Contents

Chapter 0

Getting Started

0.0 Introduction

In this chapter you will learn how to prepare your computer so it can do all of the things you will learn about in this book. There are three major things that need to be addressed: getting a text editor, installing Python, and installing Java. First, a couple of preliminary notes.

A Word About Your File System It is a smart idea to create a directory to hold all of your programs. Do this in your home directory. If you are using a text editor other than vim, avoid storing your stuff in the editor's directory tree; the path is long and annoying. If you upgrade your editor, you can lose all of your programs.

A Convention We shall refer to a *command window* as a PowerShell, cmd, or Mac/UNIX terminal window. The system prompt for any of these window will be shown as `unix>`.

We will run both Python and Java out of a command window, so get used to the command-line interface for your computer.

0.1 Server Access

If you are given a server account, you will need some (tiny) pieces of software to connect to it from your PC. This server will likely have both Python3 and Java installed.

Your system administrator will tell you three things about your account.

1. Your server's name, exxample `cs.ncssm.edu`
2. Your user name, example `hart21g`
3. A password

Windoze Obtain puTTY from
`https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html`".

This will enable you to connect to a server over the network. When you open PuTTy, a window will pop up. Put your server's name in the box labeled "hostname." Under Port, select 22. Then in the Saved Sessions area, type myLogin. Hit the Save button.

Now double-click on the word myLogin in the Saved session area. You will see a terminal window appear on your desktop.

Mac or UNIX Do this in a command window.

```
unix> ssh yourUserName@yourServerName
```

replacing yourUserName with your acutal user name and yourServerNamme with your server's name. A terminal session with the server will begin.

For Both You can enter your user name under `login:` and hit the ENTER key. Next, type your password under `password:`

Note: You will not see any characters appear when typing your pasword. If you have successfully logged in, type `exit` to cleanly log out.

Once you have the basics of login and logout mastered, the next step is to install file-transfer software. Go to the FileZilla download site at `https://filezilla-project.org/download.php?type=client` and download the client.

Once it's installed, fire it up. Enter your login name, server name, and password into the appropriate boxes. Select Port 22. Once you launch it, you will see your local machine on the left and the server on the right. File transfer works by drag-and-drop. It's butt simple, which is why we recommend it for everyone.

Editing Environments You will need a plain-text editor for creating source code files.

You cannot edit source code files using a word processor; they are full of hidden stuff for formatting that will prevent your programs from running. These editors color your code in a manner that makes it easy to spot misspellings and mistakes. Here are some recommended possibilities. All are free software.

- The Atom text editor is available here `http://atom.io`. It has many excellent features and provides syntax coloring for all major programming languages. It works on all platforms. Installation is quick and simple. It has extensions called *packages* that give it some useful capabilities.
- Windoze users can download Notepad++ at `https://notepad-plus-plus.org` This is an excellent editor. It will replace the clunky and nigh useless Notepad.
- The VSCode editor comes with many superpowers, You can download it here: `https://code.visualstudio.com/`
- Macs and UNIX boxes come equipped with vi (vim). By default they have no `.vimrc` file. Create this in your home directory and enter this text into it.

  ```
  syntax on
  set et
  set tabstop=4
  set nohlsearch
  set number
  ```

  You can download vim for Windoze from `https://www.vim.org/download.php`.
- You can work with an IDE (I don't like these for beginners). NetBeans, IntelliJ, and Eclipse are all solid choices and are freely available.

0.2 Installing Python

The people at Python make this a very simple process. Go to `https://www.python.org`. Click on Downloads and select your operating system. Obtain the latest Python 3 installer; we will be using Python3 in this book. You should note that Python2 goes end of life in December 2020. When you install, watch for a checkbox asking if you want to update your `PATH`. Check that box to save having to edit your environment variables. To test it open a command window and do the following.

```
unix> python
Python 3.8.5 (default, Sep 4 2020, 02:22:02)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Your version number might differ; it should be at least Python 3.8.

0.3 Getting a JDK

To build Java programs on your box, you will need a Java Development Kit (JDK). This piece of software is available for all major computing platforms. We will step through the process for the Windoze, Mac and Linux platforms.

- **Windoze/Mac** Go to `https://adoptopenjdk.net/`, and you can get the most current JDK (as of 11/2020, version 15). The site has complete instructions for doing the install. We will use Java 15, so make sure you are install that version or later.
  When the "custom install" window comes up, check the boxes for "update `PATH`" and "update JAVA_HOME." Then you will not need to edit your environment variables to get the commands `java` and `javac` onto your path so you can use them in the command-line interface.
  If it has installed
  properly, you should see this in a command window.

  ```
  (base) MAC:Fri Nov 27:15:27:java> java -version
  openjdk version "15.0.1" 2020-10-20
  OpenJDK Runtime Environment AdoptOpenJDK (build 15.0.1+9)
  OpenJDK 64-Bit Server VM AdoptOpenJDK (build 15.0.1+9, mixed mode, sharing)
  (base) MAC:Fri Nov 27:15:27:java>
  ```

- **Linux** You can install this from your package manager or install it from `adoptopenjdk`.

If you are going to work offine, you should also download the Java API documentation and install it on your machine. This documentation is a free "Encyclopaedia of Java" that will be extremely helpful.

How do I know it's working? Create this program in a file named `Foo.java`.

```
public class Foo
{
}
```

Now open a command window (cmd or PowerShell on windoze) and compile your Java program as follows.

```
unix> javac Foo.java
```

Then, list your files; you should see a file named `Foo.class`. If so, you are golden. Go ahead and remove it.

Exercise Take some time to customize your text editor. Here are some suggestions. The `.vimrc` file takes care of some of this if you are a vim user. In MacOSX or Linux, you can use your terminal preferences to make these changes. Here are some things you might like to do.

1. Change the background color from white to an off-white color such as `0xFFF8E7`. This is much easier on your eyes; staring at a white screen increases eye strain.
2. Adjust the font size to your liking, but do not change from a monospace font. We recommend a monospaced font such as Courier or Courier New.
3. Have line numbers displayed, since error messages in Java often cite errors by line number.
4. You can have a dark background if you wish.
5. Set your indent level to 4 spaces.

Chapter 1

Linux

1.1 Introduction

You are probably used to running a computer with a graphical user interface (GUI) that gives you a desktop and icons to work with using your mouse and keyboard.

You likely run Windoze, MacOSX, or you may be running a Linux GUI on your machine. The GUI allows you to communicate with the operating system, which is the master program of your computer that manages such things as running applications and maintaining your file system.

In this book, we will study the Linux operating system in its command-line guise. You will control the computer by entering commands into a text window called a *terminal window*. Typically they look something like this.

The name "terminal" name harkens back to the days when you had an actual appliance on your desk consisting of a (heavy) CRT screen and keyboard that was connected to a computer elsewhere. Your terminal window is just a software version of this old appliance. You will enter commands into this window to get the remote machine you are communicating with to perform various tasks.

The text string `morrison@odonata` appearing in the window is called the *prompt*. Its presence indicates that the computer is waiting for you type in a command. Your prompt will likely be different.

1.2 In the Beginning ...

As we progress, everything will seem unfamiliar, but actually relate very directly to some very familiar things you have seen working with a computer having a GUI. We assume you have basic proficiency using some kind of computer such as a Mac or a Windoze box, and we will relate the things you do in Linux to those you do in your usual operating system.

Log in to your UNIX account. If you are working in a Linux GUI, or a Mac, just open a terminal session. The first thing you will see after any password challenge will resemble this

```
[yourUserName}@hostName yourUserName}]$
```

or this

```
[yourUserName}@hostName ~]$
```

On a Mac, it will resemble this

```
John-Morrisons-MacBook-Pro:~ morrison$
```

The presence of the prompt means that Linux is waiting for you to enter a command. The token `yourUserName` will show your login name. Your prompt may have an appearance different from the ones shown here; this depends on how your system administrator sets up your host or on the type of Linux distribution you are using. The appearance of your prompt does not affect the underlying performance of UNIX. In fact, the properties of your session are highly configurable, and you can customize your prompt to have any appearance you wish.

To keep things simple and uniform throughout the book, we will always use `unix>` to represent the UNIX prompt. You will interact with the operating system by entering commands, instead of using a mouse to push buttons or click in windows.

When you see this terminal, a program called a *shell* is running. The shell takes the commands you type and passes them on to the operating system (kernel) for action. You will type a command, then hit the ENTER key; this causes the command to be shipped to the OS by the shell. The shell then conveys the operating system's reply to your terminal screen. Think of the shell as a telephone through which you communicate with the operating system. This analogy is only fitting since UNIX was originally developed at AT&T Bell Labs.

We will begin by learning how to interact with the file system. This is what give you access to your programs and data, so it is very fundamental.

1.3 Anatomy of a Command

Every UNIX command has three parts: a name, options, and zero or more arguments. They all have the following appearance

```
commandName -option(s) argument(s)
```

Notice how the options are preceded by a `-`. Certain "long–form" options are preceded by a `--`. In a Mac terminal, all options are preceded by a simple `-`.

A command always has a name. Grammatically you should think of a command as an imperative sentence. For example, the command `passwd` means, "Change my password!" You can type this command at the prompt, hit enter, and follow the instructions to change your password any time you wish.

Arguments are supplementary information that is sometimes required and sometimes optional, depending on the command. Grammatically, arguments are nouns: they are things that the command acts upon.

Options are always, well, ... optional. Options modify the basic action of the command and they behave grammatically as adverbs. All familiar features of a

graphics-based machine are present in Linux, you will just invoke them with a text command instead of a mouse click. We will go through some examples so you get familiar with all the parts of a Linux command.

Two very basic Linux commands are `whoami` and `hostname.` Here is a typical response. These commands give, respectively, your user name and the name of the host you log in to.

Now we run them. We show the results here; your computer will show your login name and your host name. Here is what they look like on a server.

```
unix> whoami
morrison
unix> hostname
carbon.ncssm.edu
unix>
```

Here is their appearance on a PC (A Mac in this instance).

```
unix> whoami
morrison
unix> hostname
John-Morrisons-MacBook-Pro.local
unix>
```

We will next turn to the organization of the file system.

1.4 Managing Directories

We will do a top–down exploration of the file system. In this spirit, we will first learn how to manage directories; this is the UNIX name for folders. You will want to know how to create and manage folders, ummmm... directories , and how to navigate through them.

You have been in a directory all along without knowing it. Whenever you start a UNIX session, you begin in your *home directory*. Every user on a UNIX system owns a home directory. This is where you will keep all of your stuff. You will see that ownership of stuff is baked right into a UNIX system.

To see your home directory, type `pwd` at the UNIX prompt. This command means, "Print working directory!" You will see something like this.

```
unix> pwd
/home/faculty/morrison
```

This directory is your *home directory*. Whenever you start a new session, you will begin here. This is the directory where all the stuff that belongs to you is kept.

In this example,`morrison` is a directory inside of `faculty`, which is inside a directory `home`, which is inside the *root directory*, `/`. Your home directory will likely be slightly different. It is very common for UNIX systems to keep all user directories inside of a directory named `home`. Often, several different types of users are organized into sub-directories of `home`. You will later see that all directories live inside of the root directory, /. Enter the `pwd` command on your machine and compare the result to what was shown here. Become familiar with your home directory's appearance so you can follow what goes on in the rest of this chapter.

If you are using Linux on your PC, your home directory will likely look like this.

```
/home/morrison
```

This directory structure is exactly the same as your hierarchy of folders and files on a Mac or a Windoze box. You already know that folders can contain files and other folders. This is also true in a UNIX environment.

To make a new directory in Mac or Windoze, you right click in the open folder and choose a menu for making a new folder. In UNIX, the `mkdir` command makes a one or more new directories. It requires at least one argument, the name(s) of the director(ies) you are creating. Let us make a directory by typing

```
unix> mkdir Projects
```

makes a directory called Projects; this directory is now empty. We can always get rid of an empty directory or directories by typing the `rmdir` command like so.

```
unix> rmdir garbageDirectory(ies)
```

In this case, `garbageDirector(ies)` stands for the directory or directories you wish removed.

The `rmdir` command will not remove a directory unless it is empty. There is a way to snip off directories with their contents, but we will avoid it for now because it is very dangerous. For now, you can delete the contents of a directory, then remove the directory. Be warned as you proceed: *When you remove files or directories in Linux, they are gone for good! There is no "undelete."*

If you got rid of the `Projects` directory, re-create it with `mkdir`. To get into our new directory Projects, enter this command.

```
unix> cd Projects
```

and type `ls`. You will see no files. This is because the directory `Projects` is empty, and `ls` by default only shows you the files in the directory you are currently occupying. The command `cd` means, "Change directory!" Having done this now type

```
unix> pwd
```

You will see a directory path now ending in `Projects`.

There is a command called `touch` which will create an empty file(s) with a name(s) you specify. Create files named `moo` and `baa` with `touch` as follows.

```
unix> touch moo baa
```

Then enter `ls` at the command line. This command means "list stuff." You will see just the files you created.

As we said before, The command `ls` displays only files in the directory you are currently occupying. This directory is called your *current working directory*, or `cwd` for short. Every terminal session has a working directory. When you first log in, your working directory is always your home directory.

```
/home/yourUserName/Projects
```

This directory is the `Projects` directory you just created.

If you type `cd` without arguments, you will go straight back to your home directory. This should make you will feel like Dorothy going back to Kansas. Now if we use `pwd` again we see our home directory printed out.

You can also see your home directory anywhere you are by typing

```
unix> echo $HOME
```

The fearsome–looking object `$HOME` is just a symbol that points to your home directory. There are various items like this present in your system. They are called *environment variables*. Other examples of environment varialbes include‘ `$PWD`, which is just your current working directory and `$OLDPWD` which is your previous working directory.

Programming Exercises

1. Navigate to a directory. Then enter this.

   ```
   unix> pushd
   ```

   Then navigate to anohter directory and repeat this a few times. Now alternately type

   ```
   unix> popd
   unix> pwd
   ```

What does this do? Think of Hansel and Gretel!

2. Crawl aroud in your directory strucutre. Each time you enter a new directory type

```
unix> echo  $PWD
unix> echo  $OLDPWD
```

3. Use `cd` to change into some directory. Then type `cd -` and then `pwd`. Repeat this. What does `-` mean?

1.4.1 Processes and Directories

We know that when we log in, we are starting a program called a *shell*. The shell is a process, or running program. Every process has a `cwd` (current working directory). When you type `pwd` into your shell, you are asking the OS to tell you your shell's current working directory. If you log in to a UNIX server in several terminal windows, each runs in a separate shell, so each has can have its own working directory.

Observe that, much of the time, your shell is idle. When you finish typing a command and hit the enter key, that command launches a program, that program runs, and any output is directed to your terminal window.

The command `cd` is a computer program. What it does is it changes the `cwd` of the shell that calls it. Now you know what it means to be "in" a directory: it means the `cwd` of your shell is that directory.

Programming Exercises

1. Enter

   ```
   unix> cd $HOME/Projects
   ```

   and see what happens.
2. Make these directories inside of Projects `labors`, `feats` and `chores`
3. Type `cd labors` at the command line then `pwd`.
4. Type `cd ..` at the command line then `pwd`. What happened?
5. Type `cd ..` at the command line again, then `pwd`. What happened?
6. What do you think `..` is?
7. Type `cd .` at the command line then `pwd`. What happened?
8. Type `ls .` at the command line then `pwd`. What happened?
9. What do you think `.` is?

1.5 Paths

The location of your home directory is specified by a *path* that looks something like this `/home/morrison`. This path is an example of an *absolute path*, because it specifies a location in the file system starting at the root directory.

All absolute paths start with a `/` or a `~`. Here are the three kinds of absolute paths.

- Paths beginning with a `/` are specified starting at the root directory.
- The symbol `~` is shorthand for your home directory. It is an absolute path. Try going anywhere in the file system and type `cd ~`; it will take you straight home, just as `cd` does by itself.
- A path beginning with `~someUserName` specifies the home directory of the user `someUserName`.

Absolute paths work exactly the same, no matter where you are in the file system.

Relative paths are relative to your `cwd`. Every directory contains an entry for its parent and itself. Make an empty directory named `ghostTown` and do an `ls -a`.

```
unix> mkdir ghostTown
unix> cd ghostTown/
unix> ls -a
.  ..
```

If you type `cd ..`, you are taken to the parent directory of your `cwd`; this path is relative to your `cwd`. Any path that is not absolute is relative. When you are navigating in your home directory, you are mostly using relative paths. Note that any relative path can also be represented as an absolute path.

Programming Exercises

1. Try typing `cd ..` then `pwd` a few times. What happens?
2. Type `cd`. Where do you go?
3. Type `cd /bin` (on a mac `/usr/bin`) then `ls cd*` You will see a file named `cd` that lives in that directory.

1.6 A Field Trip

To get to our first destination, type `cd /`. The directory `/` is the "root" directory; it is an absolute path. If you think of the directory structure as an upside-down

(Australian) tree (root at top), the directory / is at the top. Type `pwd` and see where you are. Type `ls`; you should see that the directory `home` listed with several other directories. Here is what the directory structure looks like on a PC running Red Hat Fedora Core 9. Yours may have a slightly different appearance.

```
unix> cd /
unix> ls
bin    etc            lib         mnt   root    srv    usr
boot   home           lib64       opt   run     sys    var
cdrom  initrd.img     lost+found  proc  sbin    tmp    vmlinuz.old
dev    initrd.img.old media       root  selinux vmlinuz
unix>
```

Now type if we type `cd home` then `ls`, you will see one or more directories. On the machine being used here, you would see

```
unix> cd home
unix> ls
guest lost+found morrison
```

This machine has two users, `morrison` and `guest`. Since it is a personal computer, it does not have many users. You may be working on a server in which there could be dozens, or even hundreds of other users who are organized into various directories.

Here is an example from a fairly busy server.

```
unix> cd /home
unix> ls
2016  2018  2020  gotwals          menchini  rash
2017  2019  cs    keethan.kleiner  morrison  rex.jeffries
unix>
```

The directories with the years are directories full of user's home directories. We will list one here. It has quite a few users in it.

```
unix> ls 2019
allen19m   hablutzel19k  laney19m     mullane19n                wang19e
bounds19a  hirsch19m     lheem19h     ou19j                     wolff19o
carter19d  hou19b        lin19b       overpeck19c               yang19j
cini19a    houston19b    liu19c       perrin19p                 zhuang19a
eun19e     houston19p    manocha19a   sakarvadia19m
gupta19a   knapp19t      mitchell19m  villalpando-hernandez19j
unix>
```

See if you can follow this all the way down to another user's home directory. You may be able to list the files there, or even read them, depending on that user's permissions. From this modest demonstration, you see that you can step down through the directory structure using cd. Now we will learn how to step up.

Try typing `cd ..`; the special symbol `..` represents the directory above your `cwd`. Now you can climb up and down the directory structure! The `..` symbol works like the up-arrow in a file chooser dialog box in Mac or Windoze. You saw this when you did the last group of exercises.

Practice this; go back to your home directory. Make a new directory called `mudpies`. Put some files in it. Make new directories in `mudpies`, got down inside these and make more directories and files. Practice using `cd` to navigate the tree you create. When you are done, get rid of the whole mess; remember you have to go to the bottom, empty out the files using `rm` and then use `rmdir` to get rid of the empty directories.

If you type `ls` in a directory, notice how any directories inside it are in differently colored type than regular files. This color is often blue. You can use the `-F` option in `ls` to print directory names with a slash (/) after them. Try this; it was an important option back in the days of monochrome monitors. If you use the `-l` option in `ls`, you will see that in the *permissions column*, the column begins with a `d` for any directory. Here is a possible sample

```
-rw-rw-r--    1 morrison morrison        0 Jun  9 14:54 bar
-rw-rw-r--    1 morrison morrison        0 Jun  9 14:54 foo
drwxrwxr-x    2 morrison morrison     4096 Jun  9 14:54 junk
```

You can see there that `bar` and `foo` are empty files. Notice the `d` at the beginning of the line in `junk`; this tells you `junk` is a directory.

1.7 Making and Listing Regular Files

The operating system is responsible for maintaining the file system. The file system maintained by a UNIX system consists of a hierarchy of files. Two types of files will be of interest to us: *directories* (folders) and *regular files*, i.e. files that are not directories. Regular files may hold data or programs. They may consist of text or be binary files that are not human-readable.

You are used to working with regular files and directories in Windoze or MacOSX. Things in UNIX work the same way, but we will use commands to manage files instead of mouse clicking or dragging.

As we have already seen us now use the UNIX command `touch` to create new files. This command creates an empty file for each argument given it. At your UNIX prompt, enter

```
unix> touch stuff
```

This creates the empty file named `stuff` in your account.

Now let us analyze the anatomy of this command. The name of the command is `touch`; its purpose is to create an empty file. Since you do not see a - sign, there are no options being used. The argument is `stuff`. This is the name of the file you created. Create a few more empty files. Enter these commands

```
unix> touch foo
unix> touch bar
```

You may create several files at once by making a space-separated list as we show here.

```
unix> touch aardvark buffalo cougar dingo elephant
```

Now you have eight new files in your account. Next we will see how to list the files. Enter this command at the UNIX prompt

```
unix> ls
```

The command `ls` lists your files. Notice we had neither options nor arguments. If you created the files using `touch` as instructed, they should appear on your screen like this

```
unix> aardvark bar buffalo cougar dingo elephant foo stuff
```

The command `ls` has several options. One option is the `l` option; it list the files in long format. To invoke it, type

```
unix> ls -l
```

You will see a listing like this

```
-rw-rw-r--    1 morrison morrison        0 Jun  9 10:50 aardvark
-rw-rw-r--    1 morrison morrison        0 Jun  9 10:50 bar
-rw-rw-r--    1 morrison morrison        0 Jun  9 10:50 buffalo
-rw-rw-r--    1 morrison morrison        0 Jun  9 10:50 cougar
-rw-rw-r--    1 morrison morrison        0 Jun  9 10:50 dingo
-rw-rw-r--    1 morrison morrison        0 Jun  9 10:50 elephant
-rw-rw-r--    1 morrison morrison        0 Jun  9 10:49 foo
-rw-rw-r--    1 morrison morrison        0 Jun  9 10:49 stuff
```

The first column reflects the *permissions* for the files. The sequence

```
-rw-rw-r--
```

indicates that you and your group have read/write permission and that others ("the world") has read permission. We will discuss permissions in more detail when we discuss the management of directories.

You can see the name here is listed in two columns; on this machine `morrison` is in his own group. On another system, you may live in group with several other people; if so you will see the name of that group in one of these columns. The zero indicates the size of the file; each file we created is empty. Then there is a date, a time and the file name. This is the long format for file listing; it is seen by using the `-l` option in the `ls` command.

Another option is the `-a` option. This lists all files, including "hidden" files. Hidden files have a dot (.) preceding their name. To see them, enter

```
unix> ls -a
```

at the command line. One thing you are guaranteed to see, no matter where you are are are the directories `..` (parent) and `.` (current). If you are in you home directory, You will see the files you saw using ls and several hidden files with mysterious names like `bash_profile`. Do not delete these; they provide the configuration for your account and do things like record preferences for applications you have used or installed. You can also list all files including hidden files by entering

```
unix> ls --all
```

You can use more than one option at once. For example, entering

```
unix> ls -al
```

or

```
unix> ls -a -l
```

```
unix> ls --all -l
```

shows all of your files and hidden files in long format. Try this now on your machine.

Note to Mac Users Mac users should precede verbose commands with a single `-`. So on a Mac, you type

```
unix> ls -all -l
```

and not

```
unix> ls --all -l
```

Otherwise, your Mac will respond with a cryptic error message.

Next we will show how to display a file to the screen. UNIX commands that process files are called *filters*. Filters accept input from a file,

Let us peek inside your `.bash_profile` file. Enter the command

```
unix> cat .bash_profile
```

The command name is `cat`, short for catalog (the file to the screen). The `cat` command is a filter that does not filtering at all; it simply dumps the entire file to the screen all at once. We are using no options, but the file name is an argument to `cat`. If a file is long and you want to see it one screenful at a time, use the filter `more`. The command `more` takes a file name as an argument and shows it on the screen a screenful at a time. You can hit the space bar to see the next screenful or use the down-arrow or enter key to scroll down one line at a time. To exit `more` at any time, type a `q` and `more` quits. You can use several arguments in `cat` or `more` and the indicated files will be displayed *in seriatum*.

1.8 Renaming and Deleting

Three commands every beginner should know are: `cp`,`rm` and `mv`. These are, respectively, copy, remove and move(rename). Here are their usages

```
cp oldFile newFile
rm garbageFile(s)
mv oldFile newFile
```

Warning! Pay heed before you proceed! To *clobber* a file means to unlink it from your file system. When you clobber a file it is lost and there is virtually no chance you will recover its contents. There is no undelete facility as you might find on other computing systems you have used.

If you remove a file it is clobbered, and there is no way to get it back without an infinitude of horrid hassle. If you copy or rename onto an existing file, that file is clobbered, and it is gone forever. Always check to see if the file name you are copying or moving to is unoccupied! When in doubt, do an `ls` to look before you leap. All three of these commands have an option `-i`, which warns you before clobbering a file. Using this is a smart precaution.

The first command copies *oldFile* to *newFile*. If *newFile* does not exist, it creates *newFile*; otherwise it will overwrite any existing *newFile*.

Try this at your UNIX prompt: `cp .bash_profile quack`

Notice that the command `cp` has two arguments: the source file and the recipient file. If you executed the last command successfully, you made a copy of your `.bash_profile` file to a file called `quack`.

Next, let's get rid of all the animals in the zoo we had created before. The command `rm` will accept one or more arguments and remove the named files. We can accomplish this in one blow with

```
unix> rm aardvark buffalo cougar dingo elephant
```

Now enter

```
unix> ls -l
```

You will see that `quack`'s size is nonzero because it has a copy of the contents of your `.bash_profile` file in it. The file shown here has size 191. The size is the number of bytes contained in the file; yours may be larger or smaller. You will also see that the menagerie has been sent packing.

```
-rw-rw-r-- 1 morrison morrison     0 Jun  9 10:50 bar
-rw-rw-r-- 1 morrison morrison     0 Jun  9 10:49 foo
-rw-r--r-- 1 morrison morrison   191 Jun  9 11:25 quack
-rw-rw-r-- 1 morrison morrison     0 Jun  9 10:49 stuff
```

Let us now remove the file `stuff`. We are going to use the `-i` option. Enter this at the UNIX prompt.

```
unix> rm -i stuff
```

The system will then ask you if you are sure you want to remove the file. Tell it yes by typing the letter `y`. Be reminded that the `-i` option is also available with `cp` and `mv`. You should you use it to avoid costly mistakes.

Finally, we shall use `mv` This "moves" a file to have a new name. Let's change the name of `quack` to `honk` and back again. To change `quack` to `honk`, proceed as follows.

