browse it non-linearly, read it with pretty formatting, avoid having to step around the code itself to read the next bit of documentation, *etc.*

So we seem to have two conflicting desires: to keep the documentation close to (and embedded in) the code, and to author it in a more flexible (and ideally, web-browser-accessible) way outside the code.

JavaDoc's innovation was to say: "go ahead and store the documentation in the `.java` files themselves, to promote consistency. But we'll create a separate tool that can examine the `.java` files and extract the documentation portions. The tool will then assemble those into a mini-website that other programmers can conveniently browse."

To accomplish this, we use a special syntax to denote "JavaDoc comments." Recall that one style of comment in Java is the multi-liner:

```
/*
 * This is a regular Java comment, and will be ignored by javac.
 */
```

JavaDoc comments are the same, except that they have a *double* asterisk at the beginning:

```
/**
 * This is a special JavaDoc comment, which will be ignored by
 * javac, but will be extracted by javadoc.
 */
```

You can place JavaDoc comments in three places:

- Immediately before a class definition, to provide a description of that class, and hints as to its usage.
- Immediately before a method definition, to describe what the method does, how to call it, and what will happen in exceptional conditions.

- In a special file called "`package.html`", which will be placed in the source directory for a package, if you're using Java packages.

The `javadoc` utility will automatically identify and extract the English text stored in JavaDoc comments in any of these three places, and assemble them into HTML files in the appropriate way.

Markup and tags

There are also a couple of cosmetic options you can take advantage of in JavaDoc comments. First of all, any valid HTML tag can be used directly in the comment, and will be formatted appropriately in the final mini-website. If you're familiar with HTML tags like "`<b>`" (for boldface), "`<tt>`" (for a monospace, typewriter font) or "`<ul>`" and "`<li>`" (for bulleted lists), you can use them to style your text.

Second, there are special tags called "JavaDoc tags" that can be used to set apart certain meta-information and put them in a special place in the final HTML product. The most important ones are shown in Figure 18.1, though there are others. Each development team acquires their own culture, policies, and procedures that call for different pieces of information to be highlighted.

A representative example showing many of these tags is in Figure 18.2, the HTML for which appears in Figure 18.3.

Generating the mini-website

To actually generate the HTML in Figure 18.3, you'll need to run the `javadoc` command with some options. Generally, if I'm running on a Google Cloud instance, this is how I run it:

```
$ sudo javadoc -d /var/www/html -author *.java
```

The word "`sudo`" at the start of this sentence means "please allow me to execute the following command as the **root** user of the system"; *i.e.*, the super user who has all privileges. The reason

Tag/syntax	Location	Purpose
`@author` Jezebel	class	The primary or original author of the class. Using the first name, username, or initials of the author are common choices.
`@param name` description	method	What one of the arguments to the method means. "`name`" is either the name or the type of argument. "description" should begin with a lower-case letter and end with a period.
`@return` description	method	How the return value of the method should be interpreted. "description" should begin with a lower-case letter and end with a period.
`@throws` type description	method	What type of exception might be thrown from the method and how it should be interpreted. "description" should begin with the word "if" and end with a period.
`{@link className}`	anywhere	Create a clickable hyperlink to the class named.
`{@link className#method}`	anywhere	Create a clickable hyperlink to the method named.
`@deprecated` explanation	class / method	Mark this class or method as old and not to be used by new code. (Still supported temporarily for older code, but intended to be phased out.)

Figure 18.1: Some commonly-used JavaDoc tags, and their meanings.

this is necessary is that the directory `/var/www/html`, which this command says to write content in, is by default not writeable by ordinary mortals. You have to temporarily become Superman in order to write to it, which will require typing your password to confirm you're really Clark Kent.