```
unix> mv quack honk
```

Once you do this, list the files in long format. Then change it back.

Now you know how to copy, move, and create files. You can show them to the screen and you can list all the files you have. So far, we can create files two ways, we can create an empty file with `touch` or copy an existing file to a new file with `cp`.

1.8.1 Everything is a Program

Now let us take a little look under the hood. When you log in, shell is launched. The shell accepts commands you enter at the prompt and sends them to the *kernel*, or operating system, which runs the program. This can cause output to be put to the screen, as in `ls`, or happen without comment, as in `rm`.

Programs that are running in UNIX are called *processes*. Every process has an owner and an integer associated with it called a process ID (`PID`). The user who spawns a process will generally be its owner. You are the owner of all processes you spawn. Many, such as `ls`, last such a short time you never notice them beyond the output they produce; they terminate in a fraction of a second after you enter them. When you log into your host, you actually are launching a program; this is your shell. When the shell terminates, your terminal session will be gone. At the command line, enter `ps` and you will see something like this.

```
unix> ps
PID TTY
10355 pts/1
10356 pts/1
unix>
TIME CMD
00:00:00 bash
00:00:00 ps
```

The `ps` command shows all processes currently running spawned by your shell. On this machine, the shell's (bash) process ID is 10355. By entering `ps aux` at the command line, you can see all processes running on your UNIX server, along with their process IDs and an abundance of other information. Try this at several different times. If you are using a server, you will see processes spawned by other users. You will also see other processes being run by the system to support your machine's operation.

An example of a program that does not finish its work immediately is the program bc. We show a sample bc session here; this application is a simple arbitrary-precision calculator.

```
unix> bc
bc 1.06.94
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006
Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type ‘warranty’.
3+4
7
```

```
4*5
20
2^20
1048576
2^100
1267650600228229401496703205376
quit
```

When you type `bc` at the command prompt, the shell runs the bc program. This program continues to run until you stop it by typing quit. To see `bc`'s process ID, start bc and then type Control-Z to put it to sleep. This interrupts the `bc` process, puts it in the background, and returns you to your shell. Then enter `ps` at the command prompt to see the process ID for your bc session.

```
unix> bc
bc 1.06.94
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006
Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type ‘warranty’.
[1]+
unix> ps
PID
14110
14253
14254
unix>
Stopped
TTY
pts/4
pts/4
pts/4
bc
TIME
00:00:00
00:00:00
00:00:00
CMD
bash
bc
ps
```

Try typing `exit` to log out; you will see something like this.

```
unix> exit
```

```
exit
There are stopped jobs.
unix>
```

Now type `jobs` at the command prompt. You will see this.

```
unix> jobs
[1]+ Stopped
unix>
bc
```

You can end the job `bc` labeled `[1]` by doing the following

```
unix> kill %1
unix> jobs
[1]+ Terminated
unix> jobs
unix>
bc
```

If several jobs are stopped, each will be listed with a number. You can end any you wish to by entering a `kill` command for each job. When you type `jobs` at the command line the first time, it will tell you what jobs it has suspended. After that, you will see a (possibly empty, like here) list of jobs still in the background. Do not dismiss a shell with running jobs; end them to preserve system resources.

You can bring your stopped job into the foreground by entering fg at the command prompt.

Exercises

1. Start up a session of bc and put it into the background using control-Z. Do this for several sessions. Type in some calculations into some of the sessions and see if they reappear when you bring the bc session containing that calculation into the foreground.
2. The bc calculator has variables which allow you to store numbers under a name. These play the role of the symbols described in Chapter 0, but they are limited to storing numbers. Here we show some variables being created and some expressions being evaluated.

   ```
   morrison@ghent:~unix> bc
   bc 1.06.94
   Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006
   Free Software Foundation, Inc.
   ```

```
This is free software with ABSOLUTELY NO WARRANTY.
For details type 'warranty'.
cow = 5
pig = 2
horse = 7
horse + cow
12
horse/pig
3
pig/horse
0
cow^horse
78125
```

Replicate this session. Put it into the background and bring it into the foreground. Were your variables saved? Notice that this calculator does integer arithmetic. The = sign you see is actually assignment, which was discussed in Chapter 0.

3. Look at one of the algorithms for converting a binary number into a decimal number described in Chapter 0. Can you step through the process using `bc` and make it work?

1.9 Editing Files

We can create files with `touch` and use `cp` to copy them. How do we edit text files and place information in them? This is the role of the UNIX text editor, `vi`. The O'Reilly book [6] on it comes highly recommended if you want to become a power user (you do). A second text editor, `emacs` is also available. It is powerful and extensible. Like `vi` it is a serious tool requiring serious learning, and like `vi` there is an O'Reilly book on it, too. You may use emacs instead of `vi` if you wish. Both of these are just tools for creating and editing text files, and both do a great job. You may create or modify any text file with either program. Ubuntu users can also use `gedit` or `gvim`, which have some nice advantages.

A Note for Ubuntu Users Ubuntu by default installs the package `vi-tiny`. We want `vi` with all bells and whistles. To get this, make sure you are connected to the Internet, then type the following command in an terminal window.

```
unix> sudo apt-get install vim
```

You will be asked to enter your password, then it will install the full vi package. The `sudo` command tells Ubuntu you are behaving as a system administrator, so you must enter your password to proceed. It will ask you to confirm you wish

to install, and then it will download the package from the repositories, install, and configure it for you. Ubuntu has lots of programs and packages that are freely available, and you use `sudo apt-get install` to obtain them.

1.9.1 Launching `vi`

To create a new file or open an existing file, type

```
unix> vi someFileName
```

at the UNIX command line. If the file *someFileName* exists, it will be opened; otherwise, it will be created. Now let us open the file `bar` we created with `touch`. You will see this:

```
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"bar" 0L 0C
```

DO NOT TYPE YET! Read ahead so you avoid having a passel of confusing annoying things plaguing you. The tildes on the side indicate empty lines; they are just placeholders that are not a part of the actual file. It is fairly standard for the tildes to be blue. The OL OC "bar" indicates that file `bar` has no lines and no characters. If the file `bar` were not empty, its contents would be displayed, then blue tildes would fill any empty screen lines.

1.9.2 `vi` Modes

Before you hit any keys there is something important to know. The `vi` editor is a moded editor. It has four modes: command mode, visual mode, insert mode, and replace mode. Command mode provides mobility, search/replace, and copy/paste capabilities. Insert mode allows you to insert characters using the keyboard. Visual mode provides the ability to select text using the keyboard, and then change or copy it. Replace mode overwrites existing text with new text. When you first open a file with `vi`, you will be in command mode.

We will begin by learning how to get into insert mode. You always begin a `vi` session in command mode. There are lots of ways to get into insert mode. Here are a six basic ones that are most often used.

keystroke	Action
i	insert characters before the cursor
I	insert characters at the beginning of the line
a	append characters after the cursor
A	append characters at the end of the line
o	open a new line below the cursor
O	open a new line above the cursor

Here is an easy way to remember. What happens if you accidentally step on your cat's tail? He says IAO!!!

There is **one** way to get out of insert mode. You do this by hitting the escape (ESC) key. Let's now try this out. Go into your file `bar` and hit the `i` key to enter text. Type some text. Then hit ESC. To save your current effort type this anywhere:

```
:w
```

This will write the file; a message will appear at the bottom of the window indicating this has happened. Do not panic; that message is **not** a part of the file that is saved. To quit, type

```
:q
```

this will quit you out of the file. You can type

```
:wq
```

to write and quit. The command `:qw` does not work for obvious reasons. You have just done a simple `vi` session. You can reopen the file `bar` by typing

```
unix> vi bar
```

at the UNIX command line; the contents you last saved will be re-displayed. You should take a few minutes to try all of the ways of getting into insert mode. Change the file and save it. Quit and display it to the screen with `cat` and `more`.

At first, `vi` will seem clunky and awkward. However, as you ascend the learning curve, you will see that `vi` is blazingly fast and very efficient. One of its great strengths is the ability to search for and replace text. As your skill grows with it, you will see it is an amazing productivity tool.

A Reassuring Note If you are in command mode and hit ESC, your computer will just beep at you. This is its way of letting you know you were already in command mode. Nothing additional happens. If you are unsure what mode you are in, hit ESC and you will be back in command mode, no matter what. You can hit ESC and relax.

The figure below will help you to remember the structure of vi. When you first start editing a file, you enter in in command mode. Typing i, a, o, I, A or O all put you into insert mode. You can also see in the diagram how to get out of insert mode by typing ESC.

```
|------------------------------------------------------------
|           command mode                                     |
```

```
|     ---------------i, a, o, I, A, O to get in-----------    |
|     |    insert mode:                                  |   |
|     |    insert characters                             |   |
|     |    paste in text with GUI                        |   |
|     ----------------------------------ESC to get out---    |
|          search, replace                                    |
|          copy with yy, paste with  p                        |
|          delete lines with dd                               |
|          all "colon commands" (commands that start with :)  |
|---------------------------------- --------------------------
```

Let's go back in our file now and learn some more useful commands. We will look at command mode commands now.

Sometimes, line numbers will be helpful; these are especially useful when you program. To see them, you get into command mode and type the *colon command* `:set number`. Do this and watch them appear. Now type `:set nonu` or `:set nonumber` and watch them disappear. Line numbers are not a part of the file; however, they are a helpful convenience.

Here are some useful command mode mobility features. Experiment with them in a file.

Command	Action
: *lineNumber*	Go to indicated line number.
^	Go to the beginning of the current line.
$	Go to the end of the current line.
G	Go to the end of the file.
gg	Go to the beginning of the file; note that :1 also works.

These colon commands in this table will allow you to alter your editing environment. The last two are useful editing tricks that are sometimes quite convenient. Open a file and try them all.

Command	Action
:set number	display line numbers
:set nonu	get rid of line numbers
:set autoindent	This causes `vi` to autoindent.
:set noautoindent	This causes `vi` to turn off autoindent.
r (then a character)	replace character under cursor
~	change case upper → ;lower or lower → upper

1.9.3 Cut and Paste

The vi editor has a a space of memory called the *unstable buffer*, which we nickname Mabel. Mabel provides a temporary cache for holding things while

we are editing and she is very helpful for doing quick copy-paste jobs.

This buffer is unstable because it loses its contents every time new text is placed in it. Do not use it to store things for a long time; instead write those things to files and retrieve them later. You will learn several ways to do this.

We show here a table with some cut, copy, and paste commands you will find helpful.

yy	Yank line to Mabel
dd	Delete line starting at the cursor; this cuts to Mabel
dw	Delete word; this cuts to Mabel
cw	Delete word, then enter insert mode(change word) The changed word is cut to Mabel.
p	Paste Mabel's contents at the cursor.
D	Cut line at cursor; this cuts the stricken text to Mabel
C	Cut line at cursor and enter insert mode; this cuts the stricken text to Mabel

All of these commands can be preceded by a number, and they will happen that number of times. For example typing `10yy` in command mode will yank ten lines, starting at the cursor, to Mabel. Since so many of these commands place new text in Mabel, you should know that if you copy or cut to Mabel and intend to use the text, paste it right away. You should open a file and experiment with these. Spend some time fooling around with this mechanism; you will make some delightful discoveries, as well as dolorous ones.

1.9.4 Using External Files

You can select a range of line numbers before each of these commands, or select in visual mode and use these commands.

`:w fileName`	Write a copy of the entire the file `fileName`
`:w! fileName`	Write selection to existing file `fileName`, and clobber it.
`:w >> fileName`	Append selection to file `fileName`.
`:r fileName`	Read in file `fileName` starting at the cursor

For example

```
:20,25 w foo.txt
```

will write lines 20-25 to the file `foo.txt`. If you want to write the entire file, omit the line numbers and that will happen. If you want to write from line 20 to the end of the file, the usage is as follows.

```
:20,$ w foo.txt
```

Note the use of `$` to mean "end of file." When you learn about visual mode (just ahead), you can use these command to act on things you select in visual mode as well.

Housekeeping Tip If you use this facility, adopt a naming convention for these files you create on a short-term basis. When you are done editing, get rid of them or they become a choking kudzu and a source of confusion in your file system. Use names such as `buf`, `buf1`, etc as a signal to yourself that these files quickly outlive their usefulness and can be chucked.

1.9.5 Search and Replace

Finally we shall look at search capabilities. These all belong to command mode. Enter

```
/someString
```

in command mode and `vi` will seek out the first instance of that string in the file or tell you it is not found. Type an `n` to find the next instance. Type `N` to reverse direction. You can enter

```
?someString
```

to search for someString backwards from the cursor. Type `n` to find the previous instance, and `N` to revese direction. Your machine may be configured to highlight every instance of the string you searched for. If you find this feature annoying, you can deactivate it with

```
:set nohlsearch
```

Now let us look at search and replace. This is done by a colon command having this form.

```
:s/old/new/(g|c|i)
```

The `s` means substitute; this substitutes `old` for `new`. The three *flags* at the end specify how the substitution should work By default, substitutions are confined to the cursor line, but you can control the scope of a substitution in these two ways.

Bound	Scope
`a,b s/old/new/(g|c|i)`	Perform the substitution on lines `a` through `b`, inclusive.
`a, $`	Perform the substitution on line `a` until the bottom of the file.

Here is how the flags work. At the end you can append any of `g`, `c`, or `i`. Here is a decoder ring.

`c`	Check after each substitution to see if you want ot replace.
`g`	Replaces all instances on each line. By default, only the first one is replaced.
`i`	Replace `old` case-insensitive.

You will also learn how to control the scope of substitutions in visual mode below. That method is extremely nice and quite simple to learn.

1.10 Visual Mode

The third mode of the `vi` editor, visual mode is actually three modes in one: line mode, character mode, and block mode. To enter line mode from command mode, hit `V`; to enter character mode hit `v`, and to enter block mode, hit Control-v. You can exit any of these by hitting the ESC key; this places you back in command mode. Visual mode has one purpose: it allows you to select text using keyboard commands; you may then perform various operations on these selections. First, let us see the selection mechanism at work.

Go into a file and position your cursor in the middle of a line. Hit `v` to enter visual character mode. Now use the arrow keys; notice how the selected text changes in response to arrow key movement. Try entering gg and G and see what happens. Hit ESC to finish. Now enter visual mode and use the / search facility to search up something on the page. What happens? Search backward and try that too.

Now enter visual line mode by hitting `V`; now try the keystrokes we just indicated and see how the selection behaves. This mode only selects whole lines.

Finally if you enter Control-V and you enter visual block mode, you can select a rectangular block of text from the screen by using the keyboard.

Now let's see what you can do with these selections. First let us look at character and line mode, as block mode behaves a little differently. You can delete the selected text by hitting `d`. You can yank it into Mabel by hitting `y`. Upon typing either command, you will be put back into command mode. Once any text is yanked into Mabel, you can paste it with `p` as you would any other

text yanked there. If you hit `c`, the selection will be deleted and you will be in insert mode so you can change the text.

In block mode, things are a little different. If you hit `d`, the selected block will be deleted, and the lines containing it shortened. The stricken text is cut to Mabel. If you hit `y`, the block will be yanked just as in any other visual mode, and its line structure will be preserved. If you hit `c`, and enter text, the same change will be made on all line selected provided you do not hit the ENTER key. If you do, the change will only be carried out on the first line. You can insert text rather than change by hitting I, entering your text, and then hitting ESC. If the text you enter has no newline in it, the same text will be added to each line; if it has a newline, only the first line is changed.

If you hit `r` then any character in any visual mode, all selected characters are changed to that character.

Here is a very common use for character or line visual mode. Suppose you are editing a document and the lines end in very jagged fashion. This sort of thing will commonly happen when maintaining a web or if you are editing a LaTeXdocument such as this one, where the page that is subjected to repeated edits. Use visual mode to select the affected paragraphs and hit gq (think Gentleman's Quarterly) and your paragraphs will be tidied up.

You can also do search-and-replace using visual mode to select the text to be acted upon. Simply select the text in visual mode. Then hit

```
: s/outText/inText/g
```

to perform the substitution in the selected text. For example if you select text in visual mode and change every w to a v, you will see this.

```
:'<,'>s/w/v/g
```

The `<,'>` is a quirky way of indicating you are doing a visual-mode search-replace operation.

1.10.1 Replace Mode

In `vi` if you hit `r` then a character, the character under the cursor is replaced with the character you it. If you hit `R`, you are in *replace mode*, and any test you type "overruns" existing text. Experiment with this in a file you don't care about.

Replace mode is fabulous for making ASCII art such as this.

```
< Galactophagy >
 --------------
```

```
\   ^__^
 \  (oo)\_______
    (__)\       )\/\
        ||----w |
        ||     ||
```

You should play around with this. Do a Google search to learn about ASCII art.

1.11 Copy Paste from a GUI

You can copy and paste with the mouse in a window or between windows. The way you do it varies by OS so we will quickly discuss each. If you are pasting into a file you are editing with `vi`, it is a smart idea to use the colon command `:set paste`. This will prevent the "mad spraying" of text. For certain types of files, this turns off automatic indentation or formatting. You can use `:set nopaste` to turn off the paste mode.

Windoze If you are copying *from* a Windoze application into a terminal window, select the text you want to copy and use control-C in the usual way. This places the text in your Windoze system clipboard. Now go into your terminal window and get into insert mode where you want to paste. It is also wise, in command mode, to enter `:set paste`. Right-click to paste the contents of your system clipboard into the terminal window. Many of you will say, "Why did the beginning of the text I copied get cut off or why didn't it appear at all?" This will occur if you are not in insert mode when you paste. It is important to be in insert mode before pasting to avoid unpleasant surprises. If this happens, hit ESC then `u` in command mode. The `u` command undoes the last `vi` command. Then you can take a fresh run at it.

If you are copying from a terminal window, select the text you wish to copy; PuTTY will place the text in your system clipboard. Then go into the window in which you wish to paste it. If the window is another terminal, get into insert mode and right-click on the mouse. If it's a Windoze app, use control-V as you usually do.

Mac Use apple-c to copy and apple-v to paste to or from a terminal window, just as you would with any other mac app. Mac gets this right.

Linux If you use a Linux box, use control-shift-C for copying in terminal windows and control-shift-V for pasting to terminal windows.

A Reprise: A Warning About `autoindent` and `paste` Before pasting with the mouse make sure you have autoindent turned off. Otherwise, your text will "go mad and spray everywhere," especially if you are copying a large block of text with indents in it. You can turn autoindent on with `:set autoindent` and off with `:set noautoindent`. This feature can be convenient when editing certain types of files. You can use the command `:set paste` to turn off all smart indentation; when finished use `:set nopaste` to set things back to their original state.

A Warning abut Line Numbers If you copy-paste to a GUI, line numbers will get copied. To prevent this from happening, use the colon command `:set nonu` before copying.

Experiment with these new techniques in some files. Deliberately make mistakes and see what happens. Then when you are editing files, you will know what to expect and how to recover.

There are a lot of excellent tutorials on `vi` on the web; avail yourself of these to learn more. Remember the most important thing: you never stop learning `vi`! Here are some useful `vi` resources on the web.

- The site [3] for is complete, organized and well-written. You can download the whole shebang in a PDF. Read this in little bits and try a few new tricks at a time.
- The site [8], `vi` for Smarties will introduce you to `vi` with a bit of churlish sneery attitude. It's pretty cool. And it's sneery like the author of this august volume.
- The link `ftp://ftp.vim.org/pub/vim/doc/book/vimbook-OPL.pdf` will download The Vim book for you. It is a very comprehensive guide, and it has excellent coverage on the visual mode.

1.11.1 Permissions

Now we will see how you can use permissions to control the visibility of your files on the system. You are the owner of your home directory and all directories and files it contains. This is your "subtree" of the system's directories belonging to you. You may grant, revoke or configure permissions for all the files and directories you own as you wish. UNIX was designed with the fundamental idea that your data are your property, and you can control what others see of them.

There are three layers of permission: you, your group, and others. You is letter `u`, your group is letter `g` and others is letter `o`. There are three types of permission for each of these: read, write and execute. Read means that level can read the file, write means that level can execute the file, and execute means that

level can execute the file. In the example above, the file `bar` has the *permission string*

```
-rw-rw-r--
```

which means the following.

- You can read or write to the file. You cannot execute.
- Your group can read or write but not execute.
- The world can read this file but neither write nor execute.

For the user to execute this file, use the `chmod` command as follows

```
unix> chmod u+x bar
```

The u(ser's, that's you) permission changed to allow the user to execute the file.

If you do not want the world to see this file you could enter

```
unix> chmod o-r bar
```

and revoke permission for the world to see the file `bar`. Since you are the owner of the file, you have this right. In general you can do

```
$  chmod (u or g or o)(+ or -)(r or w or x) fileName
```

to manage permissions. You can omit the u, g or o and the permission will be added or deleted for all three categories. In the next subsection, we discuss the octal representation of the permissions string. This will allow you to change all three levels of permissions at once quickly and easily.

1.12 The Octal Representation

There is also a numerical representation for permissions. This representation is a three-digit octal (base 8) number. Each permission has a number value as follows.

- The permission `r` has value 4.
- The permission `w` has value 2.
- The permission `x` has value 1.
- Lack of any permission has has value 0.

We show how to translate a string in this example.

```
-rw-r--r--
  6  4  4
```

The only way to get a sum of 6 from 1,2 and 4 is 4 + 2. therefore 6 is read-write permission. The string translates into three digits 0-7; this file has 644 permissions. It is a simple exercise to look at all the digits 0-7 and see what permissions they convey.

We show some more examples of `chmod` at work. Look at how the permissions change in response to the chmod commands. Suppose we are a directory containing one file named `empty`, which has permission string extt -rw-r–r–, or `644`. We begin by revoking the read permission from others.

```
unix> chmod o-r empty
```

We now list the files in the directory

```
unix> ls -l
total 0
-rwxr----- 1 morrison morrison 0 2008-08-26 10:52 empty
unix> ls
empty
```

We can now restore the original permissions all at once by using the octal number representation for our permissions.

```
 chmod 644 empty
unix> ls -l
total 0
-rw-r--r-- 1 morrison morrison 0 2008-08-26 10:52 empty
```

Notice what happens when we try to use a 9 for a permission string.

```
unix> chmod 955 empty
chmod: invalid mode: ‘955’
Try ‘chmod --help’ for more information.
```

Try typing the `chmod -help` command at your prompt and it will show you some useful information about the chmod command. Almost all UNIX commands have this help feature.

Directories must have executable permissions, or they cannot be entered, and their contents are invisible. Here we use the `-a` option on `ls`. Notice that the

current working directory and the directory above it have execute permissions at all levels. Try revoking execute permissions from one of your directories and attempt to enter it with `cd`; you will get a `Permission Denied` nastygram from the operating system.

```
unix> ls -al
total 20
drwxr-xr-x 2 morrison faculty 4096 2008-10-17 11:51 .
drwx--x--x 9 morrison faculty 4096 2008-10-16 08:39 ..
-rw-r--r-- 1 morrison faculty 0 2008-10-17 11:51 empty
unix>
```

Here we shall do this so you can bear witness

```
unix> mkdir fake
unix> chmod u-x fake
unix> cd fake
bash: cd: fake: Permission denied
unix>
```

Assigning 600 permissions to a file is a way to prevent anyone but yourself from seeing or modifying that file. It is a quick and useful way of hiding things from public view. Later, when you create a web page, you can use this command to hide files in your website that you do not want to be visible.

1.13 The Man

The command `man` is your friend. Type man then your favorite UNIX command to have its inner secrets exposed! For example, at the UNIX prompt, enter

```
unix> man cat
```

This brings up the man(ual) page for the command `cat`. A complete list of options is furnished. Notice that some of these have the `-`, or long form.

```
CAT(1)                User Commands                    CAT(1)

NAME
       cat - concatenate files and print on the standard output

SYNOPSIS
       cat [OPTION] [FILE]...
```

```
DESCRIPTION
       Concatenate FILE(s), or standard input, to standard output.

       -A, --show-all
              equivalent to -vET

       -b, --number-nonblank
              number nonblank output lines

       -e     equivalent to -vE

       -E, --show-ends
              display $ at end of each line

       -n, --number
              number all output lines

       -s, --squeeze-blank
              never more than one single blank line

       -t     equivalent to -vT

       -T, --show-tabs
              display TAB characters as ^I

       -u     (ignored)

       -v, --show-nonprinting
              use ^ and M- notation, except for LFD and TAB

       --help display this help and exit

       --version
              output version information and exit

       With no FILE, or when FILE is -, read standard input.

EXAMPLES
       cat f  g
              Output f's contents, then standard input,
              then g's contents.

       cat    Copy standard input to standard output.

AUTHOR
       Written by Torbjorn Granlund and Richard M. Stallman.
```

```
REPORTING BUGS
       Report bugs to <bug-coreutils@gnu.org>.

COPYRIGHT
       Copyright Â© 2006
       Free Software Foundation, Inc.
       This  is  free software.  You may redistribute
       copies of it under the terms of the GNU General
       Public License <http://www.gnu.org/licenses/gpl.html>.
       There is NO WARRANTY, to the extent permitted by law.
SEE ALSO
       The full documentation for cat is maintained
       as a Texinfo manual.  If the info and cat
       programs are  properly installed at your site,
       the command info cat should give you access
       to the complete manual.

cat 5.97     August 2006                        CAT(1)
```

You can see here that even humble cat has some options to enhance its usefulness. Here is `cat` at work on a file named `trap.py`.

```
unix> cat trap.py
def trap(a, b, n, f):
    a = float(a)
    b = float(b)
    h = (b - a)/n
    list = map(lambda x: a + h*x, range(0,n+1))
    tot = .5*(f(a) + f(b))
    tot += sum(map(f, list[1:n]))
    tot *= h
return tot
def f(x):
    return x*x
print trap(0,1,10,f)
print trap(1,2,100,f)
```

Using the -n option causes the output to have line numbers.

```
 cat -n trap.py
    1 def trap(a, b, n, f):
    2     a = float(a)
    3     b = float(b)
    4     h = (b - a)/n
```

```
    5     list = map(lambda x: a + h*x, range(0,n+1))
    6     tot = .5*(f(a) + f(b))
    7     tot += sum(map(f, list[1:n]))
    8     tot *= h
    9     return tot
   10 def f(x):
   11   return x*x
   12 print trap(0,1,10,f)
   13 print trap(1,2,100,f)
unix>
```

View the manual pages on commands such as `rm`, `ls` `chmod` and `cp` to learn more about each command. Experiment with the options you see there on some junky files you create and do not care about losing.

Exercises

1. Use the `man` command to learn about the UNIX commands `more` and `less`. You will see here, that in fact, less is more!
2. Use the `man` command to learn about the UNIX commands `head` and `tail`. Can you create a recipe to get the first and last lines of a file?
3. What does the `ls -R` command do?

1.14 Lights, Camera, Action! Where's the Script?

Sometimes you will find yourself doing certain chores repeatedly. An intelligent question to ask is, "Can't I just save this list of commands I keep typing over and over again in a file?"

Happily, the answer to this is "yes." It's called writing a *shell script*. In its simplest form, a shell script is just a list of UNIX commands in a file. We will see how to make one of these and run it. Begin by creating this file, `greet.sh`.

```
#!/bin/bash
echo Hello, $LOGNAME!
echo Here is the calendar for this month:
cal
```

Type commands you see in this file into your shell. You will see this.

```
unix> echo Hello, $LOGNAME!
Hello, morrison!
unix> echo Here is the calendar for this month:
```

```
Here is the calendar for this month:
unix> cal
    January 2020
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

Now give this file execute perimssions like so.

```
unix> chmod +x greet.sh
```

Now run it (note the slash-dot).

```
unix> ./greet.sh
Hello, morrison!
Here is the calendar for this month:
    January 2020
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

We can make this process even better. If you don't have one, create a directory named `bin` in your home directory. Then open the dotfile `.bash_profile` and add this line to it.

```
export PATH=$PATH:"/Users/morrison/bin"
```

Replace the `/Users/morrison` with the path to your home directory. Put your shiny new script into this directory. Then you don't need the slash-dot any more unless you are in the `bin` directory. This has the benefit of allowing you to run the script from anywhere in the file system.

1.15 Redirection of Standard Output and Standard Input

UNIX treats everything in your system as a file; this includes all devices such as printers, the screen, and the keyboard. Things put to the screen are generally

put to one of two files, `stdout`, or *standard output* and `stderr`, or standard error. You will see that it is veryeasy to redirect standard output to a another file.

The keyboard, by default, is represented by the file `stdin`, or *standard input*. It is also possible to redirect standard input and take standard input from a file.

UNIX filters, such as `cat` and `more` have as their default input `stdin` and as output `stdout`. This section will show you how to redirect these to files.

Sometimes a UNIX command or a program puts a large quantity of text to the screen; redirection allows you to capture the results into a file. You can open this file with vi, search it, or edit it. The examples here are based on the files `animalNoises.txt`; make them and follow along.

```
miao
bleat
moo
```

and `physics.txt`

```
snape
benettron
stephan
```

First we show how `cat` puts files to `stdout`.

```
unix> cat animalNoises.txt physics.txt
miao
bleat
moo
snape
benettron
stephan
```

Now let us capture this critical information into the file `stuff.txt` by redirecting `stdout`. We then use `cat` to display the resulting file to `stdout`.

```
unix> cat animalNoises.txt physics.txt > stuff.txt
unix> cat stuff.txt
miao
bleat
moo
snape
benettron
stephan
```

The `cat` command has a second guise. It accepts a file name as an argument, but it will also accept standard input; this is no surprise since `stdin` is treated as a file. At the UNIX command line enter

```
unix> cat
```

The `cat` program is now running and it awaits word from `stdin`. Enter some text and then hit the enter key; cat echoes back the text you type in. To finish, hit control-d (end-of-file).

```
unix> cat
me too
me too
ditto
ditto
unix>
```

The control-d puts no response to the screen. You can also put a file to the screen with

```
unix> cat < someFile
```

Here, the file someFile becomes `stdin` for the cat command. This phenomenon is shown in the man page for cat. Under the description of the command it says, " Concatenate FILE(s), or standard input, to standard output."

Let us now come back to `stdout`. Next create a new file named `sheck.txt` with these contents.

```
roach
stag beetle
tachnid wasp
```

Were we to invoke the command

```
unix> cat animalNoises.txt physics.txt > sheck.txt
```

we would clobber the file `sheck.txt` and lose its valuable contents. This may be our intent; if so very well. If we want to add new information to the end of the file we use the `>>` *append operator* to append it to the end of the receiving file. If we do this

```
unix> cat animalNoises.txt physics.txt >> sheck.txt
```

we get the following result if we use the original file `sheck.txt`.

```
unix> cat sheck.txt
roach
stag beetle
tachnid wasp
miao
bleat
moo
snape
benettron
stephan
```

The >> redirection operator will automatically create a file for you if the file to which you are redirecting does not already exist.

1.16 More Filters

It is very common to want to use `stdout` from one command to be `stdin` for another command. This will grant us the ability to chain the actions of the existing filters we have `cat`, `more` and `less` with some new filters to do a wide variety of tasks To achieve this tie, we use a device called a *pipe*. Pipes allow you to chain the action of various UNIX commands. We shall add to our palette of UNIX commands to give ourselves a larger and more interesting collection of examples. These commands are extremely useful for manipulating files of data.

1.16.1 The `sort` filter

Bring up the `man` page for the command `sort`. This command accepts a file (or `stdin`) and it sorts the lines in the file.

This begs the question: how does it sort? It sorts alphabetically in a case-insensitive manner, and it "alphabetizes" non-alphabetical characters by ASCII value. The sort command several four helpful options.

`-b`	`-ignore-leading-blanks`	ignores leading blanks
`-d`	`-dictionary-order`	pays heed to alphanumeric characters and blanks and ignores other characters
`-f`	`-ignore-cases`	ignores case
`-r`	`-reverse`	reverses comparisons

Here we put the command to work with `stdin`; use a control-d on its own line to get the prettiest format. Here we put the items `moose`, `jaguar`, `cat`

and `katydid` each on its own line into `stdin`. Without comment, a sorted list is produced.

```
unix> sort -f
moose
jaguar
cat
katydid        (now hit control-d)
cat
jaguar
katydid
moose
unix>
```

You should try various lists with different options on the sort command to see how it works for yourself. You can also run `sort` on a file and send a sorted copy of the file to `stdout`. Of course, you can redirect this result into a file using > or ».

1.16.2 The Filters `head`, `tail`, and `uniq`

The commands `head` and `tail` put the top or bottom of a file to `stdout`; the default amount is 10 lines. To show the first 5 lines of the file foo.txt, enter the following at the UNIX command line.

```
unix> head -5 foo.txt
```

You can do exactly the same thing with tail with an entirely predictable result. The command `uniq` weeds out consecutive duplicate lines in a file, leaving only the first copy in place. These three commands have many useful options; explore them in the man pages.

1.16.3 The `grep` Filter

This command is incredibly powerful; here we will just scratch the surface of its protean powers. You can search and filter files using `grep`; it can be used to search for needles in haystacks. In its most basic form grep will inspect a file line-by-line and put all lines to `stdout` containing a specified string. Here is a sample session.

```
unix> grep gry /usr/share/dict/words
angry
hungry
unix>
```

The file `/usr/share/dict/words` is a dictionary file containing a list of words, one word to a line in (mostly) lower-case characters. Here we are searching the dictionary for all lines containing the character sequence gry; the result is the two words angry and hungry. There are options `-f` and `-ignore-case` to ignore the case of alphabetical characters.

1.16.4 Serving up Delicious Data Piping Hot

Pipes allow you to feed `stdout` from one command into `stdin` to another without creating any temporary files yourself. Pipes can be used along with redirection of `stdin` and `stdout` to accomplish a huge array of text-processing chores.

Now let us do a practical example. Suppose we want to print the first 5 lines alphabetically in a file named `sampleFile.txt`. We know that sort will sort the file asciicographically; we will use the `-f` option to ignore case. The command `head -5` will print the first five lines of a file passed it or the first five lines of `stdin`. So, what we want to do is sort the file ignoring case, and pass the result to head -5 to print out the top five lines. You join two processes with a pipe; it is represented by the symbol | , which is found by hitting the shift key and the key just above the enter key on a standard keyboard. Our command would be

```
unix> sort -i sampleFile.txt | head -5
```

The pipe performs two tasks. It redirects the output of `sort -f` into a temporary buffer and then it feeds the contents of the buffer as standard input to `head -5`. The result: the first five lines in the alphabet in the file `sampleFile.txt` are put to `stdout`.

Suppose you wanted to save the results in a file named `results.txt`. To do this, redirect `stdout` as follows

```
unix> (sort -i sampleFile.txt | head -5) > results.txt
```

Note the use of defensive parentheses to make our intent explicit. We want the five lines prepared, then stored in the file `results.txt`.

Programming Exercises Here are two more filters, `wc` and a command `echo`. You will use the `man` pages to determine their action and to use them to solve the problems below.

1. Tell how to put the text "Cowabunga, Turtle soup!" to `stdout`.
2. Tell how to get the text "This is written in magic ink" into a text file without using a text editor of any kind.

3. The `ls` command has an option `-R`, for "list files recursively." This lists all of the sub-directories and all of their contents within the directory being listed. Use this command along with grep to find a file containing a specified string in a file path.
4. Put a list of names in a file in lastName, firstName format. Put them in any old order and put in duplicates. Use pipes to eliminate duplicates in this file and sort the names in alphabetical order.
5. Find the word in the system dictionary occupying line 10000.
6. How do you count all of the words in the system dictionary containing the letter `x`?
7. Find all words in the system dictionary occupying lines 50000-50500.
8. Tell how, in one line, to take the result of the previous exercise, place it in reverse alphabetical order and store in in a file named `myWords.txt`.

Chapter 2

Python

Beginning with Python

2.0 Running Python

You can visit [1] for installing the entire Python apparatus. This is available for the MacOSX, Windoze, and Linux platforms. This book will be an introduction to both plain-vanilla Python and the SciPy stack, which is an important tool for data visualization and analysis.

You will also want a plain text editor (not a word processor). If you are a UNIX/Mac user, there is good old `vim`. Other excellent choices for all platforms include include SublimeText, Atom, and VSCode. Notepad++ is an excellent choice for Windoze users. All are free and all do a great job, and can be easily found with a little Googling.

Throughout this book, we will use the symbol `unix>` to represent your system prompt, whether it is a Windoze cmd window or a UNIX terminal window. You can start Python at the command line like so.

```
unix> python3
Python 3.7.4 (default, Aug 13 2019, 15:17:50)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

What you see is the Python prompt. To exit, type control-D or `quit()`. To make a program, you can use your favorite text editor (vi, gedit, Atom, Sublime Text, etc). We will go into this in more detail later.

A useful accompaniment to this chapter is a series [4] of videos by the zany Net Ninjas. Corey Schaefer's YouTube channel [7] contains a wealth of information on Python.

2.1 Scalar Types

We begin by looking at the simplest types Python's type system. We will explore this via the interactive Python prompt. All of these types are immutible objects. The most basic type is the integer type, `int`. Integers have the expected behavior in the presence of arithmetic operators. We demonstrate the basic operators here.

You also see how comments are done; everything on a line after a # is a comment.

```
>>> 42 + 58          #addition
100
>>> 21-57            #subtraction
-36
>>> 66*5             #multiplication
330
>>> 44/3             #division
14.666666666666666
>>> 44//3            #integer division
14
>>> 365%7            #mod
1
>>> 2**20            #exponentiation
1048576
```

Integers do not overflow. This type admits integers of arbitrary precision, subject to the (gargantuan) limits on memory.

```
>>> 2**1000
10715086071862673209484250490600018105614048117055336074437503883703510511249361
22493198378815695858127594672917553146825187145285692314043598457757469857480393
45677748242309854210746050623711418779541821530464749835819412673987675591655439
46077062914571196477686542167660429831652624386837205668069376
>>>
```

Python 2 Note Note that division is integer division in Python 2. In Python 3, division of integers returns a floating-point number that conforms to the IEEE754 standard; you can learn about it in [10]. Python 2 automatically

converts huge integers into a separate type called `long`. To do integer division in Python 3, use the // operator.

Programming Exercises

1. Can you compute 2^{10000}?
2. There is an infix binary operator on the integers, ^. Can you experiment with this and figure out what it does? *Hint.* Look at the binary expansions of numbers you operate on. Use the built-in Python function `bin` to compute binary expansions for the integers you fiddle with. We show `bin` in action here. A `0b` prefix means, "this is a binary number."

   ```
   >>> bin(42)
   '0b101010'
   >>> 0b1110
   14
   ```

3. Look up `bin`, `hex`, and `oct` in [5] and read about them. Experiment with them and see their aciton.
4. There is an infix binary operator on the integers, &. Can you experiment with this and figure out what it does? Look at binary expansions for a clue.
5. There is an infix binary operator on the integers, |. Can you experiment with around with this and figure out what it does?

Since we have seen a floating point number, let us formally introduce those. This type is known as `float`. You will see few surprises.

```
>>> 2.0 + 3
5.0
>>> 6.02e23 * 100
6.02e+25
>>> 5.3 - 6.1
-0.7999999999999998
>>> 3/666
0.0045045045045045045
>>> 1.0001**10000
2.7181459268249255
```

Notice that when you add an integer and a floating point number, that the integer gets converted into a floating point number. As we said before, Python's floating point numbers are IEEE 754 double-precision 64 bit numbers.

Programming Exercise: Exploring Numbers This is a little puzzler project in which you do some scientific calculations. You are allowed *only* these facts. Remember in the metric system, centi- means 1/100, milli- means 1/1000 and kilo- means 1000. Use Python's interactive mode to make this happen.

- 1 in = 2.54 cm (length)
- 1 liter = 33.8 fluid ounces (volume) = 1000 cm3
- 1 mile = 5280 ft (length)
- 1 foot = 12 in (length)
- 1 hour = 60 min (time)
- 1 min = 60 sec (time)
- 1 yr = 365.24 days (time)
- 1 kg = 2.204 lbs
- 1 hr = 60 min, 1 min = 60 s, 1 day = 24 hr(time)
- 1 ton = 2000 lbs
- 1 acre = 1/640 mi^2

Now use these facts to answer these questions.

1. Given that light travels at 2.9979e8 (that's 2.9979*108 in scientific notation) meters per second, figure out how fast light moves in miles per second. Then convert this to miles per hour.
2. Tell the time it take for light to go from the sun to the earth if the mean distance of the sun to the earth is 93.0 million miles.
3. Use the fact that a liter of water weighs one kilogram and that one gallon of water weighs 8.33 lbs to determine the number of cubic inches in a gallon and the number of pounds in a cubic foot of water.
4. Let us assume that humans weigh an average of 140 lbs and that humans have about the same density as water. If the population of the earth is 7.0 billion humans, estimate the total volume of humanity in cubic miles. Do you find this counterintuitive?
5. An *acre-foot* of water is enough water to cover one acre one foot deep. How many gallons of water are in an acre-foot? What does that water weigh in tons?

Programming Exercises: A peek ahead Note: you will see we are using the new Python3 style of f-string, instead of the old "%" and `format` methods. Eventually `%` that will be deprecated. If you are going to write new code, use the newer methods.

1. Enter these things in an interactive session.

   ```
   f"{3/7:.1f}"
   f"{3/7:.2f}"
   f"{3/7:.8f}"
   ```

   What kind of object is returned? What do you see? What does the number to the right of the point do? If it's not clear from the examples shown, try a few more.

2. Enter this in an interactive session.

   ```
   f"{3/7:.8e}"
   ```

   Experiment with different numbers. What is nice about this?

3. Now test-drive this.

   ```
   f"{2:.5f}"
   ```

4. What happens if you put an integer in front of the point? What if that integer is negative? Experiment and determine. You will learn some nifty stuff about formatting numbers.

The most commonly-used type in any programming languages is the string; you just got a preview of strings in the exercises above. Strings are used in Python to hold globs of text. Strings support the infix binary operator + and an infix operator * that takes an integer and a string as arguments.

```
>>> "Happy " + "Happy"
'Happy Happy'
>>> "*"*50
'**************************************************'
```

The * operator requires an integer and a string and it returns a string that repeats the string operand the integer number of times. If the integer is 0 or negative, an empty string is returned.

Python has a boolean type, `bool` which has two elements `True` and `False`. There are two infix binary operators on Booleans; they are `and` and `or`. If `P` and `Q` are predicates (Boolean-valued expressions), then `P and Q` is true precisely when both `P` and `Q` are true. The predicate `P or Q` is true precisely when at least one of `P` and `Q` is true.

There is also a unary prefix operator `not` which reverses the truth-value of its operand. The order of operations in Boolean expressions is `not`, `and`, then the lowest is `or`. As you would expect, you can use parentheses to override this order of operations; when in doubt avail yourself of them.

Python has the standard relational operators. They all return a Boolean value, as you would expect. We show them here in a table.

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to

For number types they do numerical comparison. For strings, they compare asciicographically. Python has an additional infix binary operator, `is`, which tests for equality of identity.

All of the scalar types we have seen so far are immutable objects. Once created they cannot be changed in-place. You might ask, "why this immutability?" Immutability allows Python to pool objects, which enables the recycling of commonly used objects without wasting a lot of time creating and deallocating them. In fact, when Python runs, it pre-loads small integers into its memory

For evidence of this take note of this little Python session.

```
>>> hex(id(0))
'0x10b242470'
```

We see the virtual address where Python is storing zero. Now watch this.

```
>>> id(1) - id(0)
32
```

Notice that 1 is stored 32 bits away from zero. Now observe this.

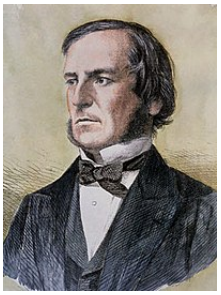
```
>>> id(256) - id(0)
8192
>>> id(260) - id(0)
3130752
```

We see that 0-256 are in the little integer pool. Once you get to 257 we surnmise things get stored elsewhere because of what we see here.

```
>>> hex(id(256)
... )
'0x10b244470'
>>> hex(id(257))
'0x10b53edf0'
```

You are encouraged to do a little spelunking and see where the negative integers go.

Programming and Writing Exercise: PEMDAS for George George Boole was a pioneer of modern logic, as well as a versatile mathematician who studied differential equations. He is the source of the name `boolean` you see that refers to a calculus of true/false values.



You are to perform experimments in the interactive shell to make an airtight case for your deteerminiation of the order of operations and, or and not.

Documentation Go to this URL, `https://docs.python.org/3/library/stdtypes.html"` for a repository of information on Python's built-in types. You will see the number and boolean types near the begging. Further down the page is a section entitled, "Text Sequence Type — str." You can explore the string type there.

This URL, `https://docs.python.org/3/library/functions.html` has all of the Python built-in functions. You should poke around in here and look up the functions we have seen so far.

2.2 Variables and Assignment

You have programmed in some other language, and if you did, you might have seen that variables have a type. This is true in Java, C, and C++. This is not true in Python. Variables are typeless names that allow you to refer to objects.

Do not make the mistake of thinking Python is "weakly typed." Objects are keenly aware of their types. Bear witness to this little session. Every Python object knows its type. Python is *duck typed*; you can read about duck typing in [9].

```
>>> type(1)
<class 'int'>
>>> type(1.0)
<class 'float'>
>>> type("caterwaul")
```

```
<class 'str'>
>>> type(True)
<class 'bool'>
```

Objects do not have an identity crisis either. Every object in a Python program has a unique ID during its lifetime. We will use Python's built-in `hex` function to see these as hex numbers. These values will vary for your session. What you see is the memory address of your object in Python's virtual machine.

```
>>> hex(id(1))
'0x100214870'
>>> hex(id(2))
'0x100214890'
>>> hex(id(1))
'0x100214870'
>>> hex(id(1.0))
'0x10036f180'
>>> hex(id("caterwaul"))
'0x1019a4bf0'
>>> hex(id(True))
'0x1001b76b0'
```

The rule for Python variable naming is that the first character must be alphabetical or an underscore. Remaining characters can be alphanumeric or underscores. The operator `=` is used to bind variable to objects. It works in a manner entirely similar to other languages. Here we show it at work.

```
>>> x = 5
>>> y = 6
>>> x*y == 30
True
```

The value `x` is *not* storing the value 5. What it is storing is the location in memory (memory address) where the value 5 is being kept. You can see that what `x` is actually storing is the `id` of 5. In a word: Python variables know *where* to find their objects. In all cases, variables refer indirectly to their objects. What they actually store is a integer indicating where that object is being stored in memory. Bear this in mind as we proceed; it will save you a lot of confusion.

```
>>> hex(id(x))
'0x1002148f0'
>>> hex(id(5))
'0x1002148f0'
```

What happened on the last line, `x*y == 30`? Here, the expression was

evaluated by fetching the values of `x` and `y`, substituting them into the expression and finding that $5 * 6$ indeed equals 30.

2.2.1 The Lowdown on Assignment

The familiar arithmetic operations and the Boolean operations all associate from left to right. To wit, when you evaluate expressions, you work from left to right. Here we see this in action with multiplication and division.

$$4 * 5/2 * 8 = 20/2 * 8 = 10 * 8 = 80.$$

Now see it in a complex operation. Consider this expression.

$$4^3 - 2 * 7 * 5/35 + 4 * 18$$

We begin by doing all exponentiation.

$$4^3 - 2 * 7 * 5/35 + 4 * 18 = 64 - 2 * 7 * 5 + 4 * 18$$

We then munch up the multiplication and division in each term from left to right.

$$64 - 2 * 7 * 5 + 4 * 18 = 64 - 14 * 5 + 72 = 64 - 70 + 72$$

Lastly we resolve addition and subtraction.

$$64 - 14 * 5 + 72 = 64 - 70 + 72 = -6 + 72 = 66.$$

Notice how we work in each case from left to right.

Assignment works backwards. It begins on the right and works left. It also has lower precedence than any other arithmetic operation, so it happens last.

Here is a very typical assignment you might see.

```
>>> x = 5
>>> x = 2*x + 10
>>> x
20
```

Let us look at this in detail. Python first sees the variable `x` being bound to the value `5`. Now we turn our attention to the second line. We begin with

```
x = 2*x + 10.
```

Multiplication is first carried out. The term `2*x` evaluate to `10` and we have

```
x = 10 + 10.
```

Now addition occurs.

```
x = 20
```

Now, `x` is bound to the value `20`. What happened to its prior value `5`? This value got orphaned. To change the value of any variable pointing at any of scalar types we have seen so far, we have we get it to point at an entirely different object. We never modify the object sitting in memory. All of the scalar types are immutable; once created in memory they never change. In particular, the objects `True` and `False` are unique in memory.

Things that can appear on the left-hand side of an assignment are called *lvalues*. Variables are always lvalues; we will meet a few other things as we go along that are also lvalues. Literals, or actual objects, are not. You cannot assign to numbers, Booleans, or strings.

Python offers a lagniappe that is a twist on assignment. Look at this.

```
>>> a = 5
>>> b = 4
>>> a,b = b,a
>>> a
4
>>> b
5
```

Just a spoonful of syntactic sugar helps the medicine go down.

Programming Exercises

1. Do this and see how Python hisses.
   ```
   5 = x
   ```
2. What happens here?
   ```
   >>> a = 1
   >>> b = 2
   >>> c = 3
   >>> a,b,c = b,c,a
   ```
3. What happens when you do this?
   ```
   >>> a, b, c = c
   ```

2.3 Pooling

Python is a *garbage-collected* language. A mechanism called the garbage collector lurks behind the scene, deallocating the memory for objects no longer in

use.

Some objects don't get picked up by the garbage collectors; these exceptions are pooled objects. Python caches small integers in memory; when they reappear because a variable needs to point at them, the variable just points at the pooled value. Python also caches small strings in memory in an area called the *string pool*. Pooling of these immutable objects increases efficiency; equality of pooled strings is achieved by comparing memory addresses, obviating the need to loop through the strings.

Now, it's time to break out the `is` operator and see it at work.

```
>>> x = 5
>>> y = 5
>>> x is y
True
>>> name = "flibbertygibbet"
>>> elisa = "flibbertygibbet"
>>> name is elisa
True
```

The variables `x` and `y` are sharing the common item `5` in memory. The same is true for the two strings pointing at `"flibbertygibbet"`. Note that only two Boolean values are ever stored in memory, `True` and `False`.

```
>>> True is True
True
>>> False is (6*4 == 5*50)
True
```

Programming Exercises: Time for a dip!

1. Run this code.

   ```
   for k in range(200,300):
       print(k, hex(id(k)))
   ```

   What can you discern about the pooling of small integers?
2. Can you find anything out about negative integers by replaying this theme?

2.4 Writing a Program

A Python program is just a sequence of Python statements in a file. We will use the creation of this example as an opportunity to introduce the built-in function `print`, which puts things to `stdout`. Enter this with your favorite text editor into a file named `print_example.py`.

```
print("Hello, World")
print(1,2,3,4, sep="|")
print(1,2,3,4, sep = " ", end = "peep")
```

Now open a cmd or terminal window and navigate to the directory containing your program. Run this program at the command line as follows. The items `sep` and `end` are referred to as *keyword arguments*.

```
unix> python print_example.py
Hello, World
1|2|3|4
1 2 3 4peep
unix>
```

A Note to UNIX (Yes, Mac too... Users) When you make your program, place this line at the top

```
#!/usr/bin env python3
```

and use `chmod` to make the program executable If you do this to our litle sample program, you can do this. Notice what the keyword arguments `sep` and `end` do.

```
unix> ./print_example.py
Hello, World
1|2|3|4
1 2 3 4peep
unix>
```

Programming Exercises

1. You can get textual input from the user with the `input` function. It works like this.

   ```
   some_variable = input("Your prompt:  ")
   ```

   Write a program that asks for a name and which replies with `Hello, <Name>`.
2. Write a program that asks for two numbers and which presents the user with their product. Note that `input` returns a string.

2.5 Objects

The term *object* simply refers to a datum stored in memory along with its associated code. There are three important properties of an object

- **State** This refers to things an object *knows*.
- **Identity** This refers to an object's physical presence in memory. It is not possible for two different objects to occupy the same space in memory. This is what an object *is*.
- **Behavior** This refers to what an object *does*.

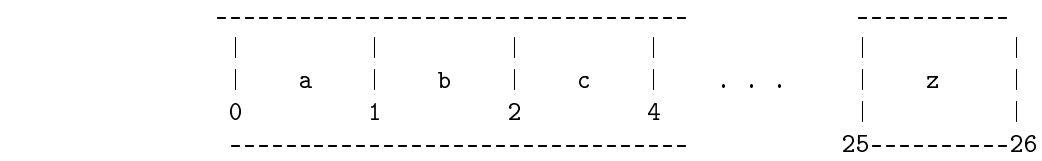
Let us look at the scalar types we have seen through this lens. Integers exhibit expected behavior when in the presence of arithmetic and relational operators. They know the value that they store. The same is true of floating-point numbers.

Strings are more complex. Their state is simple; this is just the blob of text being stored in the string.

Strings have a wide variety of behaviors. Let us look at a few. We can index into a string; most languages you have seen have this feature.

```
>>> x[0]
'a'
>>> x[1]
'b'
>>> x[2]
'c'
>>> x[25]
'z'
```

Notice that the indexing is zero-based. The best way to think about these indices is that they live, like rats, "inside of the walls." We say thes because they reside *between* then entries of the string.



As you can see in the picture, there is an index 26 in this string; it is just at the far right-hand end. Each index points at the character just to its right. There is no character for index 26 to point at. Try and see it; Python will hiss at you.

Let us illustrate another behavior, `find`.

```
>>> x.find("c")
2
>>> x.find("fgh")
```

```
5
>>> x.find("cow")
-1
```

When `find` does not find, it punts and returns a `-1`. This is often referred to as a *sentinel value*. Notice how `find` needs to be told what to find.