The "`-d /var/www/html`" bit is a command option with a parameter. The `-d` stands for "directory" and it says that the HTML that `javadoc` generates should be written to this directory. It's a system-specific thing; on Debian Linux (which I install on Google Cloud) this is the directory that Apache Web Server will look in to serve up content to browsers that connect to it. (More on that in a moment.) In other contexts, like if you have a user directory on a shared machine that you don't have control over, you can often substitute something like "`-d /home/yourusername/public_html`" which will write the content to your account's own directory for

```
/**
 * A <tt>Ballplayer</tt> represents a historical baseball player and
 * the composite statistics over his career. Each <tt>Ballplayer</tt>
 * object is associated with one {@link Team} even if he played for
 * multiple teams in his actual career.
 * @author SD
 */
public class Ballplayer {
    ...
    /**
     * Constructs a new <tt>Ballplayer</tt> object with "empty" stats
     *   (<i>i.e.</i>, all set to their initial, default values.)
     * @param name the real (no nicknames) first and last name of the
     *   player.
     * @param uni the most well-known uniform number he played under.
     * @param team the mascot name (not city) of the {@link Team} he
     *   is most commonly associated with.
     * @throws NoTeamException if the <tt>team</tt> parameter does not
     *   correspond to the mascot name of any known {@link Team}.
     */
    public Ballplayer(String name, int uni, String team)
        throws NoTeamException {
        ...
    }

    /**
     * Returns the player's career batting average, measured as total
     *   hits divided by total "at bats." If the number of "at bats"
     *   is zero, returns 0.0 rather than give a divide-by-zero error.
     * @deprecated This method should be eschewed in favor of more
     *   recent stats such as {@link Ballplayer#getOnBasePercentage}
     *   and {@link Ballplayer#getSluggingPercentage}.
     * @return the batting average on a 0.0-to-1.0 scale.
     */
    public double getBattingAvg() {
        ...
    }
}
```

Figure 18.2: HTML and JavaDoc tags in action.

hosting HTML content to the world.

The "`-author`" part of the command says "yes, please *do* extract `@author` information and include it in the HTML. (The default is to not do that, which I've never understood.) Semi-related: by default `javadoc` only includes the **public** classes and methods in the HTML it generates, since JavaDoc is normally used to document public APIs. Sometimes there are reasons to produce JavaDoc content for *everything* in the class files – private, public, or anything else – and to do this you merely need to include a "`-private`" option here as well.

Finally, the "`*.java`" means to generate HTML for all the Java source files in the current directory. If you're using packages, you can instead replace "`*.java`" with a sequence of fully-specified package names that can be located via the `CLASSPATH` variable. Note that you do have to tell `javadoc` to generate the *entire* mini-website at once; if you make just a change or two to one `.java` file, you can't just regenerate the HTML for that one because then the entire mini-website will consist of nothing *but* that one class.

Starting the Web server and connecting

After all the JavaDoc's mini-website content has been generated, you can access it via your browser. You can find out whether Apache Web Server is running by typing:

```
$ sudo systemctl status apache2
```

at the command line. If it gives a message like "whoa, apache2 not installed," then you'll need to install it via:

```
$ sudo apt install apache2
```

If it says it's not currently running, then you'll need to start it via:

```
$ sudo systemctl start apache2
```

Lastly, when this seems to be working, figure out what the external IP address is of your machine (it'll be four numbers, each in the range 0-255, set apart by periods; for example, 35.237.255.14). Then, you can point your browser to `http://thatIPAddress` and you should be able to see your prettily-formatted HTML website like in Figure 18.3.