2.5.1 How do I find all of the string behaviors?

Visit the URL `https://docs.python.org/3/library/stdtypes.html` for information on all of Python's built-in types. Then find the section on the "Text Sequence Type." Here is what you will find at the top.

String literals are written in a variety of ways:

1. Single quotes: `'allows embedded "double" quotes'`
2. Double quotes: `"allows embedded 'single' quotes".`
3. Triple quoted: `'''Three single quotes'''`, `"""Three double quotes"""`

Triple quoted strings may span multiple lines, and all associated whitespace will be included in the string literal. You can use single or double quotes to bound a triple-quoted string.

Now scroll down the page a short bit to the section entitled "String Methods." The `capitalize()` method is very simple. We show its action. It creates a new string that is capitalized. The original string is untouched.

```
>>> president = "lincoln"
>>> president.capitalize()
'Lincoln'
```

Let us look at how to read the documentation for `center()`.

```
str.center(width[, fillchar])
```

Return centered in a string of length width. Padding is done using the specified fillchar (default is an ASCII space). The original string is returned if width is less than or equal to len(s).

This method has two arguments. The first one, `width`, is required. The second, `fillchar`, is optional; this is indicated by the presence of the square brackets surrounding it. Let us show this at work.

```
>>> x = "Cows With Guns"
>>> x.center(10)          #too little space: same string comes back
```

```
'Cows With Guns'
>>> x.center(50)          #default padding is spaces
'                  Cows With Guns                  '
>>> x.center(50, "*")     #padding with stars
'******************Cows With Guns******************'
>>> x.center()            #error: width is required.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: center() takes at least 1 argument (0 given)
>>>
```

We should mention that Python has many useful built-in functions. Here are three you want to know about.

Function	Input	Output
`len`	a string	the string's length
`ord`	a one-character string	the char's ASCII code
`chr`	an integer	the char with the given ASCII code

Programming Exercises Experiment with these useful string methods. Figure out what they do.

1. `rfind`
2. `endswith`
3. `startswith`
4. `lower`
5. `upper`
6. `strip`, `lstrip` and `rstrip`

2.5.2 Compound Assignment Operators

Many languages have this feature.

```
>>> x = 5
>>> x += 3
>>> x
8
```

Here `x += 3` is shorthand for `x = x + 3`. If you have an infix binary operator `op`, then `x op= foo` is the same as `x = x op foo`. The compound assignment operator works from right to left. Its precedence, like that of `=` is lower than almost everything else.

Notice this little session with strings.

```
>>> x = "some"
>>> id(x)
4323956024
>>> x += "thing"
>>> x
'something'
>>> id(x)
4323954160
```

The variable `x` is a pointing at a new string, because strings cannot be changed in-place. You can see this because the `id` of the object pointed at by `x` changed.

2.6 Sequence Types

So far, we have concerned ourselves with scalar types that hold a single datum. Now we will look at two new types, lists and tuples. These types have some common features with strings, because a string can be thought of as a character sequence, as well as a glob of text.

Sequences in Python store sequences of memory addresses where the objects comprising them can be found. The objects themselves are not stored in these containers.

Let us first look at lists. We make a list, show its type, and index into it. Notice that the indexing mechanism looks identical to that of strings.

```
>>> x = [1,2,3,4,5]
>>> type(x)
<class 'list'>
>>> x[0]
1
>>> x[1]
2
>>> x[4]
5
>>> x[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>>
```

What is different is that lists are mutable. Watch this.

```
>>> id(x)
4323859272
>>> x[0] = 100
>>> id(x)
4323859272
>>> x
[100, 2, 3, 4, 5]
>>>
```

We changed this list in-place. It is the same object, but its state has been changed by the assignment `x[0] = 100`. Each entry in the list is an lvalue.

Python tuples are similar to lists, but they are immutable. Once you make a tuple, you cannot change it in-place. Let us imitate the list session. All of this looks the same.

```
>>> x = (1,2,3,4,5)
>>> x
(1, 2, 3, 4, 5)
>>> x[0]
1
>>> x[1]
2
>>> x[4]
5
>>> x[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
>>> type(x)
<class 'tuple'>
>>>
```

Now watch this.

```
>>> x[0] = 100
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Oops. You cannot change a tuple in-place. Tuple entries are *not* lvalues.

Both tuples and lists are heterogeneous. You can put objects of any types into them.

```
>>> motley = [True, "foo", 2, 2.0, ["cat", False, -1]]
>>> motley[4]
```

```
['cat', False, -1]
>>> motley[4][1]
False
>>>
```

You can do all of this stuff with a tuple, since you are only reading from it. Try it now!

The + operator works just as you might expect for tuples and lists.

```
>>> [1,2,3] + [4,5,6]
[1, 2, 3, 4, 5, 6]
>>> (1,2,3) + (4,5,6)
(1, 2, 3, 4, 5, 6)
```

The built-in `len` function will calculate the length of any sequence. For tuples and lists, this is the number of items present in the tuple or list. For strings, this is the number of characters in the string.

Here are common features for all sequences.

Operation	List/Tuple	String
`x in s`	`True` if `x` equals a member of `s`	`True` if `x` is a substring of `s`
`s + t`	concatenates `s` and `t`	
`s*n` or `n*s`	repeats `s` `n` times	
`len(s)`	number of objects in `s`	number of characters is `s`
`min(s)`	smallest object in `s`	character in `s` with smallest ASCII value
`max(s)`	largest object in `s`	character in `s` with largest ASCII value
`s.count(x)`	counts the number of times `x` is equal to an element of `s`	counts the number of times the sting `x` appears in `s`

When we refer to the "largest" item in a sequence, we need to do this operation on a list where it makes sense to compare the items. It is best to use this on sequences that are homogeneous, i.e., where all elements are of the same type.

2.6.1 Slicing of Sequences

Slicing is convenient way of obtaining a subset of a sequence. For strings and tuples, a slice returns a copied subset of the sequence. First we see slicing at an index for a list and a string. Notice that the slice goes all the way to the end.

```
>>> min(x)
'a'
>>> x = "abcdefg"
>>> x[:2]
'ab'
>>> x[2:]
'cdefg'
>>> foo = ["a", "bc", "defg", "hijk"]
>>> foo[2:]
['defg', 'hijk']
>>> foo[:2]
['a', 'bc']
```

You can specify both ends of a slice.

```
>>> foo[1:3]
['bc', 'defg']
>>> x[1:3]
'bc'
```

You can also specify a third "skip" parameter.

```
>>> alpha = "abcdefghijklmnpqrstuvwxyz"
>>> alpha[::3]
'adgjmqtwz'
>>> alpha[5::3]
'filpsvy'
>>> alpha[5:10:3]
'fi'
```

In the first case, we extracted every third character from the string. In the second we did so starting at index 5. In the third, we did so between indices 5 and 10. Be reminded: the indices of a sequence lurk between the sequence's elements.

Programming Exercises

1. How do you find the item in a list or tuple of strings that is first in asciicographical order?
2. How do you count the number of elements in a list of integers of even index that equal 5?
3. How do you count the number of times the letter A appears in a string, case insensitive?
4. If you have a sequence what does the slice [::-1] return? What happens if you put indices between the colons?

2.6.2 Slicing of Lists

Because lists are mutable, they have additional behavior when slicing occurs. Observe this.

```
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
>>> x[:5] = []
>>> x
[5, 6, 7, 8, 9, 10, 11, 12]
```

A slice is an lvalue. We can, within bounds, assign to it. If you assign an empty list to a slice of a list consisting of consecutive elements, the slice is removed from the list. Beware of this.

```
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
>>> x[::2] = []
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: attempt to assign sequence of size 0
to extended slice of size 7
```

There are limits. Here the slice you tried to assign to had non-consecutive elements. You can, however assign this slice to a list of equal length like so.

```
>>> x[::2] = [0, 10, 20, 30, 40, 50, 60]
>>> x
[0, 1, 10, 3, 20, 5, 30, 7, 40, 9, 50, 11, 60]
```

It is interesting to compare the action of += on lists, strings and tuples. Compare these three sessions.

```
>>> x = [1,2,3,4,5]
>>> id(x)
4323953160
>>> x += [6,7, 8]
>>> x
[1, 2, 3, 4, 5, 6, 7, 8]
>>> id(x)
4323953160
```

Here we just modified an object in-place

```
>>> id(x)
4298553304
>>> x += (6,7,8)
```

```
>>> x
(1, 2, 3, 4, 5, 6, 7, 8)
>>> id(x)
4323935288
```

Here a new object got created because tuples are not mutable.

```
>>> x = "12345"
>>> id(x)
4323956024
>>> x += "678"
>>> x
'12345678'
>>> id(x)
4323953840
```

The same thing happened to the string and the tuple. New objects got created by +=.

2.7 Casting About

A *cast* is a temporary request to view an object of one type as being that of another. We show some examples here. Any Python object may be cast to a string.

```
>>> int("12345")
12345
>>> float("12345")
12345.0
>>> str(12345)
'12345'
>>> int("12345", 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 2: '12345'
>>> int("10101110", 2)
174
>>> int("22123312", 4)
42742
```

Observe that you can pass a radix to cast a string to a number in that base. Lists can be cast to tuples and vice versa in the obvious way. You can also cast a string to a list.

```
>>> t = (1, 2, 3, 4)
>>> list(t)
[1, 2, 3, 4]
>>> l = [1,2,3,4]
>>> tuple(l)
(1, 2, 3, 4)
>>> list("caterwaul")
['c', 'a', 't', 'e', 'r', 'w', 'a', 'u', 'l']
```

Programming Exercises

1. Cast all of the types we have seen so far to a string and see what happens.
2. Strings have a method called `join` that takes a list or tuple as an input. If you cast a string to a list, how can you use `join` to undo the action?

2.8 List Behaviors

Because they are mutable, lists exhibit some behaviors not available to tuples and strings. We have noted already that they behave differently when slices are taken. Here is a table of some of the most important operations and methods.

Operation	Action
`s[x] = y`	reassigns the value held by `s` at index `x` to `y`. List entries are lvalues.
`s[m:n] = [], del s[m:n]`	deletes all values between the indices `m` and `n` provided `m < n`.
`del s[a:b:c]`	deletes all elements of the indicated slice
`s.append(x)`	appends item `x` to the list `s`
`s.extend(t)`, s += t	appends items in the sequence `t` to the list `s`
`s.clear()`	empties the list `s`
`s.insert(i, t)`, `s[i:i]=t`	splice in the sequence `t` into the list `s`
`s.reverse()`	reverses the list in place.

2.9 Hashed Types

Python has two unordered types, dictionary (`dict`) and (`set`). A set is a container that does not admit duplicate entries, as defined by `==`. A dictionary is a container holding key-value pairs. These are made very efficient via a means called *hashing*. We will begin by describing this very clever mechanism. It makes access to items in sets and dictionaries fast and efficient.

2.9.1 What hashing? Why do it?

A *hash function* is a mathematical function whose domain is some collection of Python objects (e.g. strings) and whose codomain is the integers. Here we show the hashing function at work on strings.

```
>>> hash("a")
-8506767599803020586
>>> hash("b")
-7609782221146829849
>>> hash("c")
-5419851217699219052
>>> hash("ab")
-2244008983755709986
```

Here it is on integers. "Small" integers hash as themselves; once they get to a certain outrageous size, a truncation process occurs.

```
>>> hash(0)
0
>>> hash(1)
1
>>> hash(1024)
1024
>>> hash(1024576)
1024576
>>> hash(331214214214241)
331214214214241
>>> hash(413908140942980249081902)
97417085330101598
>>>
```

A perfect hash function will give different values to different objects. Hash functions depend on the state of the object they are hashing. Because mutable objects can have their state changed, this hashing process is not possible for them. Were a mutable object hashable, its hash would change whenever its state did. And, for the purposes we are about to describe here, that is very bad news.

One way we could implement a set us just to use a list and to reject the addition of duplicate elements. This causes inefficiency. Searching a list for an item is an $O(n)$ process, because the amount of resources it takes is at worst proportional to n, the size of the list.. As the set got big, adding new elements would become burdensome.

So what do we do? The hash function provides the key. First, we reserve a big chunk of memory, say of size M. When we add an element we hash it, mod

out by M and get an nonnegative integer less than M. We then store that object at that index in the chunk of memory (I lied... we store its memory address so we have access to it). So, to check for the presence of that object, we hash it and know precisely where it is stored. Hey, this is an $O(1)$ (constant-time) procedure.

The Nasty Hairy Fly in the Sweet Ointment Even if you have a perfect hash function (in practice never), this process of modding by M can cause a new item to be placed into an occupied slot. Beezlebub! Defeat!

Nah, what we do is store a little list of objects; we can go to that location and check that list. This kind of thing happens if the chunk of memory gets too crowded.

At some level of crowding, the whole thing will be put in a new, bigger, chunk of memory and everything will be rehashed to relieve the crowding.

Both sets and dictionaries achieve quick access of elements in this manner. Next, we will learn about sets mathematically and learn how they are implemented in Python.

2.10 Sets

Informally, in mathematics, a set is a collection of objects with along with a notion of belonging. In computer science, we are concerned with finite collections of objects, so some of the hairier aspects of axiomatic set theory will not come our way. However, we will have a brief discussion of some basic ideas of set theory and we will see how they are implemented in Python.

No meaningful discussion is possible in the absence of context. So, when we discuss sets, we will discuss them withing a *universe of discourse* which we will generally denote by Ω. Such a thing is often infinite. Example: the set of all ASCII characters strings. Or, perhaps the set of all integers. When we discuss sets, we will always have some universe of discourse in mind.

Inside of our universe of discourse, we can define sets two ways. One is by making an explicit list of elements. For example if $\Omega = \mathbb{Z}$, the set of all integers,

$$A = \{1, 6, -5, 4, 3\}$$

is a legitimate way to define a set. For belonging, we use the symbol $\in$. So, here $6 \in A$. To negate belonging, we use $\notin$; for example $100 \notin A$.

A set in our universe is well-defined if we can tell if any given element in the universe is or is not in the set. This leaves us another way to define a set, by using a predicate. For example let

$$E = \{x \in \Omega | x\%2 = 0\}$$

is the set of even numbers. Note we are using mathematical notation here; the Python form of this predicate is `x % 2 == 0`.

If A and B are sets in a universe Ω, we will write $A \subseteq B$ to mean that every element of A belongs to B. We define $A = B$ to mean $A \subseteq B$ and $B \subseteq A$. In other words, sets are equal if they contain exactly the same elements. We write $A \subset B$ to indicate that $A \subseteq B$, but that B has some element not in A and we say that A is a *proper subset* of B.

One implication of this definition is that the order in which elements of a set are presented is immaterial, and if you list an element twice, it is no different from having the element in once.

Now it's time for some nitty-gritty with Python. Let's put some animals on our farm in a list. We will then cast the list to a set.

```
>>> farm = ["sheep", "sheep", "cow", "horse",
        "pig", "goose", "pig", "cow"]
>>> animals = set(farm)
>>> animals
{'pig', 'horse', 'goose', 'sheep', 'cow'}
```

Notice that all duplicates got removed.

Take note that only hashable elements can be placed in a set. In this book, we will only put immutable objects in a set; these are always hashable. This restriction makes access to items in a set very fast, as we described at the beginning of this section.

Bear witness to this punishment dished out by an irate python when we try to hash the unhashable.

```
>>> hash("cow")
7419535498109472991
>>> hash([1,2,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Also, notice that items in a set seem to be presented in no seemingly discernible order.

Now let us see how Python implments these concepts. We begin with $\in$.

```
>>> "horse" in animals
True
>>> "rhino" in animals
False
```

We see that $x \in A$ if `x in A` is `True`, and $x \notin A$ otherwise.

Python also implements $\notin$. If `x not in A` is true, then $x \notin A$.

Now let's go for $\subseteq$.

```
>>> sample = {"goose", "cow"}
>>> animals.issubset(sample)
False
>>> sample.issubset(animals)
True
```

You can also do this; its a nice mnenomonic.

```
>>> animals <= sample
False
>>> sample <= animals
True
```

The relational operator `<=` is the subset relation on sets. It has a strict version to indicate the "is a proper subset of" relation $\subset$.

```
>>> sample < animals
True
>>> animals < animals
False
>>>
```

Now we will see how boolean operations can be used in set-world. If A and B are sets in a universe Ω, we define the *complement* of A by

$$A^c = \{x \in \Omega | x \notin A\}.$$

The *union* of A and B is defined by

$$A \cup B = \{x \in \Omega | x \in A \vee x \in B\}.$$

Note that math uses $\vee$ for the infix binary `or` operator, and $\wedge$ for the infix `and` operator. It is easy to see that $A \cup B = B \cup A$, since this set is everything belonging to at least one of A or B.

We define the *intersection* of A and B by

$$A \cap B = \{x \in \Omega | x \in A \wedge x \in B\}.$$

To say it quickly, $A \cap B$ contains exactly all elements common to A and B. Think: street intersection, that which belongs to both streets.

Let us now see what Python has for union and intersection. There is no surprise here.

```
>>> zoo = {"rhino", "zebra", "sheep", "horse", "elephant"}
>>> zoo.intersection(animals)
{'sheep', 'horse'}
>>> zoo.union(animals)
{'horse', 'elephant', 'pig', 'zebra', 'goose', 'rhino',
    'sheep', 'cow'}
>>> zoo | animals
{'horse', 'sheep', 'rhino', 'pig', 'elephant', 'goose', 'cow', 'zebra'}
>>> zoo & animals
{'horse', 'sheep'}
```

There are two other set-theoretic operations that come in handy when handling data. There is the *relative complement* defined by

$$A - B = A \cap B^c = \{x \in \Omega | x \notin B\}.$$

This is the set of all things present in A not present in B. And there is the *symmetric difference*

$$A \triangle B = (A - B) \cup (B - A),$$

which consists of all elements belong to exactly one of A or B. Note the

$$A \triangle B = \{x \in \Omega | x \in A \oplus x \in B\},$$

where $\oplus$ is the infix exclusive or operator. Python handles this with aplomb.

```
>>> animals.symmetric_difference(zoo)
{'elephant', 'pig', 'zebra', 'goose', 'rhino', 'cow'}
>>> animals.difference(zoo)
{'cow', 'goose', 'pig'}
>>> zoo.difference(animals)
{'zebra', 'rhino', 'elephant'}
>>> animals ^ zoo
{'goose', 'rhino', 'elephant', 'pig', 'cow', 'zebra'}
>>> animals - zoo
{'goose', 'pig', 'cow'}
```

A set is a mutable object. It supports the `len` function, which will tell you how many elements it has.

Here is how to add new elements to a set.

```
>>> new_set=set()
>>> ne_set.add(1)
>>> ne_set.add(False)
```

```
>>> ne_set.add("cows")
>>> ne_set
{False, 1, 'cows'}
```

Now let's add a duplicate.

```
>>> ne_set.add("cows")
>>> ne_set
{False, 1, 'cows'}
```

The addition of the duplicate is ignored. To get rid of an element, use `discard`

```
>>> new_set.discard("cows")
>>> new_set
{False, 1}
```

Here is how to chuck everything.

```
>>> new_set.clear()
>>> new_set
set()
```

Programming Exercises

1. What does `is_disjoint()` do?
2. What does `pop` do? What is maddening about it?
3. Spelunking exercise: What types can you cast a set to? What types can you cast as a set? What happens in each case?
4. What happens if you try to index into a set?

2.11 Dictionaries

Imagine that you might want to have a list indexed by something other than numbers. For this purpose, Python features a second hashed data structure, the `dictionary`. Hashing, as we discussed before, gives rapid access to dictionary entries. Here we create a dictionary that stores telephone extensions. First we show how to create an empty dictionary.

```
>>> phone = {}
```

Now we show how to pre–populate a dictionary with a couple of entries.

```
>>> phone = {"morrison":2746, "yeh": 2725}
>>> phone["morrison"]
2746
```

Each dictionary entry consists of two parts. The first part is called the *key* and the second part is called the *value*. For any key `k`, its corresponding value is `phone[k]`. Shortly, we shall see that the value `phone[k]` is an lvalue. Because dictionary entries are retrieved via their keys, the keys of the dictionary must be hashable objects.

Notice the action of the `in` operator in a dictionary; this checks for membership in the keys.

```
>>> "morrison" in phone
True
>>> "sarocco" in phone
False
```

We can also check for the presence of a value.

```
>>> 2746 in phone.values()
True
>>> 2020 in phone.values()
False
>>>
```

It is very easy to add a new entry. Just associate a value with a key not present in the dictionary.

```
>>> phone["sarocco"] = 2722
>>> phone
{'yeh': 2725, 'sarocco': 2722, 'morrison': 2746}
>>> phone["miller"] = 2741
>>> phone
{'miller': 2741, 'yeh': 2725, 'sarocco': 2722, 'morrison': 2746}
```

You can get all of the key values in a list by using the keys() method. You can do a similar thing for getting all of the values in the dictionary.

```
>>> phone.keys()
dict.keys('miller', 'yeh', 'sarocco', 'morrison')
>>> phone.values()
dict.values([2741, 2725, 2722, 2746])
>>>
```

Finally you can change the value for any key as follows.

```
>>> len(phone)
3
```

So if you make an assignment `phone[foo] = blah`, the dictionary checks itself for the presence of `foo`; if `foo` is present, the value attached to it is changed to `blah`. If not, then the value `foo` is added to the keys and `blah` is assigned as its value.

Also, dictionaries know their size; just use `len`.

```
>>> phone["miller"] = 3714
>>> phone
```

2.12 Terminology Roundup

- **cast** This is a temporary request to regard an object to be regarded as another type
- **complement**A is a set, then the complement of A is the set of all elements in the universe of discourse not belonging to A.
- **disjoint** Two sets are disjoint if the have no elements in common.
- **garbage-collected language** these languages manage memory for you. Unused objects are automatically deallocated by a background process called the *garbage collector*
- **f-string** This is a format string. It is preceded by an `f` and can contain expressions that are surrouned by curly braces. These expressions are evaluated, converted into strings, and placed in evaluation of the f-string.
- **hash function** This is a function whose domain is a set of Python objects and whose codomain is the integers. A hash function is perfect if different objects always return different values.
- **intersection** The intersection of two sets is the set of all elemenets belonging to both of the sets.
- **keyword arguments** These are named arguments which are optional and which go at the end of a function's argument list.
- **lvalue** This is a symbol representing addressable memory. Variables, list items, and list slices are all lvalues. The term **object** simply refers to a datum stored in memory along with its associated code. Objects have three attributes, state (what an object knows), identity (an object's presence in memory), and behavior (what an object does).
- **proper subset** We say that A is a proper subset of B if every element of A belongs to B as well, and B contains at least one element not belonging to A.

- **relative complement** This is the set of all elements lying in one set but not another.
- **string** This refers to a contiguous piece of a sequence type. You can use the postfix [:::] operator to obtain a slice of a sequence
- **string pool** This is an area of memory in which small strings are kept and which is not garbage collected.
- **sentinel value** This is a return value for function that tells you something has gone wrong.
- **string** This is a character sequence.
- **subset** We say that A is a subset of B if every element of A belongs to B as well.
- **symmetric difference** The symmetric difference of two sets is the set of all element beloning to exactly one of the two sets.
- **type** This refers to the species of an object. Examples include integer, string, and boolean. The **union** The union of two sets is the set of all elements belonging to at least one of the sets.
- **universe of discourse** In set theory, this is the contextual bounds of the discussion; to wit, it is the set of all objects we speak of.

Chapter 3

Boss Statements

Python Boss Statements

3.0 Introduction

Here is what we have at our disposal so far. Python objects can respond to operators and methods. We have a rich ecosystem of very useful objects, including collections such as sets, lists and dictionaries. In this way, a Python method can do quite a bit of work for us.

Despite this, our palette for writing programs is pretty limited. All we can do at this time is to write a list of Python worker statements and to execute them *in seratum*. Since you have programmed before you will find this to be very dull indeed. We shall remedy that. The cure is the boss statement.

Statements work like clauses in English. A statement that reads as a complete sentence, or independent clause, is called a *worker statement*. A worker statement is a simple imperative sentence with tacit subject "Python." Here are some some examples.

1. `x = 5` Read this as "`x` gets 5" or "make `x` point at the value 5."

2. `print(x*x + 3)` Read this "Evaluate the expression `x*x + 3` and put it to `stdout`."

3. `x = 3*x - 7` Read this as "Evaluate the expression `3*x + 7` and have `x` point at the result.

3.1 Functions

Our first boss statement is the function header. Functions in Python work much like functions in other languages such as Java, JavaScript, or C. Let us begin by showing a function that squares a number.

```
def square(x):
    return x*x
```

The first statement `def square(x)` should be read, "To define `square(x)`," Notice that it is a grammatically incomplete sentence. This statement is an example of a *boss statement*, which controls the flow of execution in a program. A boss statement must own a *block* of code, which is one or more lines of code underneath it that are indented the same amount. This block of code, combined with the boss statement, constitutes a grammatically complete sentence. Notice that the block is indented one tab stop.

Reminder for users of `vi/vim` Change to your home directory now and edit our `.vimrc` file; if you don't have it, create it. Make sure these lines are in it

```
syntax on
set tabstop=4
set et
```

The first line gives you syntax coloring for all of your favorite file types. The second line sets the tab stop at 4 spaces. Do not put spaces around the equal sign or you will get annoying error messages! The third line turns every tab into 4 spaces. This eliminates a serious aggravation. Do this now and there will be peace in the valley.... or else. This will spare you annoying "inconsistent use of tabs and spaces," and other whitespace-related errors. You can also do this in other text editors by hunting in the preferences.

Now add to your function as follows. Put this in a file named `boss.py`.

```
def square(x):
    return x*x
print(type(square))
print(f"The Square of 10 is {square(10)}")
```

Now we run it and we see this.

```
unix> python boss.py
<class 'function'>
The Square of 10 is  100
```

A function is just another Python object! Now we see why `def square(x):` is really not a complete sentence. It is about as complete as the half-done assignment `x =`. In fact, defining a function *is* an assignment.

On the second line we are using, or *calling* our function. It does what it expected; it squares the number given or *passed* it and it outputs, or returns 100.

The code following the `def` boss statement `return x*x` is its block of code. Read it now; you have the complete sentence, "To define square(x), return x*x." Et voila! The block completes the boss statement grammatically. The block of a boss statement can contain one or more lines of code. The block ends where the indentation ends. Boss statements can be nested; a block of code can include boss statements, each of which owns a block of code.

3.2 Scoping

Variables created outside of any function live in the program's *global scope*. It is best not to minimize the use of these, for they will give you worlds of pain and little joy or use. In this book, we will eschew them assiduously.

Functions residing directly in a file also have global scope. This means that functions can be seen by other functions and that they can call each other. This is good. It allows you to create "teams" of functions that work together to accomplish a task.

Let us do something interesting with our little square function. Make this program called evil_scope.py

```
def square(x):
    y = x*x
    return y

print(square(12))
print(x)
print(y)
```

We now get hissed at by an angry Python.

```
unix> python evil\_scope.py
144
Traceback (most recent call last):
  File "evil\_scope.py", line 6, in <module>
    print(x)
NameError: name 'x' is not defined
```

It would seem that `x` would be storing the value `12`. Evidently not. Here is what happened.

1. A copy of the memory address stored by the variable is sent to the function's parameter `x`.
2. If a literal is passed, a copy of its location in memory is sent to the parameter `x`.
3. An internal variable `y` got created here. It got the value `144`.
4. The memory address where `144` is stored is passed back to the `print` function, which prints `144`.
5. Once the function returned, all traces of the variable `x` disappeared. The same thing happened to `y`. To see this, delete the line where `x` is printed and re-run the program.

We have just learned two things about Python. One is that Python has *function scope*. Arguments of functions and variables created inside of a functions are not visible outside of that function.

The other is that Python is a pure pass-by-value language. When you pass a variable or a literal to a function, the function actually gets a *copy* of the memory address of that item.

You might ask, "Why all of this hiding of stuff?" You have seen us use some built-in functions such as `len`, `min` and `max`. Would you want to have to worry about variables created inside of these and have to keep track of their names? This would quickly lead to choking levels of complexity. When using functions all we need to care about is what sorts of parameters they need and what they do. This frees us to think about the problem we are trying to solve and not about the internal finickiness of our tools.

When writing a function, you must concern yourself with these major areas.

1. **parameters** How many of these are there are of what type(s) should they be?
2. **preconditions** More generally, what should be true when you call this function? Really, a description of your parameters is part of the preconditions.
3. **side-effects** Does this function leave stuff behind once it returns? Does it create a file? Does it put things to `stdout`? Does it modify the state of any mutable objects?
4. **return value** What kind of object, if any is returned by the function? If you do not return anything a graveyard object named `None` is automatically returned.
5. **postconditions** This is what is true when a function is done executing. This contains a description of a function's side-effects and return value, if any.

As with any programming language, Python has a system that makes it easy to document your function. You insert a *docstring*, as shown here.

```
def square(x):
    """precondition:  x is a number (integer or float)
postcondition:  This function returns the square of x.
It has no side-effects."""
    y = x*x
    return y
print(square.__doc__)
```

Now we run this; you will see how to print the docstring of any function that has one.

```
unix> python docstring.py
precondition:  x is a number (integer or float)
postcondition:  This function returns the square of x.
It has no side-effects.
```

3.3 Conditional Logic

A mainstay of computer languages is the `if` statement and its friends. We will now explore these in Python. There are three important boss statements in Python's conditional logic, `if`, `else`, and `elif`.

We being by looking at the simple `if`. Its usage is as follows.

```
if predicate:
    code
```

The item `predicate` is any boolean-valued expression. If `predicate` evaluates to `True`, the code in the block executes. If not, the code in the block is skipped.

The `else` construct allows you to provide code that is executed if an `if` statement's predicate evaluates to `false`. Here is its usage.

```
if predicate:
    codeIfPredicateIsTrue
else:
    codeIfPredicateIsFalse
```

This puts a two-way fork in your code. One of the two alternatives *must* be executed.

The `elif` construct provides a means of producing a multi-way fork. Let us create an example. Imagine you are assigning grades. Let's do this. For A,

B, C and D, you can have a + or - mark, but not for F. The input will be a percentage; if the last digit is 7,8 or 9, a + is granted. If it is a 0, 1, or 2, a - is given. Otherwise, there is no + or -. Let us build this.

So here is how we will get the score.

```
score = input("Enter a percentage:  ");
```

Run it and see that it works. We will write two functions, one for the letter grade, the other for the +/- modifier.

```
def letter_grade(n):
    if n < 60:
        out  = "F"
    elif n < 70:
        out  = "D"
    elif n < 80:
        out  = "C"
    elif n < 90:
        out  = "B"
    elif n <= 100:
        out  = "A"
    else:
        out = "Illegal grade!"
    return out
score = input("Enter a percentage:  ");
```

Run this and you are now reminded: cast things you get from `input` to a number type. For the sake of simplicity, we will use integers here.

```
unix> python grades.py
Enter a percentage:  95
Traceback (most recent call last):
  File "grades.py", line 16, in <module>
    print(letter_grade(score))
  File "grades.py", line 2, in letter_grade
    if n < 60:
TypeError: unorderable types: str() < int()
```

Just add this line

```
score = int(score)
```

The letter in the grade looks good.

```
unix> Feb 02:13:15:ppp> python grades.py
Enter a percentage:  95
A
unix> python grades.py
Enter a percentage:  90
A
Enter a percentage:  88
B
unix> python grades.py
Enter a percentage:  77
C
unix> python grades.py
Enter a percentage:  66
D
unix> python grades.py
Enter a percentage:  44
F
```

One lurking issue to watch for is to ensure a 100 gets an A+. We will make a second function to produce the modifier. Notice that if the student fails, he gets no modifier.

```
def modifier(n):
    out = ""
    if n >= 60:
        if n % 10 <= 2:
            out = "-"
        if n % 10 >= 8:
            out = "+"
    if n == 100:
        out = "+"
    return out
```

Here is some testing. You should check all branches and see to it that you are happy.

```
Enter a percentage:  100
A+
unix> python grades.py
Enter a percentage:  98
A+
unix> python grades.py
Enter a percentage:  94
A
unix> python grades.py
Enter a percentage:  91
```

```
A-
unix> python grades.py
Enter a percentage:  88
B+
unix> python grades.py
Enter a percentage:  59
F
unix> python grades.py
Enter a percentage:  61
D-
```

But what if the fool enters gibberish? Take exception. The fool! Spectate, videte, et ecce!

```
unix> python grades.py
Enter a percentage:  cats
Traceback (most recent call last):
  File "grades.py", line 28, in <module>
    score = int(score)
ValueError: invalid literal for int() with base 10: 'cats'
```

We will "try" to convert the input and punt if the `ValueError` rears its ugly head. Raising this error brings the program to a screeching halt at the scene of the crime. Here is the full code listing.

```
def letter_grade(n):
    if n < 60:
        out  = "F"
    elif n < 70:
        out  = "D"
    elif n < 80:
        out  = "C"
    elif n < 90:
        out  = "B"
    elif n <= 100:
        out  = "A"
    else:
        out = "Illegal grade!"
    return out
def modifier(n):
    out = ""
    if n >= 60:
        if n % 10 <= 2:
            out = "-"
        if n % 10 >= 8:
```

```
            out = "+"
    if n == 100:
        out = "+"
    return out
def grade(n):
    return letter_grade(n) + modifier(n)
score = input("Enter a percentage:  ")
try:
    score = int(score)
except ValueError:
    print("Illegal entry.  Try an integer 0-100.")
    quit()
print(grade(score))
```

You now begin to see how functions can call each other and how they can work as a team to separate a problem into manageable pieces that you can easily code. The function `grade` is an orchestrator that spits out the grade and which is really the only function the end-user ever calls.

So, now by example we see three conditional situations. The simple `if` just skips its code if its predicate is true. An `if-else` progression executes the `if` block if the predicate is true and the `else` block if the predicate is not true.

An `if-elif-else` progression keeps trying until a predicate is true. When it occurs, it executes that predicate's block and drops out of the progression. Is is good practice when using these to have an `else` block to handle any potential errors. Note that without an `else` block, the progression can go by and do nothing.

Programming Exercises Time for a date! Here are some calendar-oriented programming challenges. Functions shown are stubbed in so you can put this Python code in a file and it will run.

1. Write a function called `is_leap(year)` which returns `True` if the year leaps and which returns `False` otherwise. Here is the rule.
   - If a year is divisilble by 4 it leaps.
   - BUT every 100 years there is an exception.
   - BUT every 400 years there is an exception to the exception!
2. Here are some other functions to implement.

```
def date_plus(the_date):
    """Precondition: the_date is a string containing a date.
Postcondition: Return the sum of the year, month, and day.  You must accept any of the follo
dd/mm/yyyy
dd-mm-yyyy
ddmmyyyy
```

```
    ddmmyy
    dd/mm/yy
    dd-mm-yy
    """
        pass
    #here are some tests to implement.
    date_plus("01/01/1970") == 1972
    date_plus("08/12/1995") == 2015
    date_plus("08/12/95") == 2015
    date_plus("08/12/14") == 2034
```

3. ```
   def dayInYear(year, month, day):
       """prec:  year/month/day is a valid date
       postc: returns the ordinal position of the day in the year
       (Feb 15 is the 44th day of year 2000).
       Hint:  The list method sum is your friend. Learn about it."""
       return 0
   ```

4. ```
   def daysLeftInYear(year, month, day):
       """prec:  year/month/day is a valid date
       postc: returns the number of days left in the year
       (Feb 15 is the 44th day of year 2000)."""
       return 0
   ```

3.4 Stack and Heap

There are two chunks of memory that are important to your programs. The *heap* is a data warehouse where all of your objects are stored. Space on the heap is quite plentiful but not unlimited. If a Python program is running out of heap memory, it can request the operating system allocate it more.

The *stack* is the portion of memory where function calls are managed. This segment of memory is of a fixed size; if you use all of its memory up you will get a dreaded stack overflow error handed to you and your program will die.

3.4.1 The Heap

If you make a variable pointing at *War and Peace*, the text of that book will be stored on the heap, and the variable will store the memory address where the text is kept. This is true of all Python objects. We learned earlier that small integers are kept in an easy-access portion of the heap.

What about functions? The instructions necessary for them to execute are stored on the heap.

What about a list or other container? The entries of any container actually store the memory address of the objects they represent. So, for example, a list lives on the heap. Its entries all store heap addresses for the objects they represent. Visually, you can think of a list as resembling long-legged millipede, whose "legs" point ot the objects being stored in the list.

3.4.2 Program Life Cycle

When your program is run, it begins by creating the *global frame*. This is a container that stores all of the data you create in the main routine of your program. Python reads all of your functions and stores the code for then to execute on the heap. The variable names (function names) are stored in the global frame, and under them the heap addresses for their code is stored. The global frame only stores variables and their memory addresses. All of the objects you create live on the heap.

In the beginning of our example,`letter_grade`, `modifier`, and `grade` are read into memory. We then get to this code in the "main routine" of our program.

```
score = input("Enter a percentage:  ")
try:
    score = int(score)
except ValueError:
    print("Illegal entry.  Try an integer 0-100.")
    quit()
print(grade(score))
```

The variable `score` is a global variable and it and the memory address of its string are stored in the global frame. The global frame is at the bottom of the call stack.

The stack stores data structures called *activation records* or *stack frames*; the global frame forms the bottom of this stack. A stack frame contains several pieces of information.

1. It stores a `return address` that tells the program where to go back to after the function returns. The global frame is an exception; once it returns execution ends.
2. It stores the parameters and the memory addresses of the objects passed to them.
3. It holds a local symbol table with all of the variables created inside of the function. The data from the parameters is loaded into this symbol table as well. When the function returns, this symbol table is exposed to be overwritten and it becomes inaccessible.

When a function is executing, its local symbol table and the global symbol table (which contains the other functions) are visible. All of Python's core linguistic infrastructure are visible during the entire lifetime of your program.

The first line of the main routine makes a function call to `input`. This call is placed on the stack. The `input` function waits for the user to type in an entry; when the enter key is hit, it returns the string the user typed in. Once this function returns, its stack frame is exposed to be overwritten; for all practical purposes it is destroyed. It disappears with no trace.

Then we perform the cast, which is really a function call. If the user types in gibberish, a `ValueError` is raised and, if there is no `except` block, the program dies right on the spot, right in the middle of the call to the `input` function.

Now let us suppose that we successfully get an integer from the user. A call is made to `print`. Right in the middle of `print` trying to do its job, we call `grade` and pass it `score`. So, on top of `print`'s stack frame we plop a stack frame for `grade`.

This frame stores a return address so it can get back to where we left off in `print`'s execution. Now `grade` is running and it can see `n` in its private symbol table, which contains the memory address where `score` is living on the heap.

The first line of `grade` has a call to `letter_grade` in it; this call is being passed `n` so it gets a copy of the memory address where `score` is stored. You see a local variable `out` that gets created. The appropriate branch hands `out` a letter grade. The actual letter is created on the heap; `out` merely knows its memory address. That gets sent back to its caller, `grade`. This process is repeated for `modifier`.

With both parts of its return value in place, `grade` returns a grade with its modifier to `print`, which ships the result to `stdout` and returns. The program then ends. Throughout this book we will adhere to this style convention: *All functions should be defined before any other code is placed in a program.* This keeps things sane.

So to summarize, as a program runs, the following happen.

1. Python reads the program from top to bottom.
2. As functions are defined, their code is placed in heap memory and their memory addresses where they are stored goes into the global symbol table.
3. The first unindented line after the functions are seen is the beginning of the program's "main routine." Variables created in the main routine go in the global symbol table. Since the main routine is last, variables in it cannot be seen by the functions without a little extra code. This is bad practice; we will refrain from doing this.
4. When a function is called, its stack frame goes on the call stack. The function puts the function's parameters and local variables in a local symbol

table, which resides in its stack frame. The stack frame also maintains a return address to go back to once it returns, and a bookmark of where it is in its progress. If a function calls another function, a stack frame is created for that function and it is placed on the stack.

5. When a function returns, its stack frame is popped (removed) from the call stack and it is no longer accessible. All of the function's local variables die.
6. The stack will grow and shrink as the program runs.
7. When the main routine returns, program execution is over. The OS reclaims the space occupied by your program.

Recommendation A smart way to make programs is to place your main routine inside of a function called `main` and just call `main` in the main routine. Here is an example. In this way, the *only* occupants of the global symbol table are functions. The only worker statement in the main routine is `main()`. Here is a modest example.

```
def greet(name):
    return "Hello, " + name
def main()
    print(greet("General Grant"))
main()
```

The only items in your global symbol table are functions.

Programming Exericises

1. Run this program

   ```
   def square(x):
       y = x*x
       print(f"inside squre: {locals()}")
       return y
   def cube(x):
       y = x*square(x)
       print(f"inside cube: {locals()}")
       return y
   def main():
       print(f"inside cube: {locals()}")
       print(cube(5))

   main()
   ```

   What is it showing you?

2. Use the device you saw in the first problem to spy on the local variables in the grading program.

3.5 Recursion

Functions can call themselves, and this can often be useful. In fact, we can achieve repetition with this mechanism. Here is an example.

```
def rectangle(width, height, ch):
    if height == 0:
        return
    print(ch* width)
    rectangle(width, height - 1, ch)
rectangle(5,6, "*")
```

Let us run this for the doubting Thomases out there.

```
unix> python rectangle.py
*****
*****
*****
*****
*****
*****
```

So, what happened? To understand this we must analyze the call stack. Begin by adding a line to our code to see the local symbols that are visible.

```
def rectangle(width, height, ch):
    if height == 0:
        return
    print(dir())
    print(ch* width)
    rectangle(width, height - 1, ch)
rectangle(5,6, "*")
```

Running this we see the following.

```
unix> Feb 03:11:32:p1Code> python rectangle.py
['ch', 'height', 'width']
*****
['ch', 'height', 'width']
*****
```

```
['ch', 'height', 'width']
*****
['ch', 'height', 'width']
*****
['ch', 'height', 'width']
*****
['ch', 'height', 'width']
*****
unix>
```

Now let us print out the values held by these symbols. Delete the `dir` line and do this.

```
def rectangle(width, height, ch):
    if height == 0:
        return
    print("ch = %s, height = %s, width = %s" % (ch, height, width))
    print(ch* width)
    rectangle(width, height - 1, ch)
rectangle(5,6, "*")
```

Now run this to see more bread crumbs.

```
unix> python rectangle.py
ch = *, height = 6, width = 5
*****
ch = *, height = 5, width = 5
*****
ch = *, height = 4, width = 5
*****
ch = *, height = 3, width = 5
*****
ch = *, height = 2, width = 5
*****
ch = *, height = 1, width = 5
*****
unix>
```

We can now begin to see what has happened. The call to `rectangle(5, 6, "*")` causes a line of stars to print, then it spawns a call to `rectangle(4, 6, "*")`. The call to `rectangle(4, 6, "*")` causes a line of stars to print, then it spawns a call to `rectangle(3, 6, "*")`. This continues until we have a call to `rectangle(1,6,"*")` which spawns a call to `rectangle(0, 6, "*")`

We have now arrived at what is called the *base case* of our recursive function. Since the height is now 0, this function immediately returns. Where are we now?

We are at the end of the call `rectangle(1, 6, "*")`. Since we already printed the stars and we made the function call inside, there is no more code. As a result, the function does a tacit return. This function now returns to the call `rectangle(2, 6, "*")`. This process continues until the original call returns and we are back in the main routine. The one function call in the main routine was done, so the main routine returns and our program's execution is over. What is left over? Rows of stars.

How about printing a list? Here is a tip: to print a list, print do nothing if the list is empty; otherwise print the first element, and then the rest of the elements. If you think this way, the recursive recipe is simple.

```
def print_list(x):
    if x == []:
        return
    first = x[0]
    rest = x[1:]
    print(first)
    print_list(rest)
def main():
    test = [1, 2, 3, 4, 5, 6]
    print_list(test)
main()
```

Programming Exercises

1. Move one *entire* line of code so the list prints in the reverse order. Can you also find another way to print the list backwards?
2. Use `os.listdir` to get the contents of your `cwd`. Use `print_list` to print it out.
3. Can you make it print in asciicographical order? reverse asciicographical order?
4. Can you write a recursive function that computes the sum of a list of numbers? the concatenation of a list of strings?
5. Can you write a function that accepts a list of numbers and returns its product?

3.6 The Standard Library

A *module* is a Python program. Modules in the standard library consist largely of functions and constants. There are a plethora of these that accomplish a wide array of very useful tasks. We will learn here how to use a standard

library module and how to read its documentation. We shall begin with the library `math`, and then take a peek into the very useful `os` module. Open a Python session. Use the `import` statement to make a library visible.

```
>>> import math
```

To see its contents, use the built-in function `dir`.

```
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh',
'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp',
'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

You can see some familiar looking things. Here are some constants. Note that they are prefixed by `math.`.

```
>>> math.e
2.718281828459045
>>> math.pi
3.141592653589793
```

Here we test-drive the logarithmic functions

```
>>> math.log2(1024)
10.0
>>> math.log10(1000)
3.0
>>> math.log(math.e)
1.0
```

Now let us see help on these functions.

```
>>> print(math.log.__doc__)
log(x[, base])

Return the logarithm of x to the given base.
If the base not specified, returns the natural logarithm (base e) of x.
>>> print(math.log2.__doc__)
log2(x)

Return the base 2 logarithm of x.
```

```
>>> print(math.log10.__doc__)
log10(x)

Return the base 10 logarithm of x.
```

We see a mysterious function log1p. Let us plumb the depths.

```
>>> print(math.log1p.__doc__)
log1p(x)

Return the natural logarithm of 1+x (base e).
The result is computed in a way which is accurate for x near zero.
```

The thoughtful makers of the standard libraries provided docstrings for their functions. When in doubt about a function, this is a quick handy reference. Don't be shy.

Programming Exercises

1. Write a function `sind` that computes sines using degree angle measure. Do the same for `cos` and `tan`. Make the best available use of standard library functions.
2. What do `floor` and `ceil` do?
3. For a how large an argument can you compute `math.factorial`?

3.6.1 Accessing the File System

Let us learn about another standard library, `os`. This is very handy for interacting with your file system. Begin by importing it and viewing its contents.

```
>>> import os
>>> dir(os)
['CLD_CONTINUED', 'CLD_DUMPED', 'CLD_EXITED', 'CLD_TRAPPED',
'EX_CANTCREAT', 'EX_CONFIG', 'EX_DATAERR', 'EX_IOERR', 'EX_NOHOST',
'EX_NOINPUT', 'EX_NOPERM', 'EX_NOUSER', 'EX_OK', 'EX_OSERR', 'EX_OSFILE',
'EX_PROTOCOL', 'EX_SOFTWARE', 'EX_TEMPFAIL', 'EX_UNAVAILABLE', 'EX_USAGE',
'F_LOCK', 'F_OK', 'F_TEST', 'F_TLOCK', 'F_ULOCK', 'MutableMapping',
'NGROUPS_MAX', 'O_ACCMODE', 'O_APPEND', 'O_ASYNC', 'O_CLOEXEC', 'O_CREAT',
'O_DIRECTORY', 'O_DSYNC', 'O_EXCL', 'O_EXLOCK', 'O_NDELAY', 'O_NOCTTY',
'O_NOFOLLOW', 'O_NONBLOCK', 'O_RDONLY', 'O_RDWR', 'O_SHLOCK', 'O_SYNC',
'O_TRUNC', 'O_WRONLY', 'PRIO_PGRP', 'PRIO_PROCESS', 'PRIO_USER', 'P_ALL',
'P_NOWAIT', 'P_NOWAITO', 'P_PGID', 'P_PID', 'P_WAIT', 'RTLD_GLOBAL',
'RTLD_LAZY', 'RTLD_LOCAL', 'RTLD_NODELETE', 'RTLD_NOLOAD', 'RTLD_NOW',
```

```
'R_OK', 'SCHED_FIFO', 'SCHED_OTHER', 'SCHED_RR', 'SEEK_CUR', 'SEEK_END',
'SEEK_SET', 'ST_NOSUID', 'ST_RDONLY', 'TMP_MAX', 'WCONTINUED', 'WCOREDUMP',
'WEXITED', 'WEXITSTATUS', 'WIFCONTINUED', 'WIFEXITED', 'WIFSIGNALED',
'WIFSTOPPED', 'WNOHANG', 'WNOWAIT', 'WSTOPPED', 'WSTOPSIG', 'WTERMSIG',
'WUNTRACED', 'W_OK', 'X_OK', '_DummyDirEntry', '_Environ', '__all__',
'__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', '_dummy_scandir', '_execvpe',
'_exists', '_exit', '_get_exports_list', '_putenv', '_spawnvef',
'_unsetenv', '_wrap_close', 'abort', 'access', 'altsep', 'chdir',
'chflags', 'chmod', 'chown', 'chroot', 'close', 'closerange', 'confstr',
'confstr_names', 'cpu_count', 'ctermid', 'curdir', 'defpath',
'device_encoding', 'devnull', 'dup', 'dup2', 'environ', 'environb',
'errno', 'error', 'execl', 'execle', 'execlp', 'execlpe', 'execv',
'execve', 'execvp', 'execvpe', 'extsep', 'fchdir', 'fchmod', 'fchown',
'fdopen', 'fork', 'forkpty', 'fpathconf', 'fsdecode', 'fsencode', 'fstat',
'fstatvfs', 'fsync', 'ftruncate', 'get_blocking', 'get_exec_path',
'get_inheritable', 'get_terminal_size', 'getcwd', 'getcwdb', 'getegid',
'getenv', 'getenvb', 'geteuid', 'getgid', 'getgrouplist', 'getgroups',
'getloadavg', 'getlogin', 'getpgid', 'getpgrp', 'getpid', 'getppid',
'getpriority', 'getsid', 'getuid', 'initgroups', 'isatty', 'kill',
'killpg', 'lchflags', 'lchmod', 'lchown', 'linesep', 'link', 'listdir',
'lockf', 'lseek', 'lstat', 'major', 'makedev', 'makedirs', 'minor',
'mkdir', 'mkfifo', 'mknod', 'name', 'nice', 'open', 'openpty', 'pardir',
'path', 'pathconf', 'pathconf_names', 'pathsep', 'pipe', 'popen', 'pread',
'putenv', 'pwrite', 'read', 'readlink', 'readv', 'remove', 'removedirs',
'rename', 'renames', 'replace', 'rmdir', 'scandir',
'sched_get_priority_max', 'sched_get_priority_min', 'sched_yield',
'sendfile', 'sep', 'set_blocking', 'set_inheritable', 'setegid', 'seteuid',
'setgid', 'setgroups', 'setpgid', 'setpgrp', 'setpriority', 'setregid',
'setreuid', 'setsid', 'setuid', 'spawnl', 'spawnle', 'spawnlp', 'spawnlpe',
'spawnv', 'spawnve', 'spawnvp', 'spawnvpe', 'st', 'stat',
'stat_float_times', 'stat_result', 'statvfs', 'statvfs_result', 'strerror',
'supports_bytes_environ', 'supports_dir_fd', 'supports_effective_ids',
'supports_fd', 'supports_follow_symlinks', 'symlink', 'sync', 'sys',
'sysconf', 'sysconf_names', 'system', 'tcgetpgrp', 'tcsetpgrp',
'terminal_size', 'times', 'times_result', 'truncate', 'ttyname', 'umask',
'uname', 'uname_result', 'unlink', 'unsetenv', 'urandom', 'utime', 'wait',
'wait3', 'wait4', 'waitpid', 'walk', 'write', 'writev']
```

Wow! The abundance of things is overwhelming. Let us play with a few. Here we learn our Python process's current working directory and we list its contents.

```
>>> os.listdir()
['.pp1core.tex.swp', '_minted-p0', '_minted-p1', 'bmp', 'bol', 'bv',
'p0.aux', 'p0.log', 'p0.out', 'p0.pdf', 'p0.tex', 'p0.toc', 'p0Code',
'p1.aux', 'p1.log', 'p1.out', 'p1.pdf', 'p1.tex', 'p1.toc', 'p1Code',
```

```
'piCode', 'pp0core.tex', 'pp1core.tex', 'rectangle.py']
>>> os.getcwd()
'/Users/morrison/book/ppp'
```

Now here is something that is interesting.

```
>>> print(os.path.__doc__)
Common operations on Posix pathnames.

Instead of importing this module directly, import \texttt{os} and refer to
this module as \texttt{os.path}.  The "os.path" name is an alias for this
module on Posix systems; on other systems (e.g. Mac, Windows), os.path
provides the same operations in a manner specific to that platform, and is
an alias to another module (e.g. macpath, ntpath).

Some of this can actually be useful on non-Posix systems too, e.g.
for manipulation of the pathname component of URLs.
>>> dir(os.path)
['__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', '_get_sep', '_joinrealpath', '_varprog',
'_varprogb', 'abspath', 'altsep', 'basename', 'commonpath', 'commonprefix',
'curdir', 'defpath', 'devnull', 'dirname', 'exists', 'expanduser',
'expandvars', 'extsep', 'genericpath', 'getatime', 'getctime', 'getmtime',
'getsize', 'isabs', 'isdir', 'isfile', 'islink', 'ismount', 'join', 'lexists',
'normcase', 'normpath', 'os', 'pardir', 'pathsep', 'realpath', 'relpath',
'samefile', 'sameopenfile', 'samestat', 'sep', 'split', 'splitdrive',
'splitext', 'stat', 'supports_unicode_filenames', 'sys']
>>>
```

Look at the "is" functions. We can test and see if an item present is a directory with `isdir`. Notice we can peek inside, too, using `os.listdir`.