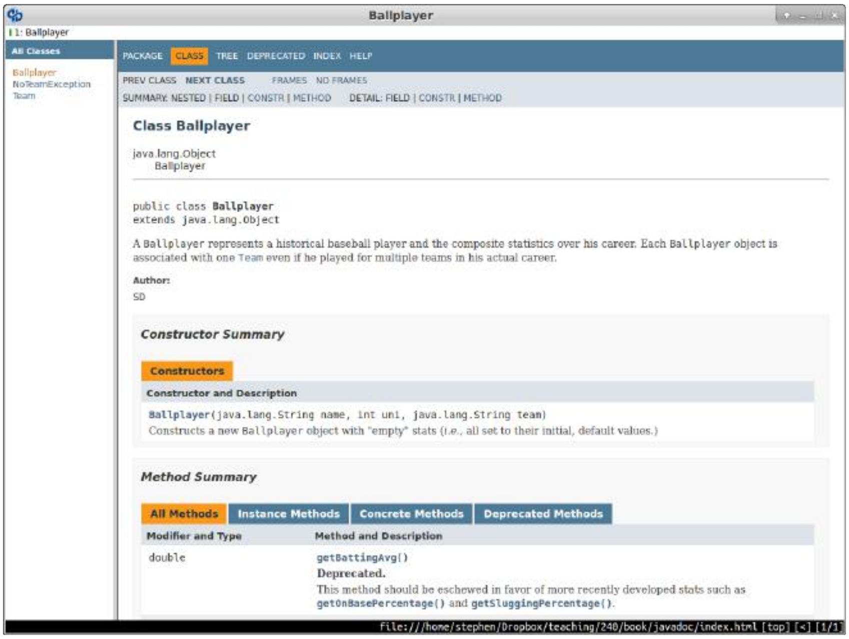
Firewall settings

If you can't reach your JavaDoc site via the above URL, your problem might be that your Google Cloud firewall is blocking the traffic. At the time of this writing, here's how to fix that problem:

1. Go to `https://console.cloud.google.com/networking/firewalls/list`.
2. Click "Create firewall rule."
3. Give it a name like "`allowhttp`".
4. Choose "All instances in the network" from the "Targets" drop-down.
5. Make sure direction is "Ingress" and action is "Allow".
6. For "Source IP ranges" put "`0.0.0.0/0`".
7. Click TCP under "Specified protocols and ports", and put in `80`.
8. Click "Create" to create the rule.

18.3 JavaDoc: content

Okay, so that's all the minutiae of how to get the JavaDoc syntax right and generate the website. You have to know this, but it actually isn't the important part. The truly important question in all this is less-easily defined: how to actually write quality documentation that will communicate effectively to programmers I may never meet?



Ballplayer

1: Ballplayer

All Classes

Ballplayer
NoTeamException
Team

PACKAGE CLASS TREE DEPRECATED INDEX HELP
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class Ballplayer

java.lang.Object
    Ballplayer

```
public class Ballplayer
extends java.lang.Object
```

A `Ballplayer` represents a historical baseball player and the composite statistics over his career. Each `Ballplayer` object is associated with one `Team` even if he played for multiple teams in his actual career.

Author:
SD

Constructor Summary

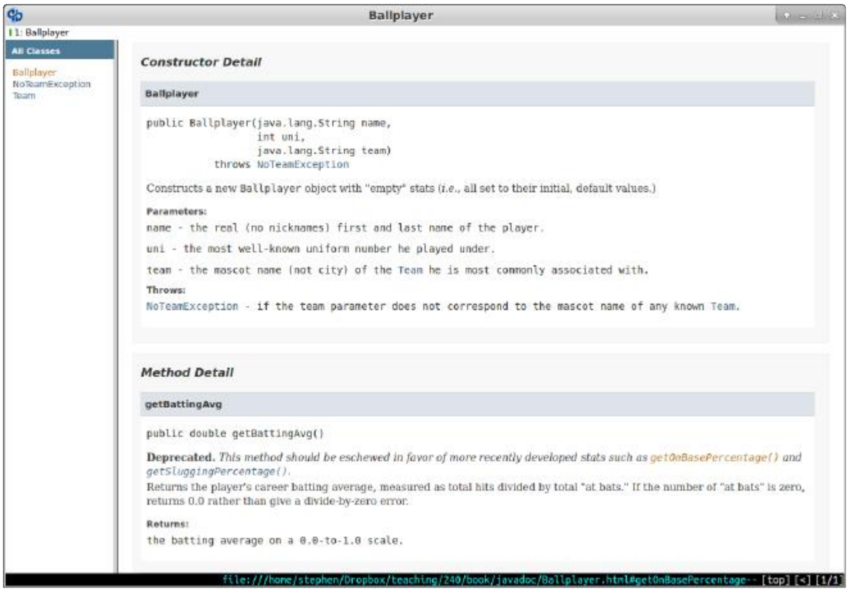
Constructors

Constructor and Description
`Ballplayer(java.lang.String name, int uni, java.lang.String team)` Constructs a new `Ballplayer` object with "empty" stats (*i.e.*, all set to their initial, default values.)

Method Summary

Modifier and Type	Method and Description
double	`getBattingAvg()` **Deprecated.** This method should be eschewed in favor of more recently developed stats such as `getOnBasePercentage()` and `getSluggingPercentage()`.

```
file:///home/stephen/Dropbox/teaching/240/book/javadoc/index.html [top] [<] [1/1]
```



Ballplayer

1: Ballplayer

All Classes

Ballplayer
NoTeamException
Team

Constructor Detail

Ballplayer

```
public Ballplayer(java.lang.String name,
                  int uni,
                  java.lang.String team)
           throws NoTeamException
```

Constructs a new `Ballplayer` object with "empty" stats (*i.e.*, all set to their initial, default values.)