```
>>> os.path.exists("p1.pdf")
True
>>> os.path.isdir("p1Code")
True
>>> os.listdir("p1Code")
['.grades.py.swp', 'docstring.py', 'evilScope.py', 'grades.py']
```

This module gives you all sorts of great access to your file system. You should experiment with it and see what you can do. Be careful: you can delete files with it! Explore it and its submodule `os.path`.

Programming Exercises

1. How can you compute the size of a file?
2. Write a function called get_permission_string(file) that does the following
   - It prints out an error message if the file does not exist.
   - If the file exists, it prints its permission string (do an ls -l to remind yourself what this looks like)
3. How do you find the absolute path of a file or directory?

3.6.2 Random Thoughts

If you ever want to write a game, you will want the ability to deal with random phenomena, such as the drawing of a card from a deck or rolling dice.

The Python library `random` is a powerful tool for accomplishing this sort of task. This library has the capability of generating *pseudo-random* numbers. There is a whole industry behind the generation of random numbers. Computers actually use determininistic formulae to generate random numbers, hence the term pseudo-random.

Let us have a peek at what is inside of this module with our old friend `dir`.

```
>>> import random
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random',
    'SG_MAGICCONST', 'SystemRandom', 'TWOPI', '_BuiltinMethodType',
    '_MethodType', '_Sequence', '_Set', '__all__', '__builtins__',
    '__cached__', '__doc__', '__file__', '__loader__', '__name__',
    '__package__', '__spec__', '_acos', '_bisect', '_ceil', '_cos',
    '_e', '_exp', '_inst', '_itertools', '_log', '_pi', '_random',
    '_sha512', '_sin', '_sqrt', '_test', '_test_generator',
    '_urandom', '_warn', 'betavariate', 'choice', 'choices',
    'expovariate', 'gammavariate', 'gauss', 'getrandbits',
    'getstate', 'lognormvariate', 'normalvariate', 'paretovariate',
    'randint', 'random', 'randrange', 'sample', 'seed', 'setstate',
    'shuffle', 'triangular', 'uniform', 'vonmisesvariate',
    'weibullvariate']
>>>
```

Suppose you want to roll a pair of dice. You will want to get a tuple whose entries are random numbers 1-6. The `randint` function is just the tool for the job. Here we see its docstring.

```
>>> print(random.randint.__doc__)
Return random integer in range [a, b], including both end points.

>>>
```

The roll of a die gives us a number 1-6, so we can roll a die with a call to `random.randint(1,6)`. Let's have a look by running it a few times.

```
>>> import random
>>> random.randint(1,6)
4
>>> random.randint(1,6)
3
>>> random.randint(1,6)
1
>>> random.randint(1,6)
1
>>> random.randint(1,6)
4
>>> random.randint(1,6)
```

We can now create a function to roll a pair of dice. Create the program `dice.py`

```
import random
def roll_dice():
    return (random.randint(1,6), random.randint(1,6))
for k in range(6):
    print(roll_dice())
```

Run and see this. Your result, of course, will likely be different.

```
unix> python dice.py
(5, 6)
(4, 6)
(5, 6)
(2, 4)
(4, 6)
(1, 3)
```

Another useful method is `random.choice`, which will pick a random element out of a sequence. Here we create a function that tosses a coin.

```
def toss():
    return random.choice(["H", "T"])
```

Put this in a file and call `toss` repatedly. Two related method are `shuffle` and `sample`. We demonstrate them here.

```
>>> team = ["Pete", "Jane", "Evelyn", "Mario", "Maria", "Edward"]
>>> import random
```

```
>>> random.shuffle(team)
>>> team
['Edward', 'Evelyn', 'Jane', 'Mario', 'Maria', 'Pete']
>>> random.shuffle(team)
>>> team
['Edward', 'Jane', 'Maria', 'Pete', 'Evelyn', 'Mario']
>>> random.sample(team, 3)
['Mario', 'Evelyn', 'Edward']
>>> random.sample(team, 3)
['Maria', 'Mario', 'Evelyn']
>>> random.sample(team, 3)
['Maria', 'Evelyn', 'Edward']
```

The `sample` method picks a sample without replacement.

Programming Exercises

1. Write function that chooses a random number in $[0, 1)$ and squares it. Run this ten times and average the results. What do you see?
2. Write a function `toss_coin` that randomly returns a `"T"` or an `"H"`.
3. Write a recursive function that tosses a fair coin until a head appears and which returns the number of tosses.

3.7 Termnology Roundup

- **base case** This is a case in a recursive function that causes its ultimate return.
- **block** This is one or more lines of code that is indented a tabstop.
- **boss statement** This is a statement that controls the flow of a program. It must own a block of code, and it is a grammatically incomplete sentence.
- **call** To use a function.
- **docstring** This is a string in the first line of a function that gives information on the function's action.
- **function scope** Varibles created inside of functions and parameters of functions are invisible when the function is not executing.
- **global scope** Variables with global scope are visible at all times.
- **module** This is a file with Python code.
- **parameter** This is a value that is passed to a function.
- **pass** This is the act of sending an argument of a function to the function's code when the function is called.

- **postcondition** This is what is true when a function is done executing.
- **precondition** A condition that should inhere before we call a function
- **return value** The object, if any, that is returned by the function?
- **side-effect** This refers to actions whose consequences persist beyond the lifetime of a function.
- **worker statement** This is a Python statement containing executable code. Grammatically, it is a complete sentence.

Chapter 4

Repetition

4.0 Introduction

The goal of this chapter is to bring you to the point where Python is *Turing-Complete*, which means that, given sufficient time and memory, it can solve any solvable computational problem.

In the beginning, all Python programs were lists of worker statements that executed *in seriatum*. Then we started realizing, "If we keep doing the same thing over and over again, can't we store a procedure under a single name so we can reuse it?" This brought us to functions. Functions are objects that store sets of instructions. In fact, they are first-class objects of type `<class 'function'>`.

We then decided that our programs should be able to make decisions based on visible variables; this brings us conditional logic.

The flow of programs is no longer linear. However, if we do things right, it is structured. And the use of functions can help make programs more understandable if we choose names for our functions that are evocative of their actions.

We have actually taken the final step on the road to Turing-completeness: all repetition in programs can be done by recursion. However, the appearance of our code might be somewhat recondite and opaque. Take note, however, that recursion can be a very handy tool for solving problems that initially appear to be unwieldy impossible snarls.

Python provides two programming constructs for repetition: `while` and `for`. It also provides objects called *iterators* that walk through collections and show us objects in succession, and some python objecs are *iterables*, which means they can be walked through with a definite loop.

Once we master these ideas, Python becomes a Turing-complete language; it, given sufficient memory and time, can be used to solve any computational problems that is solvable. Let us now set out on this next exploration.

4.1 Iterables and Definite Loops

Iterables show us a collection of objects in succession. A very simple iterable is called a `range` object. If you cast a `range` object as a list, you can see all of the values it exposes.

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1,5)
range(1, 5)
>>> list(range(1,5))
[1, 2, 3, 4]
>>> list(range(2,101,7))
[2, 9, 16, 23, 30, 37, 44, 51, 58, 65, 72, 79, 86, 93, 100]
```

To do something with each value of an iterable, you use the `for` keyword as follows.

```
>>> for quack in range(10):
...     print(f"{quack}\t\t{quack*quack}")
...
0       0
1       1
2       4
3       9
4       16
5       25
6       36
7       49
8       64
9       81
```

Lurking inside of any collection (list/tuple/string/dictionary/set) is an iterator that serves its items up in some order; this object is what makes these collections iterable. For a hashed collection, the iterator serves up the items in no particualar discernable order. A dictionary iterates through its keys. The use of the `for` loop automatically brings out that feature.

```
>>> for f in os.listdir():
...     print(f"{f}\t\t{os.path.getsize(f)}\t{os.path.abspath(f)}"
...
docstring.py        200 /Users/morrison/book/ppp/p1Code/docstring.py
evil\_scope.py       78  /Users/morrison/book/ppp/p1Code/evil\_scope.py
grades.py       671 /Users/morrison/book/ppp/p1Code/grades.py
rectangle.py        215 /Users/morrison/book/ppp/p1Code/rectangle.py
```

Here we see the name, size and absolute path of each file in a directory.

Watch the `for` loop work on tuples and strings.

```
>>> t = (1,2,3,4,5)
>>> tot = 0
>>> for k in t:
...     tot += k
...
>>> print(tot)
15
```

We just found the sum of the entries in the tuple. A string's iterator walks through the string one character at a time.

```
>>> for k in s:
...     print(k*6)
...
ffffff
oooooo
oooooo
mmmmmm
eeeeee
nnnnnn
tttttt
```

A Cautionary Tale Watch this attempt to zero out a list.

```
>>> x = [1,2,3,4,5,6]
>>> for k in x:
...     k = 0
...
>>> x
[1, 2, 3, 4, 5, 6]
```

What happened? What the iterator did is assign each element in succession to the temporary name `k`, so reassigning `k` has no effect on the list itself.

Contrast that to this.

```
>>> for k in range(len(x)):
...     x[k] = 0
...
>>> x
[0, 0, 0, 0, 0, 0]
```

Here we are indexing into the list using copies of the integers starting at `0` and ending before `len(x)`. Note that the indexed entries of the list are lvalues, so we can assign to them.

The `for` loop is a *definite* loop; its purpose is to walk through a specified collection of objects, or visit all of the objects offered up by an iterable. Its "food" is an iteraable. Objects of type `range` are iterables. Lists, tuples, and strings all automatically offer their iterators when used in a `for` loop.

Two Useful Modifiers Here is clunkiness of the first class.

```
>>> for k in range(len(x) - 1, -1, -1):
...     print(x[k])
...
elephant
dingo
carical
bat
aardvark
>>>
```

This is a far better way. Use it.

```
>>> for k in reversed(x):
...     print(k)
...
elephant
dingo
carical
bat
aardvark
```

So, the `reversed` function hands you an iterator that walks backward through a collection.

Now consider this. It's kinda ugly.

```
>>> for k in range(len(x)):
    ...     print(f"x[{k}] = {x[k]}")
```

```
...
x[0] = aardvark
x[1] = bat
x[2] = carical
x[3] = dingo
x[4] = elephant
>>>
```

Now let's see the better way.

```
>>> for k, item in enumerate(x):
...     print(f"x[{k}] = {item}")
...
x[0] = aardvark
x[1] = bat
x[2] = carical
x[3] = dingo
x[4] = elephant
```

In general, you should only rarely walk through a list or tuple by traversing its indices. These two tools will help make that occasion rare. The underlying collection is never altered by either of them.

Programming Exercises

1. Can you use `reversed` and `enumerate` on a `range` object?
2. Write a loop that will produce this output

   ```
   5. aardvark
   4. bat
   3. carical
   2. dingo
   1. elephant
   ```

   Can you come up with two reasonable solutions?

4.2 File IO

Reading and writing text files in Python is achieved using the built-in `open` function. This function has one required argument, a filename. The second, optional, argument is the mode for opening the file; its default value opens a file for reading. It is recommended you open a file for reading explicitly; remember, "explicit is better than implicit." When we open a file, there is an iterator in it that can read the file line-by-line.

`r`	This is read mode. It is the default mode as well. The file you are reading from needs to exist, and you need to have read permission, or your program will error out.
`w`	This is write mode. It clobbers any existing If the file exists and you lack write permission, your program will error out. file you open. If the file does not exist, it is created.
`a`	This is append mode. It causes additional text to be appended to the end of an existing file. If the file does not exist, it gets created.
`b`	This is binary mode. It opens a file as raw bytes and can be combined with read or write mode.
`t`	This is text mode. It opens a file as text; it is the default.
`x`	This opens a file for writing but throws a `FileExistsError` if the file exists. if the file exists.

In read mode, there are several ways to access the contents of the file. Create this text file.

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!@#$%^&*()_+
,./;'[]\<>?:"{}|+
```

Now we will demonstrate some features in a live session. Let us open the file for reading.

```
>>> fp = open("sampler.txt", "r")
>>> fp
<_io.TextIOWrapper name='sampler.txt' mode='r' encoding='UTF-8'>
>>>
```

Now we take a byte.

```
>>> fp.read(1)
'a'
>>> fp.tell()
1
```

We pass the number of bytes we want to read to `read` and they are returned. In addition, the file object has inside it a pointer to the next byte it is to read. You can be told that byte by using `tell`. You can move to any byte by using `seek`.

```
>>> fp.seek(10)
10
>>> fp.read(1)
'k'
```

To go back to the beginning of the file, use `seek(0)`. To read the rest of the file, pass no argument like so.

```
>>> fp.seek(0)
0
>>> fp.read()
'abcdefghijklmnopqrstuvwxy... 789\n!@#%$^&*()_+\n,./;\'[]\\<>?:"{}|+\n'
```

In this case, you get the entire file in a single string. If the file is large, you might not want to do that. In addition to having the file pointer, the file object has its own iterator. Watch this.

```
>>> for line in fp: print(line)
...
abcdefghijklmnopqrstuvwxyz

ABCDEFGHIJKLMNOPQRSTUVWXYZ

0123456789

!@#%$^&*()_+

,./;'[]\<>?:"{}|+
```

Hey, why did this double space? Remember, `print` puts a newline at the end by default. But each line of the file has a newline at the end as well. We can suppress this annoyance as follows.

```
>>> for line in fp: print(line, end="")
...
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!@#%$^&*()_+
,./;'[]\<>?:"{}|+
```

Here is one other nifty trick.

```
>>> fp.seek(0)
0
```

```
>>> stuff = fp.readlines()
>>> stuff[0]
'abcdefghijklmnopqrstuvwxyz\n'
>>> stuff[1]
'ABCDEFGHIJKLMNOPQRSTUVWXYZ\n'
>>> len(stuff)
5
```

The `readlines` method returns a list of strings each containing a line of the file in seriatum.

Now, let us turn to writing files. The `w` mode corresponds to C's and UNIX's write mode for `open` and `fopen`. If you open an existing file for writing it will be clobbered. You must use append mode (`a`) to open a file and to add text to it. Both write and append mode will create the file if it does not yet exist. Let us now create a file in an interactive session.

```
>>> out\_file = open("bilge.txt", "w")
>>> out\_file.write("quack")
5
>>> out\_file.write("moo")
3
>>> out\_file.write("baa")
3
>>> out\_file.close()
>>> in\_file = open("bilge.txt", "r")
>>> print(in\_file.read())
quackmoobaa
```

What can be gleaned from this session? Firstly, the `write` method returns the number of bytes written to the file. Notice that it *does not* put a newline at the end of the byte sequence you are entering. You have to do this yourself or you will end up with a file with one long line. Also beware that the file is not saved until you close it. Why is this? FileIO in Python is buffered so that python is not pestering the kernel every time it wants to write a character to a file. It has a temporary storage place for the characters you write, and when that storage place gets full, Python ships the buffer to the file. Similarly, when you read from a file, you are actually reading from a buffer. Most kernels will do file operation a disk sector at a time. Closing the file causes the buffere to be flushed into the file, where it belongs, and it discontinues the use of certian system resources.

Know when to `flush` You can trigger this manually with the `flush` method. Observe this.

```
>>> fout = open("dragons.slay", "w")
>>> fout.write("Bart, eat my shorts")
19
>>> fout.write("NOW")
3
```

Now, during this session, do this

```
unix> cat dragons.slay
```

and you will see that the file is empty! Now do this.

```
>>> fout.flush()
```

```
unix> cat dragons.slay
Bart, eat my shortsNOW
unix>
```

Most of the time you never need to flush, because if you do this

```
>>> fout.close()
```

the buffer is automatically flushed.

4.2.1 A Helpful Tool: Raw Strings

Python supports a version of strings called *raw strings*. To make a raw string literal, just prepend with an `r`. When Python encounters a raw string, all backslashes are read literally. No special meaning is given them by the language. This interactive session shows how it works.

```
>>> path = 'C:\nasty\mean\oogly'
>>> print (path)
C:
asty\mean\oogly
>>> path = r'C:\nasty\mean\oogly'
>>> print (path)
C:\nasty\mean\ugly
>>>
```

Notice that in the raw string, the `\n` did not expand to a newline; it was a literal backslash-n. This is a great convenience when dealing with file paths in Windoze and for writing regular expressions. You can also make a triple-quoted string a raw string.

Warning! You may not end a raw string with a \. This causes the close-quote to be escaped to a literal character and causes a string-delimiter leak. Think for a moment: there is an easy work-around for this!

4.3 Some FileIO Applications

Let us try to imitate the UNIX command `cat`, which puts a file to the screen. We begin by developing an outline for what we want to do.

```
# The filename we wish to cat should be a command-line argument
# It will be argv[1].
# We will open this file for reading
# read the contents
# put them to the screen
# finish up by closing the file.
```

Now, let's get started with the command-line arguments.

```
# The filename we wish to cat should be a command-line argument
from sys import argv
filename = argv[1]
# We will open this file for reading
# read the contents
# put them to the screen
# finish up by closing the file.
```

Now, let's open the file and aspirate its contents.

```
# The filename we wish to cat should be a command-line argument
from sys import argv
filename = argv[1]
# We will open this file for reading using the \texttt{with} statement
with open(filename, "r") as fp:
# read the contents
    s = fp.read()
# put them to the screen
```

Now let's finish up.

```
# The filename we wish to cat should be a command-line argument
from sys import argv
filename = argv[1]
# We will open this file for reading
```

```
with open(filename, "r") as fp:
# read the contents
    s = fp.read()
# put them to the screen
print(s)
```

You think we are done? Think again. It's time to run this raw program.

```
from sys import argv
filename = argv[1]
with open(filename, "r") as fp:
    s = fp.read()
print(s)
```

Create this little test file, `simple.txt`

```
Here is a file
with a little text in it.
We are going to use this to see how
well our at program works.
```

Now run our program on it.

```
unix> python cat.py simple.txt
Here is a file
with a little text in it.
We are going to use this to see how
well our at program works.
```

It looks great, eh? Now let us do this.

```
unix> python cat.py
Traceback (most recent call last):
  File "cat.py", line 2, in <module>
    filename = argv[1]
IndexError: list index out of range
```

Uh oh. That idiotic end user. Let us now fend off this form of folly with a little parry.

```
from sys import argv
if len(argv) < 2:
    print("Usage:  python cat.py filename; enter a filename")
    quit()
```

```
filename = argv[1]
with open(filename, "r") as fp:
    s = fp.read()
print(s)
```

Now let's run this.

```
python cat.py
Usage:  python cat.py filename; enter a filename
```

Think we are in the clear? Check this out.

```
MAC:Thu Dec 06:14:23:ppp> python cat.py not.real
Traceback (most recent call last):
  File "cat.py", line 6, in <module>
    fp = open(filename, "r")
FileNotFoundError: [Errno 2] No such file or directory: 'not.real'
```

We ran this, giving a file that fails to exist. There are two ways to handle this. One way is to take exception.

```
from sys import argv
if len(argv) < 2:
    print("Usage:  python cat.py filename; enter a filename")
    quit()
filename = argv[1]
try:
    with open(filename, "r") as fp:
        s = fp.read()
    print(s)
except FileNotFoundError:
    print("File {0} does not exist.".format(filename))
    quit()
```

```
unix> python cat.py not.real
File not.real does not exist.
```

Another way is to use `os.path.exists`

```
from sys import argv
import os
if len(argv) < 2:
    print("Usage:  python cat.py filename; enter a filename")
    quit()
```

```
filename = argv[1]
if not os.path.exists(filename):
    print("File {0} does not exist.".format(filename))
    quit()
with open(filename, "r") as fp:
    s = fp.read()
print(s)
```

Feel safe? Not yet. One more form of nonsense can occur.

```
unix> chmod 000 simple.txt
```

We just revoked all read, write, and execute privileges for this file. Now watch this.

```
unix> python cat.py simple.txt
Traceback (most recent call last):
  File "cat.py", line 10, in <module>
    fp = open(filename, "r")
PermissionError: [Errno 13] Permission denied: 'simple.txt'
```

We could take exception and handle this that way. Or, here is another useful too, `os.access`. Let's do this.

```
unix> chmod 644 simple.txt
```

Now start Python in interactive mode.

```
>>> os.access("simple.txt", os.F_OK)
True
>>> os.access("simple.txt", os.R_OK)
True
>>> os.access("simple.txt", os.W_OK)
True
>>> os.access("simple.txt", os.X_OK)
False
```

The first line tests if the file exists. It does. The second, third and fourth check for read, write and execute permission.

```
from sys import argv
import os
if len(argv) < 2:
    print("Usage:  python cat.py filename; enter a filename")
```

```
    quit()
filename = argv[1]
if not os.path.exists(filename):
    print("File {0} does not exist.".format(filename))
    quit()
if not os.access(filename, os.R_OK):
    print("You lack permission to read file {0}.  Bailing....")
    quit()
with  open(filename, "r") as fp:
    s = fp.read()
print(s)
```

Now revoke all permissions and run

```
unix> chmod 000 simple.txt
MAC:Thu Dec 06:15:09:ppp> python cat.py simple.txt
You lack permission to read file {0}.  Bailing....
```

Our suit of armor is complete.

Programming Exercises In these exercises, you will learn how to imitate the behavior of various UNIX commands for file processing. You will need to prowl the `os` and `os.path` documentaton to solve this problems.

1. Write a program named `ls.py` which accepts a filename as a command line argument. If the file is a regular file, display the file's name. If the file is a directory, list the directory's contents. If the file does not exist or cannot be read, emit an appropriate nastygram.
2. Write a function named `p_string(filename)` which shows the permission string for a file, and which takes appropriate action if the file does not exist. Example: if a file has 644 permissions and is not a directory its permission string is `"-rw-r-r-"`. If a file is a directory and it has 711 permissions, its permission string is `"drwx-x-x"`.
3. Write a program show_sizes.py that accepts a commmand-line argument that is a file (regular or directory) and which displays the file name with its size if it is a regular file and which displays the contents and their sizes it it is a directory.
4. Write a program `copy.py` that copies a donor file to a recipient file. Give appropriate error warnings if something goes awry.
5. Open a file for reading and the invoke the `readlines` method. What does it do?
6. Write a program named wc.py that counts the number of characters, lines, and words in a file. Test it against UNIX's wc.

7. Write a program named `grep.py` that takes as arguments a string and a filename and which puts all of the lines of the file containing the specified string to `stdout`.

4.4 `while` and Indefinite Looping

Python has a second looping construct, `while`. Let us begin by showing an very simple example.

```
def main():
    x = input("Enter a number  ")
    x = int(x)
    while x < 100:
        print("The number you entered, {}, is less than 100".format(x))
        x = int(input("Enter a number  "))

    print("You finally entered {} and it is at least 100.".format(x))
main()
```

This is an example of a "nag" loop that will keep asking until the user does the desired thing.

Indefinite loops can be dangerous. It is very easy to have an "infinite loop," which just keeps running until the OS or the user calls the process running it to halt. Look at this little program.

```
x = 5
while x < 10:
    print(x)
    x -= 1
```

The value of `x` will keep marching farther and farther from resolution. This loop will just keep printing a countdown to the screen. If you run this, use control-C to stop it. This loop causes "spewing;" to wit, it causes great volumes of text to keep streaming into `stdout`.

Loops can also "hang;" in this event, the program simply sits there doing nothing. You can kill a hung or spewing program with control-C.

Here is a loop that is guaranteed to hang.

```
x = 1
while x > 0:
    x += 1
```

Programming Exercises Wrap these solutions in functions.

1. Write a `while` loop that prints out the entries in a list.
2. Repeatedly roll a pair of dice until you get doubles. Print the rolls as they occur. If a double 1 occurs, print `"Snake Eyes!` and if a double 6 occurs, print `"Boxcars!`
3. Repeatedly toss a coin until you get five heads in a row. return the tosses in a string like this: `"HTHHHTTTHTTTHHHTTTTTHHHHH"`.

4.5 Programming Projects

Here you will use your skills to perform simulations of two stochastic systems. One will introduce you to `stopping times`, which entail experiments that are repeated until a specified event occurs.

The other is an analysis of risk in the Parker Brothers' game Monopoly. Everyone should love the orange monopoly.

Project 1: System Simulation for a Waiting Time In the first project, you will perform a simulation of a stochastic (random) system. In this project, you are going to perform a simulation in which you toss a fair coin until a head appears. You are going to run this experiment one million times and maintain a tally of how many times the first head came up on the first toss, second toss..., etc. An outline of suggested functions for you to implement is shown.

1. Implement the function toss_coin, which produces an "H" or a "T" at random.
2. Implement the function first_head, which tosses a fair coin until a head appears and returns the number of that trial.
3. Implement the function perform_sim(num_trials) that returns a dictionary whose keys are integers and whose values are the number of times that result was returned by repeated trials of `first_Head`. This dictionary just amouts to a tally of the trials.
4. Run this first_Head for one million trials. What do you see? What seems to happen as you double the number of trials?

Project 2: The Orange Monopoly Simulation In this next project, you are playing Monopoly and your client is on the square Just Visiting/Jail, and is just visiting. His opponent has hotels on the New York Avenue Monopoly. You have the actuarial duty of determining the price for an insurance policy to indemnify your client against the cost of the visiting the hotels for one turn.

To do this project, you will need to find an image of a Monopoly board. Here are the pertinent rules. The rent for New York Avenue is $1000. The rents for St. James and Tennessee Ave are $950. You do not need to concern yourself with the other squares, save for Go to Jail, which ends your turn and puts you in jail.

To start a turn, you roll a fair pair of dice (6-sided) and advance that number of squares. If you land on a property with a hotel, you must pay the applicable rent. If you roll doubles, you roll again and the same rules apply. If you roll doubles three times in a row, you go to straight to jail for speeding and your turn ends. In this case, you never land on the square you rolled for and are not obliged to pay rent theere.

So it's possible to have `#$!$!@#` luck and roll double threes, hit St. James, then roll a 1 and a 2 and hit New York Avenue for a total damage of $1,950.

Here are some suggestions for how to proceed.

1. Write a function that produces a tuple with two fair die rolls in it.
2. Make a tuple that represents all reachable squares in one turn.
3. Write a function that peforms the rolls in a single turn and which returns the total damage from the hotels in that turn.
4. Write a function `doTrials(n)` that accepts an integer as an argument, and which computes the average damage from the hotels in `n` trials.
5. Run a ton of trials and see what the average damage is. How would you price this policy?
6. Can you do a big trial, keep running averages in a file, and plot the results?

Project 3: Gambler's Ruin Suppose we have two gamblers and they have a total of $M (an integer) between them. They repeatedly play a game until one of them goes bust; each trial of the game has win probablility p for Player One and $1-p$ for Player Two. For $M = 10$, perform simulations of this experiment. What do the probabilities of bust look like for Player one if his initial stake is $ k, $1 < k < 10$? What is the average time to bust in each of these cases?

4.6 Function Flexibility

When you define a function such as this one

```
def f(x, y, z):
    return x + 2*y + 3*z
```

the arguments `x`, `y`, and `z` are called `positional arguments`. This is so because when you call the function like this

```
print(f(1,2,3))
```

the arguments are sent, in order to `f`. Notice that the position of each argument is critical. Permute them and the result is not the same.

You have seen some nifty stuff that make functions flexible and which expand their purpose, but now the time has arrived to for you to be able to use these things yourself.

Let us begin with an example. Consider the function `math.log`. This function can be called as follows.