Parameters:

`name - the real (no nicknames) first and last name of the player.`

`uni - the most well-known uniform number he played under.`

`team - the mascot name (not city) of the Team he is most commonly associated with.`

Throws:

`NoTeamException - if the team parameter does not correspond to the mascot name of any known Team.`

Method Detail

getBattingAvg

```
public double getBattingAvg()
```

Deprecated. *This method should be eschewed in favor of more recently developed stats such as `getOnBasePercentage()` and `getSluggingPercentage()`.*
Returns the player's career batting average, measured as total hits divided by total "at bats." If the number of "at bats" is zero, returns 0.0 rather than give a divide-by-zero error.

Returns:

`the batting average on a 0.0-to-1.0 scale.`

```
file:///home/stephen/Dropbox/teaching/240/book/javadoc/Ballplayer.html#getOnBasePercentage-- [top] [<] [1/1]
```

Figure 18.3: The generated HTML for the code in Figure 18.2.

The answer is at once super simple and extremely nuanced. Here is the one and only rule that should guide your API documentation process:

> **"Put yourself in the shoes of a Java developer who has never seen your code before. Write whatever that person would need to know in order to use your code properly."**

You might be surprised how much students (and professionals) struggle implementing this advice. It turns out that the human brain has a very difficult time envisioning what it's like to *not already know* something. "Putting yourself in someone else's shoes" just doesn't come naturally, it seems. Nevertheless, you *must* do it, otherwise your documentation will be pretty useless.[2]

Class documentation

Classes are the easier of the two main components (the other being methods) to write JavaDoc for. That's because the purpose of class JavaDoc is mostly to orient the reader to the class's purpose and what it collaborates with. Still, it's certainly possible to leave important information undefined or ambiguous.

Let's take a look a bad attempt to document the `Team` class from the baseball simulator example, followed by some better ones.

[2]As an aside, this same mantra – "put yourself in the shoes of someone who doesn't already know" – is the essence of another common activity: *teaching*. With few exceptions, I've discovered that a good teacher is one who can mentally put themselves in the shoes of their students, and remember what it was like to not already know the material. Conversely, bad teachers are inevitably poor at this exact skill, which is why it can sometimes seem as if they're assuming you already possess the knowledge before they began to teach it!

First (bad) attempt:

```
/**
 * A <tt>Team</tt> represents a group of {@link Ballplayer}s
 * who play baseball together.
 */
public class Team {
    ...
}
```

This JavaDoc succinctly sums up what the `Team` class is for, but I claim it leaves out at least two important pieces of *non-obvious* information. “Non-obvious” is the key word here: the stuff that would be obvious to a reader isn’t particularly important to document. It’s the things that *aren’t* clear that deserve attention.

One thing this JavaDoc is missing is the motivating *reason* for using the class. A fellow developer might read this and say, “okay, a `Team` is a group of `Ballplayer`s, but if that’s all it is, I’ll just use `ArrayList` instead, since I’m more familiar with it.” It’s a fair point: our JavaDoc hasn’t made the sale as to why it’s worth using.

Okay, so why *is* it worth using? At least two reasons are (1) it can be used as input to the `Simulator` class, in order to simulate a virtual ballgame, and (2) it has useful methods on it that provide summary statistics for the entire team. So let’s say that:

Second (better) attempt:

```
/**
 * A <tt>Team</tt> represents a group of {@link Ballplayer}s who play
 * baseball together, and can compute summary statistics about the
 * group's past performance. Two <tt>Team</tt> objects are required to
 * run a single-game simulation (see {@link Simulator#simSingleGame}.)
 * Methods like {@link Team#getTeamBattingAverage()} and
 * {@link Team#getWonLossRecord()} can be called to get aggregate
 * information about the team's performance.
 */
public class Team {
    ...
}
```

Better. And now for the second thing I found missing. One crucial aspect of documentation to include – and one that is easily overlooked – is the *assumptions* the developer was making when she wrote the class, but which a user of the class might not make. In this case, a glaring question is: "can a `Ballplayer` be a member of more than one `Team`?" Frank Robinson, for instance, was a Hall of Fame outfielder for both the Cincinnati Reds and the Baltimore Orioles. What would happen if I attempted to add him to two different `Team` objects? Is it perfectly okay? Is it forbidden, but not checked by the code? Or will adding him to the second object trigger a run-time exception?

It's imperative that we know, because all three of the above behaviors are reasonable. In order to use this `Team` class, we need to know which one is the true behavior.

So let's add that information in, together with an author tag, and call it done (at least for now):

Final attempt:

```
/**
 * <p>A <tt>Team</tt> represents a group of {@link Ballplayer}s who
 * play baseball together, and can compute summary statistics about
 * the group's past performance. Two <tt>Team</tt> objects are required
 * to run a one-game simulation (see {@link Simulator#simSingleGame}.)
 * Methods like {@link Team#getTeamBattingAverage()} and
 * {@link Team#getWonLossRecord()} can be called to get aggregate
 * information about the team's performance.</p>
 *
 * Note that a <tt>Ballplayer</tt> can be a member of <i>any number</i>
 * of teams. (This will happen if a player was traded during his
 * career, for instance.) In this case, the summary statistics for each
 * <tt>Team</tt>, and its performance in a simulation, will take place
 * as though that player's entire career stats applied to <i>each</i>
 * of his teams.
 *
 * @author Stephen
 */
public class Team {
    ...
}
```

Apparently, `Stephen` has decided to write the `Team` code to deliberately *allow* a ballplayer to be a member of more than one team. This JavaDoc also spells out a possibly unexpected caveat of this decision: the system doesn't separately keep track of which stats a player accumulated on which of his teams, but simply lumps them all together in a single `Ballplayer`. This has an important impact on what to expect from the simulator's behavior, if (for instance) a player was traded late in his career. If you have an aging Ricky Henderson on your L.A. Dodgers squad, that team is going to benefit from *all* Ricky's legendary base-stealing, even though almost all of it occurred with different teams earlier in his career.

The point is that this clarifies an important case that perhaps wasn't obvious at first. In general, it's really easy to think of only the "sunny day" scenarios, and to write the documentation describing what is normally pretty obvious anyway. It's harder to step out of the box and recognize what cases *aren't* so obvious – in this example, the multiple-team-players question – and give the user of the class guidance on what to expect.

Method documentation

Method documentation is a higher-stakes affair than class documentation, simply because there are more details to remember and to get right. It's not often that a fellow developer gets off in the weeds about what a class is even used for; but it's not rare for them to code to a method incorrectly because the JavaDoc is spurious or misleading.

Taking again the baseball example, let's look a bad attempt to document the `.getOPS()` method.

First attempt:

```
/**
 * Return the player's OPS.
 */
public double getOPS() {
    ...
```

This is an example of wasting time typing. The programmer might as well have typed nothing at all, since "returning" was implied in "`get`..." and `OPS` remains undefined. Let's try again.

Second attempt:

```
/**
 * Return the player's career OPS (On-base-plus-slugging)
 * statistic, defined as the player's on-base percentage
 * plus his slugging percentage.
 */
public double getOPS() {
    ...
```

This is much more useful, at least, since it defines for a reader who may not be familiar with the more advanced Sabermetric stats what "OPS" even is. Still a few things missing, though. For one, the on-base percentage and slugging percentage are both defined on a 0.0-to-1.0 scale rather than a 0-to-100 scale, and so "percentage" is a very misleading (incorrect, actually) word. We don't want to move away from standard baseball conventions, so we'll keep the terms but then make the scale clear in a note in the JavaDoc.

Speaking of standard baseball conventions...all the stats like batting average, slugging percentage, OPS, *etc.* are traditionally reported to exactly *three* decimal places. We say "Simpson is hitting .325," not "Simpson is hitting .3257886442." Another question that arises, then, is: do methods like `.getBattingAvg()` and `.getOPS()` return a figure rounded to the three decimal places of convention, or do they return hits-divided-by-at-bats (or whatever) with all possible precision? One can see advantages to doing it either way; the point here is that the JavaDoc must specify which it is.