```
>>> math.log(10)  #natural log
2.302585092994046
>>> math.log(1000, 2)
9.965784284662087 #log base 2
```

You see an optional second argument. How did they do this? We can make it happen. We will create a function called `lincoln` that does the same thing.

```
import math
def lincoln(x, b = math.e):
    return math.log(x)/math.log(b)
print(lincoln(1000,2))
print(math.log(1000, 2))
```

The second argument has a default value of `math.e`. Notice that at least one argument is required.

Here is another example. We create the illusion that this function can have as many as five arguments.

```
def product(a = 1, b = 1, c = 1, d = 1, e = 1):
    return a*b*c*d*e
print("product()  = ", product())
print("product(3)  = ", product(3))
print("product(3, 4) =", product(3, 4))
print("product(3, 4, 5) =", product(3, 4, 5))
print("product(3, 4, 5, 6) =", product(3, 4, 5, 6))
print("product(3, 4, 5, 6, 7) =", product(3, 4, 5, 6, 7))
```

If you take the default values away from the first two arguments, then at least two arguments are required to call this function. You should try this now.

End of List Rule All default arguments for any function must be grouped together at the end of the argument list. Something like this is illegal.

```
def dumb(x, y = "cows", z):
    pass
```

Run this program and see the error message. This is because these arguments are really polymorphic. If you pass a value to them, they behave as positional arguments. If you don't the default value is used. It does not take a great deal of imagination to see why the End of List Rule is necessary.

4.6.1 A Star is Born

Now let us consider the `print` function. You have noticed this sort of flexibility in its use.

```
    print("Foo", "Bar", "Baz", sep="moo", end="cats\n")
```

It seems that this function accepts an unlimited number of comma-separated arguments, and they allows you to specify behavior at the end of the argument list using keywords. We might like to have this particular arrow in our quiver, so let us set about getting it.

Consider the problem of finding writing a function to find the sum of a a bunch of numbers whose call looks like this.

```
>>> total(2,3,6,8)
19
```

To solve it, let us see if we can plagiarize `print`'s mechanism. Here is what `print`'s header looks like.

```
def print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False):
```

Its first argument is a *star argument* or *stargument*. A stargument must appear after all other positional arguments.

Now let's make `total`. We will use a stargument.

```
def total(*x):
    out = 0
    for k in x:
        out += k
    return out
print(total(2, 3, 6, 8))
```

You might want to require at least one argument be passed to `total`. We can enforce this by adding a postional argument at the beginning. We do this in function `totall`.

```
def total(*x):
    out = 0
    for k in x:
        out += k
    return out
def total1(y, *x):
    out = y
    for k in x:
        out += k
    return out

print(total(2,3,6, 8))
print(total1(2,3,6, 8))
print(total())
print(total1())
```

Note the opprobrious ululation after the last call. At least one number is required.

```
unix> python keywords.py
19
19
0
Traceback (most recent call last):
  File "keywords.py", line 15, in <module>
    print(total1())
TypeError: total1() missing 1 required positional argument: 'y'
unix>
```

Let us now make a simple function to count the number of files in a directory with a given extension that uses all default arguments. For defaults, we will have the directory be the `cwd` and the extension be `.txt`.

```
import os
from sys import argv
def countFiles(directory=".", end="txt"):
    files = os.listdir(directory)
    out = 0
    for item in files:
        if item.endswith("." + end):
            out += 1
    return out
folder = "." if len(argv)== 1 else argv[1]
ext = "" if len(argv) < 3 else argv[2]
print(folder)
print("There are {0} files in {1} with extension .{2}".format( countFiles(directory=fo
```

Programming Exercise Modify the output routine so that no mention of extension is made if `argv[2]` does not exist and all files are counted.

4.6.2 Keyword Arguments

Now let us see how to use keyword arguments. Recall that `print`'s header looks like this.

```
def print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False):
```

Its first argument is a stargument. Following it are several keyword arguments. Each has the form `keyword=value`. Each of the specified values is the default for that argument. If no value is passed to a keyword argument, its default value is used.

Order in the court! You can have positional, star, and keyword arguments in a function. You must obey these rules

1. Positional arguments come first.
2. One stargument can come next.
3. Keyword arguments must all occur at the end.

Here is a cheesy example of these rules at work.

```
def f(*x, y="cows", z = "horses"):
    return "{}{}{}".format(sum(x), y ,z)
def g(a, b, *x, y="cows", z = "horses"):
    return "{}{}{}{}{}".format(a, b, sum(x), y, z)
print (f(2,3,4,5,y="rhinos", z="pigs"))
print (g("moo", "baa", 2,3,4,5,y="rhinos", z="pigs"))
```

4.7 Generators

A *generator* is a stateful function that remembers its local symbol table between calls. You will meet a new keyword `yield` when creating genearators, which returns a value to the caller without destroying the stack frame containing the function. Between calls, this object remembers where it left off and it remembers the values of local variables. Note that the scope of these local variables is still confined to the body of the generator. Let us see this at work in a very simple example.

Each time a generator is called it can either yield a value, in which case it can be called again, or it can return a value, which termimnates its execution. We begin with a super-simple example, and we see how a generator is an iterable.

```
def simple():
    yield "quack"
    yield "moo"
    yield "baa"
    yield "neigh"
    yield "woof"
s = simple()
for k in s:
    print(k)
```

Now we run it.

```
unix> python simple.py
quack
moo
baa
neigh
woof
```

Generators can iterate through finite or infinite sets, as we shall soon see. Let's make one that starts counting at 1. Generators are a handy form of iterable.

```
def counter(n):
    k = 0
    while k <  n:
        k += 1
        yield k

for k in counter(10):
    print(k)
```

Now run it.

```
unix> python counter.py
1
2
3
4
5
```

```
6
7
8
9
10
```

Here is an imitation of `range`.

```
def strange(start=0, stop = 10, skip = 1):
    while start <  stop:
        yield start
        start += skip

for k in strange(0, 11, 2):
    print(k)
for k in strange(0, 1, .1):
    print("{0:.2f}".format(k))
```

```
unix> python strange.py
0
2
4
6
8
10
0.00
0.10
0.20
0.30
0.40
0.50
0.60
0.70
0.80
0.90
1.00
```

Generators can serve up infinite sequences. For example, we will create a generator here that serves up the nonnegative integers.

```
def z():
    out = 0
    while True:
        out += 1
        yield out
```

```
progression = z()
for k in progression:
    print(k)
```

Now we run it. Be prepared to hit control-C to stop the spewage.

```
unix> python z.py
0
1
2
...
```

```
(big number)
```

You will see that the value of the local variable `out` is rememembered between calls. This will iterate through all of the positive integers.

This beast requires some taming. You could put logic into `z` to tell it to stop, but here is a more flexible way. We pass our generator to a function that does the dirty work for us.

```
def z():
    out = 0
    while True:
        out += 1
        yield out
def stopper(generator, n):
    well = z()
    for k in well:
        if k < n:
            yield k
        else:
            return

progression = z()
for k in stopper(progression, 10):
    print(k)
```

Now we run it.

Now we run it.

```
unix> python z.py
1
2
3
4
5
6
7
8
9
```

The *stopper* routine can be recycled.

```
def z():
    out = 0
    while True:
        out += 1
        yield out
def squares():
    out = 0
    while True:
        out += 1
        yield out*out
def stopper(generator, n):
    well = generator
    for k in well:
        if k < n:
            yield k
        else:
            return

progression = squares()
for k in stopper(progression, 200):
    print(k)
```

Now we run it.

```
unix> python z.py
1
4
9
16
25
```

```
36
49
64
81
100
121
144
169
196
```

4.7.1 Holy Iterable, Batman!

A generator is an iterable! Here is proof.

```
def squares():
    out = 0
    while True:
        out += 1
        yield out*out
def stopper(generator, n):
    well = generator
    for k in well:
        if k < n:
            yield k
        else:
            return
s = squares()
for k in range(10):
    print(next(s))
```

Now we run it.

```
unix> python z.py
1
4
9
16
25
36
49
64
81
100
121
```

```
144
169
196
```

You can also make generators on-the-fly using a mechanism akin to a comprehension. Here is an example.

```
>>> s = (x*x*x for x in range(1,10))
>>> for k in s:
...     print(k)
...
1
8
27
64
125
216
343
512
729
```

Take note that once you do this, the generator is "used up." Watch this; we are iterating with it again.

```
>>> for k in s:
...     print(k)
...
>>>
```

The solution? Just make another one. Here is another way to use it. We will begin by reconstituting it.

```
>>> s = (x*x*x for x in range(1,10))
>>> next(s)
1
>>> next(s)
8
>>> next(s)
27
>>> next(s)
64
>>>
```

What happens if you go too far? Watch.

```
>>> next(s)
125
>>> next(s)
216
>>> next(s)
343
>>> next(s)
512
>>> next(s)
729
>>> next(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

This `next` function is handy if a generator generates an infinite progression. We create an example with Fibonacci numbers.

```
def fib():
    little = 1
    big = 0
    while True:
        little, big = big, little + big
        yield little
f = fib()
for k in range(10):
    print(next(f))
```

Now run this.

```
unix> python fibonacci.py
0
1
1
2
3
5
8
13
21
34
```

4.8 Terminology Roundup

- **definite loop** This is a loop that walks through a collection or an iterable, executing a block of code for each item. Python implements this with the `for` construct.
- **indefinite loop** This is a loop that repeats until its predicate becomes false. Python implements this with `while`
- **iterable** This is an object that can be walked through using a `for` loop. All Python collections are iterables that are walked through an item at a time. Python strings are walked through a character at a time. Generators and range objects are also iterables.
- **keyword argument** This is named argument given at the end of a function's argument list. Examples include `sep` and `end` in `print`.
- **star argument** This is an argument in a function that is preceded by a star (*), which behaves like an array inside of a function.
- **stargument** Synonym of for star argument.
- **Turing-Complete** This describes a full-featured computer language capable of solving any computational problem, given sufficient time and memory.

Chapter 5

Algorithms

5.0 Introduction

Recall that an algorithm is a precise recipe for carrying out a task. We have a wide variety of tools at our disposal for accomplishing computational tasks in Python. We have looping, conditional logic and Python's smart objects at our disposal that can easily automate complex procedures. The purpose of this chapter is to allow us to flex these new muscles.

5.1 A Rough Measure of Growth

Suppose that f and g are functions. We will say that $f = O(g)$ as $x \to \infty$ if for sufficiently large x, we have some constant M so that

$$|f(x)| \leq M|g(x)|,$$

for x sufficiently large.

To say this is simple English, f is at most proportional to g when x gets big. This symbol O is called a *Landau symbol*, after the mathematician Edmund Landau, who invented it along with Paul Bachmann. We will use this notation to describe the time/space efficency of algorithms.

5.2 Searching

Consider the task of searching a list for a particular item. If the list is unsorted and contains n elements, we will just have to walk through the list, checking if each element is the desired element we seek (the quarry). At most we will have

to do n checks. If the item is not present, we end up checking the whole list to no avail.

We see that this procedure takes an amount of time at worst proportional to the size of the list, so it is an $O(n)$ or a *linear-time* algorithm. Note that when the search field's size doubles, the time and memory it takes to do the search also doubles.

Programming Exercise Use a loop to write `contains(c, quarry)`, where `c` is a list or tuple and `quarry` is an object. Have it return `True` if `quarry` is present in `c` and `False` otherwise.

5.2.1 Binary Search

Later in this chapter, we will discuss sorting, but here will discuss a smart scheme for searching a sorted list. This method will be quicker than the linear search you just wrote because you will not need to inspect every element in the list.

Suppose we have a list that is in sorted order. Here is a scheme for searching it.

1. Look at the item halfway into the list.
2. if this item is your quarry, report `True` and you are done. If this item is is before your quarry, discard the first half of the list; the quarry, if present, is in the second half. If the item is after the quarry, the item is in the first half of the list.
3. Repeat. If you end up discarding the entire list and never find the quarry, return `False`

This method is called *binary search*.

Suppose your list contained a million elements. After each pruning of the list, we have the following largest size of the remaining list.

```
 1 1000000
 2 500000
 3 250000
 4 125000
 5 62500
 6 31250
 7 15625
 8 7813
 9 3907
10 1954
```

Figure 5.1: Fast-Moving Banana Slug

```
11 977
12 489
13 245
14 123
15 62
16 31
17 16
18 8
19 4
20 2
21 1
```

This procedure requires a maximum of 20 steps. Each time, the search field is cut in half. So, all we are really asking is: How many times do we need to divide a number by 2 until it is reduced to 1 or 0? This would be $\log_2(n)$. So the time for this procedure to do its job is $O(log(n))$. We say that such an algorithm is a *log-time* algorithm.

You will notice an absence of a base in the logarithm; this is because all logarithms are proportional to each other. The reason? Apply the change-of-base formula for logs. We will use the notation log for the natural log function.

We all know from Ren and Stimpy that logs are big, heavy and wood. They are also very slow-growing functions. So sorting a list makes finding things in it worlds faster. You can search a sorted list with a billion elements in 30 checks. That's a whole lot better than a billion checks.

Programming Exercise Implement this algorithm in Python. Load the Scrabble dictionary into a list using `readlines`. Then write a procedure to search it for a word, returning `True` if the word is present and `False` otherwise.

5.3 Root Finding

We are going to describe an algorithm for finding a zero (x–intercept) of a continuous function that changes sign on an interval. This uses a "divide and conquer" mechanism similar to that of the binary search algorithm you just wrote. We need a one useful fact, the Weierstraß Intermediate value theorem.

Theorem. Let $[a, b]$ be a closed, bounded interval and f be a continuous real-valued function defined on this interval. If f changes sign on the interval, then there is at least one $x \in [a, b]$ so that $f(x) = 0$.

There is an easy way to check if two numbers x and y are of opposite sign; just check to see if $xy < 0$.

Let us begin with an example. Suppose we are trying to approximate $\sqrt{2}$. We know that the function $f(x) = x^2 - 2$ has $\sqrt{2}$ as a root. Also we can see that $f(1) = -1$ and $f(2) = 2$, so f must have a root on $[1, 2]$.

We know that $\sqrt{2} = 1.5 \pm .5$. Now we do this. Check the midpoint; $f(1.5) = .25$ and $f(1) = -1$ so we have now bounded our root to being in $[1, 1.5]$. This means $\sqrt{2} = 1.25 \pm .25$. At each step we see that we can obtain a value that is within a tolerance of $\sqrt{2}$. When that tolerance is small enough, we can stop.

Programing Exercise Continue this calculation until you know $\sqrt{2} \pm .001$. Can you automate the process with a Pyhon program?

Note that the error is at most `(right - left)/2` where [`left`, `right`] is the interval on which we know the function changes sign.

Programming Exercise Write a function `zero(a, b, tol, f)` that finds an approximation for a root of the function `f` on $[a, b]$ with error tolerance `tol`. The preconditon for calling this function is that f changes sign on $[a, b]$.

5.4 A little number theory

Let us begin by looking at some elementary ideas of number theory. Back in your Wormwood days, you learned about long division. You learned that if a

and b are positive integers, you can write

$$b = aq + r,$$

where q is an integer called the *quotient*, r is the *remainder*, and $0 \leq r < a$. In fact, these integers q and r are unique. If the remainder r is zero, then a divides into b evenly. In this event, we write $a|b$.

Python computes these quantities easily. Let us take $b = 365$ and $a = 7$.

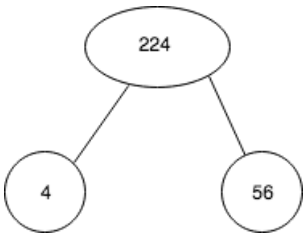
```
>>> b = 365
>>> a = 7
>>> q = b//a
>>> r = b%a
>>> a*q + r == b
True
```

So, the quotient is just obtained by integer division and the remainder is computed with our friend mod, `%`.

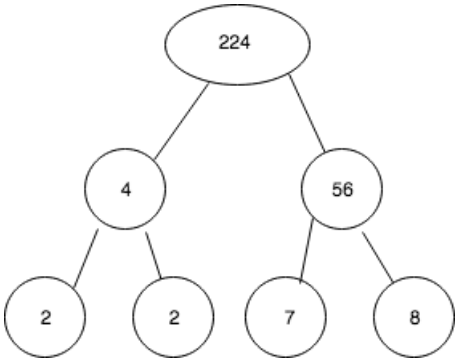
Testing for divisibility of numbers is easy. If you have integers `a` and `b`, then you can test for `a|b` with the predicate `b%a == 0`. For example, to test if a number `n` is even, you use the predicate `n%2 == 0`.

Back in your Wormwood days, you likely learned about prime numbers. The number 1 was defined not to be prime. An integer 2 or larger is prime if its only positive divisors are 1 and itself. For example, 4 is not prime, because $2|4$. More generally, a number is not prime if it can be factored into two smaller integers.

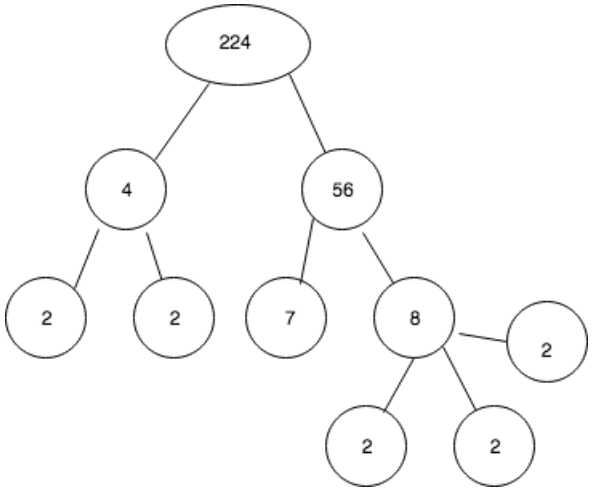
The *prime factoriaztion theorem* states that every number $n \geq 2$ can be factored uniquely into primes. Back in the day, Miss Wormwood taught about factor trees, which provide a nifty method for doing this. Here is how they worked. Suppose we want to factor the number 224. It's a cinch that 224 is divisible by 4. We begin by drawing this.



Both of the leaves on the tree can be factored into smaller integers; do so. The wonderful machinery of number theory says that the result at the end is the same no matter how you do this.



Two of the leaves on this tree now have 2s in them; they are prime. Similarly, there is nothing to be done with the 7. We will "cheat" and factor the 8 into three 2s.



Every leaf on the tree is prime; it cannot be factored further. We simply pluck up the numbers inside of the leaves of the tree. We now have this prime factorization for 224.

$$224 = 2 \cdot 2 \cdot 2 \cdot 7 \cdot 2 \cdot 2 = 2^5 \cdot 7.$$

Now we will create a function that tests positive integers for primality.

To begin, observe that the number 2 is prime because it cannot be factored into two smaller integers. However, any even number n larger than 2 satisfies $2|n$, so even numbers larger than 2 are not prime. So, we can begin writing our `isPrime` function as follows.

```
def isPrime(n):
    if n == 1:
```

```
        return False
    if n == 2
        return True
    if n%2 == 0:
        return False
    ##we are not done yet.
    return "cows"
```

Test this function out. It will return `False` if `n` is 1 or if it is an even number larger than 2. It will return `True` if `n` is 2. For all other cases, it will punt and return `"cows"`, a ridiculous value we provide to remind ourselves we are not done yet.

We could accomplish the rest as follows. We then take a couple of known primes for a spin.

```
def isPrime(n):
    if n == 1:
        return False
    if n == 2:
        return True
    if n%2 == 0:
        return False
    k = 3
    while k < n:
        if n % k == 0:
            return False
        k += 1
    return True
print(isPrime(997))
print(isPrime(10000019))
```

Ugh. We're movin' kinda slow at the junction.

```
unix> time python wasteful.py
True
True

real        0m1.980s
user        0m1.508s
sys        0m0.073s
```

This has to do a whole lot of checks before it resolves; the cost of the procedure is roughly proportional to the number you ask it to check if you pass it a prime and less otherwise.

Now here is an interesting little jewel of a fact that will make our function sweetly efficient. Suppose that you have a positive integer n and you write $n = ab$ where a and b are integers. Then at least one of a or b must satisfy the condition $k * k \leq n$.

Let us show an example. Take the number 36 and write $36 = 9 * 4$. Notice that $4 * 4 = 14 \leq 36$. If you write $36 = 6 * 6$, then both factors have a square that is at most 36. As you shall see all we care is that at least one factor has a square that is at most n.

So let us improve this. For starters, because of our preliminary tests, we can begin with `k = 3`. Also, we can discard all even numbers, so each time the loop runs let us update with `k += 2`. This will keep us testing with only odd numbers. Et Voila! We have a 50% improvement.

You should take your existing implementation of `isPrime` and integrate this improvement.

But, like the Ginsu Knife Man says, "There is more!" A while back, we learned this: if for a positive integer n we have no k so that $k * k \leq n$ and $n\%k = 0$, then n is prime! One thing we know here is that the square root of a large number is a whole lot less than the original number. This becomes more acutely so as the original number gets bigger. Let us put this together. Here is what we had.

```
def isPrime(n):
    if n == 1:
        return False
    if n == 2
        return True
    if n%2 == 0:
        return False
    ##we are not done yet.
    return "cows"
```

Now we start at `k = 3` and go up by 2 each time. The cows get the boot, because if the loop ends without finding a prime, we know that `n` is prime.

```
def isPrime(n):
    if n == 1:
        return False
    if n == 2:
        return True
    if n%2 == 0:
        return False
    k = 3
    while k < n:
```

```
        if n%k == 0:
            return False
        k += 2
    return True
```

Next, insert our other improvement.

```
def isPrime(n):
    if n == 1:
        return False
    if n == 2:
        return True
    if n%2 == 0:
        return False
    k = 3
    while k*k <= n:
        if n%k == 0:
            return False
        k += 2
    return True
```

Now let's run it.

```
unix> time python faster.py
True
True

real        0m0.073s
user        0m0.042s
sys        0m0.017s
```

Va voom!

Programming Exercises

1. Implement a function called `smallestFactor(n)` that does the following. It accepts a positive integer `n`. If `n` is 1 it just returns 1. Otherwise, it finds the first positive integer `k` so that `k|n`. Here are some handy test cases.

   ```
   smallestFactor(997) == 997
   smallestFactor(323) == 17
   smallestFactor(1) == 1
   smallestFactor(1728) == 2
   smallestFactor(1005973) == 997
   ```

Try hard to minimize the number of times the loop runs. Borrow from the ideas usedin `isPrime`.

2. Why does `smallestFactor` return prime numbers if `n >= 2`?
3. Use `smallestFactor` to implement a function called `primeFactors(n)`, which returns a list of all prime factors of the positive integer `n` passed it. If `n == 1`, return an empty list. Can you use recursion to do this?
4. Run `primeFactors` on several different cases. Why is the list it returns always in numerical order?

5.5 The Performance of `isPrime`

We just saw that there is more than one way to test a number for primality. Both ways worked, but, using a little knowledge of mathematics, we were able to achieve a huge boost in performance. Now it's time to break out some Landau symbols.

The unimproved `isPrime` function used the predicate `k < n` in its `while` loop. This loop could run as many as `(n - 1)//2` times. So, the cost of executing this procedure is at most proportional to `n`, the size of the number we are testing for primality. We would say this algorithm is $O(n)$, or *linear-time*.

The improved algorithm used the predicate `k*k < n` in its `while` loop. It could not possibly run more than $\sqrt{\texttt{n}}$ times. This makes the improved algorithm a $O(\sqrt{n})$ algorithm.

You know that when n is large, $\sqrt{n}$ is far smaller than n. So this improved algorithm is much faster than its unimproved counterpart.

5.6 Sorting using Iterative Techniques

Python performs sorting on lists as a service. However, it is important to understand how sorting works. We will study several sorting algorithms, starting with the dreadful and moving to the fleet. What we show here is just a small sampling of what is out there.

Imagine you have a deck of cards with numbers on them. Here is a possible way to sort them; it is called the *Bozo sort*. It is named after the character Bozo, who was a clown in a show that was popular from the late 1940s into the 1960s.

1. Check to see if the deck is in order; if so, you are done.
2. If not, shuffle the deck and repeat Step 1.

Let us see that this can work. Create a program called `sorts.py`. We can shuffle a list using its `shuffle` method.

First, let us write a function to see if a list is in order; let us agree on ascending order. Just reverse the inequalities in the predicates for descending order. To do this, we just walk up to each pair of elements and check if they are in order. If a pair is not, a `False` is returned.

```
def isInOrder(deck):
    for k in range(len(deck) - 1):
        if deck[k] > deck[k+1]:
            return False
    return True
```

Now let us write code to test our result. This code will shuffle a "deck", which is a list of consecutive integers, then sort with the bozo method. We will stub the bozo method in. and put our test code in.

```
import random
def isInOrder(deck):
    for k in range(len(deck) - 1):
        if deck[k] > deck[k+1]:
            return False
    return True
def bozo(deck):
    pass  #do nothing
def main():
    pass
main()
```

Running this code will not shuffle the deck. Now we add the code to do that.

```
import random
def isInOrder(deck):
    for k in range(len(deck) - 1):
        if deck[k] > deck[k+1]:
            return False
    return True
def bozo(deck):
    while(not isInOrder(deck)):
        random.shuffle(deck)

def main():
    deck = list(range(9))
    random.shuffle(deck)
```

```
    bozo(deck)
    print(deck)
main()
```

Here we are shuffling a deck with 9 items in it. Now run it. Even for a handful of elements, this works dreadfully. Remember, $9! = 362880$. This sort works on the average in $O(n!)$ time, which is terrible.

```
Tue Feb 07:13:28:ppp> time python sorts.py
[0, 1, 2, 3, 4, 5, 6, 7, 8]

real    0m4.711s
user    0m4.344s
sys 0m0.080s
Tue Feb 07:13:28:ppp> time python sorts.py
[0, 1, 2, 3, 4, 5, 6, 7, 8]

real    0m7.150s
user    0m6.993s
sys 0m0.060s
Tue Feb 07:13:29:ppp> time python sorts.py
[0, 1, 2, 3, 4, 5, 6, 7, 8]

real    0m5.379s
user    0m5.185s
sys 0m0.055s
```

We can do better.

Let us look at a sort that works in a more reasonable amount of time, the bubble sort. Suppose we have a list, and we walk up to each pair of elements in succession. If the elements are in order, do nothing. If not, switch them. After one pass, you will see that the largest element will have "bubbled" to the top.

Now we can repeat the procedure and skip checking the last element. Then the two largest elements will be at the top. We continue until the list is sorted.