Whether to document here what "on-base percentage" and "slugging percentage" themselves are is a judgment call. If there are other methods on the `Ballplayer` class specifically for those two stats, then it's probably better to link to them in the `.getOPS()` JavaDoc rather than duplicate the text.

Finally, you always have to ask yourself "what corner cases are there?" What scenarios might unfold that are unusual and need special treatment? Here, the important one turns out to be a ballplayer *with no at-bats.* In this case, both his on-base percentage and his slugging percentage will have a denominator of *zero*, which is an illegal mathematical operation. ("Zero hits divided by zero at bats" is not zero, but undefined.) Traditionally, players with no batting chances are reported as having a .000 batting average, slugging percentage, on-base percentage, *etc.*, so it makes sense for our method to do that here. When it does, though, it's strictly speaking going beyond the definition of "successes divided by attempts" that all these averages are based on.

Answering all these questions adequately, then, leads to this JavaDoc:

Final attempt:

```
/**
 * Return the player's career OPS (On-base-plus-slugging) statistic,
 * defined as the player's on-base percentage plus his slugging
 * percentage. On-base "percentage" is computed on a 0-to-1 scale,
 * not 0-to-100 (and slugging "percentage" is similar, though it can
 * be as high as 4.0 (all home runs)), so this method's return value
 * should not be interpreted as a "percentage" either.
 *
 * Although OPS is typically reported to three decimal places, this
 * method will not perform any rounding to ensure that; full precision
 * to as many decimal places as the system allows will be reported.
 *
 * For <tt>Ballplayer</tt>s with zero plate appearances, this method
 * will return 0, not a divide-by-zero error.
 *
 * @return the player's career OPS statistic.
 */
public double getOPS() {
    ...
```

You may be thinking that the `@return` line at the end doesn't really add anything useful. You would be correct. I included it only for completeness; in general, it's fine to leave out redundant information. Some developers like to use the "`@`" tags religiously, while others like to put key information in the running text of the JavaDoc. This is a stylistic choice, and either way is okay. The crucial thing is that *the information has to be present somewhere.*

You also may be thinking that it's tough to come up with all this stuff. You would also be correct about that. Once you see me explain that "zero plate appearances" is a non-obvious special case, or that it's an open question whether the method would round to three decimal places, you can probably say "oh yeah, we'd better mention that detail." Of course, the challenge is to recognize what those details are *before* they're pointed out to you.

Honestly, I can think of no way to make this easier other than (1) practice, and (2) really truly trying to put yourself in the mindset of a new developer. It's hard to pretend you're someone else – and to momentarily, deliberately forget what you know – but it's not impossible. And as I said earlier, it's really the key to communication of all kinds.

18.4 How to lose a battle through bad documentation

I'll close this chapter – and book – with a somewhat humorous anecdote which was nevertheless deadly serious in its ramifications.

On the eve of the Civil War's Battle of Fredericksburg, on December 11, 1862, General Ambrose Burnside was running the show for the Union. His northern soldiers outnumbered the Confederates almost two-to-one, and they had had months to prepare their crossing of the Rappanhannock river and the siege of the town. Abe Lincoln and the other civilian leaders of the north expected a great, perhaps decisive, victory.

Battles sometimes come down to small things. In this case, Burnside was so swamped with preparations the night before the battle that he had only one hour of sleep. That may explain the quality (or lack thereof) of his last-minute orders to his generals. Here's an excerpt of what he wrote to Major General William Franklin in the wee hours of the morning:

"Keep your whole command in position for a rapid movement down the Old Richmond Road and send a division at least to seize, if possible, the height near Capt. Hamilton's, taking care to keep it well supported and its line of retreat open."

Franklin's reaction, upon reading this note at 4am, could be described as: "***Huh??***"

If Burnside had had at least another few hours of sleep, he would doubtlessly have written more coherently about what he wanted. But without clear directions, and guided only by the above gibberish, Franklin couldn't really figure out what to do. His troops floundered ineffectively most of the day. This helped produce 12,653 casualties, two mortally wounded Union generals, and an unprecedented disaster that came perilously close to ending the Civil War prematurely in favor of the South.

It just goes to show how absolutely crucial written communication can be. All the armies in the world – and all the Java coding chops in the world – will profit you nothing if you can't effectively give instructions on how to use them.

Index