Let us begin to write this procedure in a function named `bubble`. We will have two integer variables, `k` and `end`. The variable `end` will point at the index dividing the sorted and unsorted portions of the list. The variable `k` will do the walking up to the pairs. So here is how we get started.

```
def bubble(x):
    end = len(x)
    #more code
```

Each time the inner loop runs we have a `loop invariant`, which is a set of statements that is true after each execution of the loop. Our loop invariant is

this: items to the right of `end` are the largest elements in sorted order and items to the left have no guaranteed order. Yep, it's a jungle out there.

Let us make a general pass of the loop.

```
def bubble(x):
    end = len(x)
    for k in range(end) - 1):
        if x[k] > x[k+1]:
            x[k],x[k+1] = x[k+1],x[k]
    #more code
```

As is, this does one pass of the loop. At the end of each pass, `end` may be reduced by 1; let's put that in.

```
def bubble(x):
    end = len(x)
    for k in range(end - 1):
        if x[k] > x[k+1]:
            x[k],x[k+1] = x[k+1],x[k]
    end -= 1
```

When are we done? How about when `end` gets down to 1; by the process of elimination, the first element must be in place. We will need to enclose our `for` loop in another loop.

```
def bubble(x):
    end = len(x)
    while end > 1:
        for k in range(end) - 1:
            if x[k] > x[k+1]:
                x[k],x[k+1] = x[k+1],x[k]
        end -= 1
```

Let's test this.

```
import random
def isInOrder(deck):
    for k in range(len(deck) - 1):
        if deck[k] > deck[k+1]:
            return False
    return True
def bozo(deck):
    while(not isInOrder(deck)):
        random.shuffle(deck)
def bubble(x):
```

```
    end = len(x)
    while end > 1:
        for k in range(end - 1):
            if x[k] > x[k+1]:
                x[k],x[k+1] = x[k+1],x[k]
        end -= 1
def main():
    deck = list(range(9))
    random.shuffle(deck)
    bubble(deck)
    print(deck)
main()
```

That was so fast on 9 elements it was basically instant. Let's try this on 10,000 elements.

```
$ time python sorts.py

real    0m13.159s
user    0m13.011s
sys 0m0.057s
```

Here is the result on 5000 elements. It is about 1/4 as long

```
Tue Feb 07:14:08:ppp> time python sorts.py

real    0m3.509s
user    0m3.305s
sys 0m0.039s
```

If you did this with the bozo sort, the sun would likely nova first and consume the Earth before your list would ever be sorted. Even the bubble sort seems a little slow.

You might ask, "What is the big-O classification for this sort?" Let us think about it. The first time through we perform $n-1$ checks. The second time through, we perform $n-2$ checks. Each time the number of checks diminishes by 1. The total number is

$$(n-1)+(n-1)+\cdots 1=\sum_{k=1}^{n-1} k.$$

Yes, there is a formula for this,

$$\sum_{k=1}^{n-1} k=\frac{n(n-1)}{2}=O(n^2).$$

The bubble sort is an example of a *quadratic sort*.

Now we will look at two other sorting methods, insertion sort and selection sort. Both of these are quadratic sorts, but the constant of proportionality is somewhat smaller. The insertion sort works in a manner similar to that of a player picking up cards and inserting them in his hand. As he picks up each card, he places it in the appropriate place in his hand until all of the cards are inserted.

It is best if we can do our sort in-place without using additional memory. So here is an idea. If the array is empty, do nothing and return immediately. Otherwise, make a variable called `end` and have it point at index 1 in the array. The loop invariant is this: items to the left of `end` are sorted there is no assertaion about items to the right. We now do the following. Add 1 to end. Then, keep swapping the new entry with its neighbor to the left until it is in the right place (a "trickle down") procedure. We keep doing this until `end` reaches the right-hand end of the array.

Let us begin here. If the list is empty, we bail. If not, we mark `end` at `1`. Notice here that everthing to the left of `end` is sorted, because there is only one entry over there.

```
def insertion(x):
    if x == []:
        return
    end = 1
```

Now let's block in some more code.

```
def insertion(x):
    if x == []:
        return
    end = 1
    while end < len(x) - 1:
        end += 1
        #then trickle down
```

Now let us explain carefully. Suppose we have this array

```
1  3  5  9 |4  0
          (end)
```

We increment end.

```
1  3  5  9  4 |0
             (end)
```

We will keep swapping with the neighbor to the left until 4 is in the right place.

```
1  3  5  9  4 |0
1  3  5  4  9 |0
1  3  4  5  9 |0
            (end)
```

Let us do this again. Begin by incrementing `end`

```
1  3  5  9  4 |0
1  3  4  5  9  0 |
               (end)
```

Now do the swaps. WARNING: We do not want to attempt to check the left-hand neighbor of 0 when it gets to the bottom! If we do so we will be comparing with the item at the end of the list. This will make a horrid mess. We will use short circuiting to prevent this.

```
def insertion(x):
    if x == []:
        return
    end = 1
    while end < len(x) - 1:
        end += 1
        k = end
        #trickle down; notice k >= 1 condition
        while  k >= 1 and x[k] <  x[k - 1]:
            x[k], x[k - 1] = x[k - 1], x[k]
            k -= 1
```

You can put this in `sorts.py` and test it. This method is quite a bit faster than the bubble sort. Typically, unless you are very unlucky, the trickle down procedure only trickles about half-way down on average. This means it is executing far fewer instructions than the bubble sort.

Now we will implement the insertion sort. If the list is empty do nothing. Otherwise we set `end = 0`.

```
def selection(x):
    if x == []:
        return
    end = 0
```

Now we scan the list beyond `end` for the smallest element and swap it with `x[end]`. After this we update `end` by incrementing it.

```
def selection(x):
    if x == []:
        return
    end = 0
    while end < len(x):
        j = end
        for k in range(1 + end, len(x)):
            if x[j] > x[k]:
                j = k
        x[j],x[end] = x[end], x[j]
        end += 1
```

All of the sorting algorithms we used here are quadratic sorts, i.e. they operate in $O(n^2)$ time. Next, we will use recursion to perform sorts and we will realize a significant increase in performance for sorting large lists.

5.7 A Recursive Sort: Mergesort

Imagine you want to sort a humongous list. You might do this. Break the list in two. Use a quadratic sort on each half. Each half will take $1/4$ as long as doing a quadratic sort on the entire list. Then, once you have the two sorted lists, zipper them back together; that procedure is $O(n)$ as we shall soon see. We realize a significant performance boost.

Any handle worth cranking once is probably worth cranking a whole lot. What if we take each of the two smaller lists, cut them in half and ...?

That is the idea behind the *merge sort*. To make the merge sort work, we must first write a procedure to zipper two sorted lists together.

So suppose we have sorted lists `x` and `y`. Begin by making variables `j` and `k` and setting them both to be 0. We then make a new list. In each iteration we put the smaller of `x[j]` and `x[k]` on the new list, then iterate the index we used. So some code looking like this might be in order.

```
def zipper(x,y):
    j = 0
    k = 0
    out = []
    while j < len(x) and k < len(y):
        if x[j] < y[k]:
            out.append(x[j])
            j += 1
        else:
            out.append(y[k])
            k += 1
```

Once we get here, we will have arrived at the end of one or both of the two lists. Just append their tails and return our result as follows.

```
def zipper(x,y):
    j = 0
    k = 0
    out = []
    while j < len(x) and k < len(y):
        if x[j] < y[k]:
            out.append(x[j])
            j += 1
        else:
            out.append(y[k])
            k += 1
    out += x[j:]
    out += x[k:]
    return out
```

Now let us write the actual sort. Lists of length 0 or 1 are fully sorted. So we can no proceed as follows.

```
def merge(x):
    if len(x) <= 1:
        return x
```

Now comes the recursive step. Cut the list in two and call the `merge` function on each half and zipper them up!

```
def merge(x):
    if len(x) <= 1:
        return x
    mid = len(x)//2
    first = x[:mid]
    second = x[mid:]
    return zipper(merge(first), merge(second))
```

The disadvantage to this method that is is not an in-place method. You have to create an additional copy of the list. However, you will see that is now feasible to sort a 1,000,000 integer list on your PC.

Programming Exercises In this exercise, you will learn about the *Gnome Sort*, which was named by its inventor Hamid Sarbazi-Azad as the *Stupid Sort*. What is interesting is that our quadratic sorts have an inner and outer loop. This has a single loop. Here is the idea.

1. If the list has fewer than two items, you are done. Otherwise proceed as follows.
   (a) Start at the beginning of the list.
   (b) Move right one.
   (c) Check your value and the value to the left. If they are in order, move right one.
   (d) If they are out of order, swap them and move one to the left. If you are at the beginning, move right one.
   (e) Once you skid off the right hand end of the list, you are done.
2. Shuffle the list `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]` and run this procedure on it, printing it after each iteration. What do you see?
3. Learn about Timsort; this is the sorting method actually used by Python and Java. It is a variant on mergesort that seeks out runs in the data and zips them together.

5.8 Terminlogy Roundup

- **binary search** This is a divide and conquer technique by which we eliminate half of the search field in each iteration.
- **Bozo sort** This is sorting technique in which elements are shuffled and checked for proper order repeatedly until they are in the proper order.
- **Gnome aka Stupid Sort** This is a sort due to Hamid Sarbazi-Azad that looks like a modified insertion sort
- **linear-time** This describes an algorithm whose computational cost is proportional to the size of the problem.
- **merge sort** This is a recursive sort that keeps splitting the list into pieces until they are of size 1 or zero, and which zips the results together.
- **quotient** This is the result of dividing two numbers.
- **remainder** This is the remainder left by integer division.
- **prime factoriaztion theorem**
- **quadratic sort** This describes an algorithm whose computational cost is proportional to the size of the square of the problem.
- **sequence** This is a data structure that contains a collection indexed by the nonnegative integers.

Chapter 6

Introducing Java

6.0 Introduction

We will begin our study of the Java language using its REPL, `jshell`. This chapter will introduce you the idea of `class`. A class is a blueprint for the creation of objects. Both Python and Java support classes; this chapter will introduce those.

You will notice some important stylistic and linguistic differences. One is that all Java code *must* live in an class. Clearly this is optional in Python. Variable naming rules are the same in both languages. However, there is a style difference. Names for classes in both languages are capitalized and are formatted in camel notation. Variable and method names in Java are formatted in camel notation, while these in Python are predominantly formatted in `snake_notation`.

You will also see that worker statements in Java must end with a semicolon, and that boss statements have no mark at the end.

Python programs only have one epoch, run time. A Python program will run until it ends or it encounters a fatal error. Variables in Python are typeless names.

Java programs have two epochs, compile time and run time. Java runs on a *virtual machine*, which is a computer implemented in software. The first step is to compile, which translates your Java code into the machine languages for the JVM which is called *Java byte code*.

If you program does not make syntactic sense, compilation aborts and error messages issue forth. You must stamp out all syntax errors before your program will compile.

For a Java program to run, you will need to create a special main method. To get started we will be using `jshell` to learn about the type system and to inspect some simple classes we will create.

6.1 Welcome to JShell!

If you run Python with no file, you get an interactive Python shell. Java has a similar feature named `jshell`. To fire up `jshell`, type `jshell` at the command prompt and you will see this.

```
unix> jshell
> jshell
|  Welcome to JShell -- Version 9.0.1
|  For an introduction type: /help intro

jshell>
```

JShell has command that allows you to see the symbol table for your session, obtain help, and quit. We will use it quite a bit for inspecting and exploring classes.

Here is a summary of the commands. All of them begin with a /.

- `/exit` This quits `jshell`.
- `/help` This gives help on commands. The ussage is
- `/imports` This lists all classes you have imported.
- `/list` This lists all snippets you have entered.
- `/open` This reads a class into `jshell` so you can inspect it.
- `/types` This lists all active classes in the session.
- `/vars` This lists all variables and their values `/help /command` and you get a little man page for that command.

Here we show how to quit `jshell` using the `/exit` command.

```
unix> jshell
jshell
|  Welcome to JShell -- Version 14.0.1
|  For an introduction type: /help intro

jshell>

jshell> /exit
```

```
|  Goodbye
unix>
```

You can also quit it by typing control-d on Mac/UNIX or control-z on Windoze.

You enter pieces of code called *snippets* into the shell and `jshell` runs them. Let's do that and test-drive some of the commands. Begin by entering some variable declarations.

```
jshell> int x = 4;
x ==> 4

jshell> String y = "spaghetti";
y ==> "spaghetti"

jshell> int z = 15;
z ==> 15
```

You should notice some things right off. In the declaration `int x = 4;` we specified a type for the variable `x`. Java variables have type and this type must be known at compile time. Java variables can only point at objects of thier type. The compiler enforces this rule.

You have now seen examples of Java's integer type, `int` and its string type, `String`.

Now, watch `jshell` evaluate an expression.

```
jshell> x*z
$4 ==> 60
```

The symbol `\$4` is a valid variable in your `jshell` session. The `jshell` application is a "hoarder;" it gives a local variable name to just about everything you don't name.

```
jshell> $4
$4 ==> 60
```

Now we try the `/vars` command.

```
jshell> /vars
|    int x = 4
|    String y = "spaghetti"
|    int z = 15
|    int $4
```

Ooh, yummy, here is our visible symbol table. Now let's make a `/list` and check it twice.

```
jshell> /list
   1 : int x = 4;
   2 : String y = "spaghetti";
   3 : int z = 15;
   4 : x*z
   5 : $4
```

Here we see all of the snippets we have created this session. Next let us try `/vars`. Now let us quit this session. We will show how to inspect a class. Create this class.

```
public class Example
{
    public void go()
    {
        int x = 5;
        System.out.println("x = " + x);
    }
}
```

Next crank up `jshell` and open the file and inspect a method as follows.

```
jshell> /open Example.java

jshell> Example e = new Example();
e ==> Example@26653222

jshell> e.go()
x = 5
```

Now see the `/types` command in action.

```
jshell> /types
|    class Example
```

Observe the use of the langauge keyword `new`. Its use indicates that Java is creating an object on the heap and the associated variable is storing the memory addreess of that object. The dot notation you see in `e.go()` has the same effect as a Python method. It is reflective of the genetive case; you are calling `e`'s `go()` method.

While we are here, let us create the same class in Python. You will see a mysterious argument `self` in Python class methods. We will explain more about this later.

```
class Example:
    def go(self):
        x = 5
        print(f"x = {x}")
```

Now we can import it into our Python session and inspect it.

```
>>> from Example import Example
>>> e = Example()
>>> e.go()
x = 5
>>>
```

In both cases, our `Example` object can do one thing, `go()`.

6.2 How does Java Work on a Mechanical Level?

We will begin by looking at the mechanics of producing a program. We will then sketch a crude version of what actually happens during the process and refine it as we go along. Here is a simplified life-cycle for a Java program. So you can follow along, make this empty class. `Foo.java`.

```
public class Foo
{
}
```

1. **Edit** You begin the cycle by creating code in a text editor and saving it. Each file of Java will have a `public` class in it. The class is the fundamental unit of Java code; all of your Java programs will be organized into classes. Java classes are similar to those in Python; later we will compare them. The name of the class must match the name of the file; otherwise, you will get a nastygram from the compiler. As you saw in the example at the end of the last section, the file containing `public class Foo` must be be named `Foo.java`.
   Deliberately trigger the naming convention error creating an empty class `Right` in the file `Wrong.java`. Compile and you will receive this beating.

   ```
   unix> javac Wrong.java
   Wrong.java:1: error: class Right is public,
       should be declared in a file named Right.java
   public class Right
          ^
   1 error
   unix>
   ```

An optional but nearly universal convention is to capitalize class names. You should adhere to this rule in the name of wise consistency. This is done by all serious Java programmers; uncapitalized class names just confuse, annoy, and vex others.

2. **Compile** Java is an example of a *high–level language*. A complex program called a *compiler* converts your program into an executable form.
Compilation of Java in a command window is simple. To compile `Foo.java`, proceed as follows

```
unix> javac Foo.java
```

When done, list your files and you should see `Foo.class`. If your program contains syntactical errors that make it unintelligible to the compiler, the compilation will abort. When this happens, nothing executes and no executable file is generated. In contrast, in the Python language, the program stops running when an syntactical error is encountered; in Java the program does not run at all unless it compiles successfully. There is no compile time in Python.
If your program does not compile, you will get one or more error messages. These will be put to `stderr`, which by default, is your terminal window. You will need to re–edit your code to stamp out these errors before it will compile.
Java compiles programs in the machine language of the *Java Virtual Machine*; this machine is a virtual computer that is built in software. Its machine language is called *Java byte code*. In the `Foo.java` example, successful compilation yields a file `Foo.class`; this file consists of Java byte code. Your JVM takes this byte code and converts it into machine language for your particular platform, and your program will run. Java is not the only language that compiles to the JVM. Others include Scala, Clojure, a Lisp dialect, Processing, an animation language created in the MIT media lab, and Groovy, a scripting language. There is even a JVM implementation of Python called Jython.

3. **Run** For a program to run, it needs a special method called a *main method*. This method goes inside your class and it looks like this. It works in a manner somewhat similar to, but not entirely like, a Python main routine. You should notice one thing: Java has no global variables. This `main` method is the starting point for your program's execution. It is the first thing to go on the call stack.

```
public class Foo
{
    public static void main(String[] args)
    {
        System.out.println("foo");
    }
}
```

Save and compile this. If your compilation succeeds, you will be able to run your program as follows.

```
unix> javac Foo.java
unix> java Foo
foo
```

You will run your program and see if it works as specified. If not... back to the step **Edit**. You can have errors at run time, too. These errors result in an "exploding heart;" these ghastly things are nasty error messages containing many lines of ugly incantations. You can also have logic errors in your program; in this case, the program will reveal some unexpected behavior you did not want.

You will often hear the terms "compile time" and "run time" used. These are self-explanatory. Certain events happen when your program compiles, these are said to be compile time events. Others happen at run time. These terms will often be used to describe errors.

It is a good idea to compile any class before attempting to inspect it in `jshell`. If it fails to compile, your `jshell` session will be polluted with error messages.

Next, we take a brief tour of Python classes.

6.3 Python Classes

Python classes have a very simple structure. You can create a Python class with two lines of code.

```
class Simple(object):
    pass
```

A class is a blueprint for creating objects; this principle works the identically Python and Java. Making an object from this class is ... simple. Just do this.

```
>>> class Simple(object):
...     pass
...
>>> s = Simple()
>>> t = Simple()
>>>
```

We have created two objects `s` and `t` that are *instances* of this class.

We have learned that objects have state, identity and behavior. Recall that state is what an object knows, behavior is what an object does, and identity is

what an object is (a hunk of memory). Since the body of our class is empty, `Simple` objects know nothing and do nothing of great use. The can, however do some basic stuff. They can be represented as strings, and they can be checked for equality. Here we see this.

```
>>> s
<__main__.Simple object at 0x12e3250>
>>> t
<__main__Simple object at 0x12e3290>
>>> s == t
False
>>>
```

In Python, you can attach state to objects. Watch what is happening here.

```
>>> s.x = "I am x."
>>> s.y = "I am y."
>>> s.x
'I am x.'
>>> t.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Simple' object has no attribute 'x'
>>>
```

Now the object `s` knows its `x` and `y`, but `t` still knows nothing. We can make all instances of our class know `x` and `y` as follows.

```
>>> class Simple:
...     x = "I am x."
...     y = "I am y."
...
>>> s = Simple()
>>> t = Simple()
>>> s.x
'I am x.'
>>> t.x
'I am x.'
>>> s.y
'I am y.'
>>> t.y
'I am y.'
>>>
```

So far, our classes have fixed state. Objects of type `Simple` all have the same `x` and `y`. This is not terrible useful. Suppose we want to make a `Point` class to

represent points in the plane with integer coördinates. When we create a `Point`, we might want to specify its coördiantes. To do this, we will use a special method called `__init__`, which runs immediately after the object is created.

Let us now consider this program.

```
class Point(object):
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

p = Point()
print (f"p = ({p.x}, {p.y})")
q = Point(3,4)
print (f"q = ({q.x}, {q.y})")
```

Now run this program and see the following.

```
unix> python3 Point.py
p = (0, 0)
q = (3, 4)
```

We see a lot of new stuff here, so let us go through it with some care. We know that the `__init__` method runs immediately after a Point object is created. Its argument list is `(self, x, y)`. The purpose of the `x` and `y` seem clear: they furnish coördiantes to our `Point` object.

We also see this `self`. What is this? When you program in the `Point` class, you *are* a `Point`. So `self` is you. In the statement `self.x = x`, you are attaching the value `x` sent by the caller to yourself. The quantities `self.x` and `self.y` constitute the state of an instance of this class. This is how a `Point` knows its coördinates. The symbols `self.x` and `self.y` have scope inside of the entire class body. Take note that all argument lists of methods in a Python class must begin with `self`.

Now let us make a `Point` capable of doing something. Modify your `Point.py` program as follows.

```
import math
class Point(object):
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
    def distance_to(self, other):
        return math.hypot(self.x - other.x, self.y - other.y)

p = Point()
```

```
print(f"p = ({p.x}, {p.y})")
q = Point(3,4)
print(f"q = ({q.x}, {q.y})")
print(f"p.distance_to(q) = {p.distance_to(q)}")
```

Now run this program.

```
unix> python Point.py
p = (0, 0)
q = (3, 4)
p.distance_to(q) = 5.0
unix>
```

The class mechanism enables us to create our own new types of objects. Python supports the class mechanism, and object-oriented programming in general.

Java goes even further: all code in Java *must* appear in a class.

6.4 Java Classes

A Java program consists of one or more classes. All Java code that is created is contained in classes. So far you have created a class called `Foo` containing only a main method that prints to the screen.

In Python, you created programs that consisted of functions, one of which was the "main routine," which lived outside of all other functions. Your programs had variables that point at objects and functions that remember procedures. Java has these features but it works somewhat differently. Let us begin by comparing the time–honored "Hello, World!" program in both languages. In Python we code the following

```
#!/usr/bin/python
print("Hello, World!")
```

A Java vs. Python Comparison Python has classes but their use is purely optional. In Java, all of your code *must* be enclosed in classes. Throughout you will see that Java is more "modest" than Python. No executable code can be naked; all code must occur within a function that is further clothed in a class, with the exception of certain initialization of variables that still must occur inside of a class.

Also, we should remember we have two types of statements in Python, worker statements and boss statements. In Python, boss statements are grammatically incomplete sentences. Worker statements are complete sentences. All boss

statements in Python end in a colon (:), and worker statements have no mark at the end. All boss statements own a block of code consisting of one or more lines of code; an empty block can be created by using Python's *pass* keyword.

Java uses the system of boss and worker statements; the difference is cosmetic. In Java, boss statements have no mark at the end. Worker statements must be ended with a semicolon (;).

In Python, delimitation is achieved with a block structure that is shown by tabbing. In Java, delimitation is achieved by curly braces $\{\cdots\}$.

In Python, a boss statement must own a block containing at least one worker statement. In Java, a boss statement must have a block attached to it that is contained in curly braces. An empty block can be indicated by an empty pair of matching curly braces. Technically, you can get away with omitting the empty block, but it is much better to make your intent explicit.

Knowing these basic facts will make it fairly easy for you to understand simple Java programs.

Now, make the following class in Java and save it in the file `Hello.java`

```
public class Hello
{
    public void go()
    {
        System.out.println("Hello, World!");
    }
}
```

You can compile this class, but it needs a main method to run. Therefore, we add one.

```
public class Hello
{
    public void go()
    {
        System.out.println("Hello, World!");
    }
    public static void main(String[] args)
    {
        Hello greet = new Hello();
        greet.go();
    }
}
```

What's in the `main`? In the first line we see this.

```
    Hello greet  = new Hello();
```

This mysterious tells Java, "make a new object of type `Hello`." The variable `greet` points at a `Hello` object. Observe that `new` is a language keyword that is designated for creating objects. The `Hello` in front of `greet` says that `greet` is a variable of type `Hello`. Note, in contrast to Python, variables in Java have types. In fact, the compiler requires the type of all variables be known at compile time.

On the next line, we call `h`'s `go()` method, which puts the text `Hello, World!` to `stdout`

```
unix> java Hello
Hello, World!
```

You can think of the Java Virtual Machine as being an object factory. The class you make is a blueprint for the creation of objects. You may use the `new` keyword to manufacture as many objects as you wish. When you use this keyword, you tell Java what kind of object you want created, and it is brought into existence. Here we show a second `Hello` object getting created by using `new`. Modify your main method as follows

```
public static void main(String[] args)
{
    Hello greet = new Hello();
    greet.go()
    Hello snuffle = new Hello();
    snuffle.go()
}
```

Compile and run and you will see this.

```
unix> javac Hello.java
unix> java Hello
Hello, World!
Hello, World!
Hello@3cb075
Hello@e99ce5
```

Now there are two `Hello` objects in existence. Each has the capability to "`go()`." This is the only capability we have given our `Hello` objects so far. Every Java object has the built–in capability of representing itself as a string. The string representation of a `Hello` object looks like

```
Hello@aBunchOfHexDigits
```

You can see we have created two distinct `Hello` objects; the string representation of `greet` is `Hello@3cb075` and the string representation of `snuffle` is `Hello@e99ce5`. Each of the variables `greet` and `snuffle` is pointing at its own `Hello` object. Note that you will likely get different strings of hex digits from the ones seen here. Recall that a similar state of affairs inhered in Python; every object is able to represent itself as a string, but the default representation is not terribly useful.

The method `go()` represents a *behavior* of a `Hello` object. These objects have one behavior, they can `go()`. You can also see here that objects have identity. They "are". The two instances, `snuffle` and `greet` we created of the `Hello` class are different from one another. So far, we know that *objects have identity and behavior.*

This is all evocative of some things we have seen in Python. For example, if we create a string in Python, we can invoke the string method `upper()` to create a new string with all alpha characters in the string converted to upper case. Here is an example of this happening.

```
>>> x = "abc123;"
>>> x.upper()
'ABC123;'
>>>
```

The Python string object and the Java Hello object behaved identically. When we sent the string `x` the message `upper()`, it returned a copy of itself with all alpha characters converted to upper-case. In the Python interactive mode, this copy is put to the Python interactive session, which acts as `stdout`.

The Java Hello object `greet` put the text `"Hello, World!"` to `stdout`.

Now let us go through the program line-by-line and explain all that we see. The first line

```
public class Hello
```

is a boss statement. Read it as "To make a Hello," Since "To make a Hello," is not a complete sentence, we know the class head is a boss statement. Therefore it gets NO semicolon.

The word `public` is an *access specifier*. In this context, it means that the class is visible outside of the file. Python has no such modesty; it lacks any system of access specifiers. Later, you may have several classes in a file. Only one may be public. You may place other classes in the same file as your public class. This is done if the other classes exist solely to serve the public class. Java requires the file bear the name of the public class in the file. The compiler enforces this convention.

The words `public` and `class` are language keywords, so do not use them

as identifiers. The `class` keyword says, clearly enough, "We are making a class here." Now we go to the next line

```
{
```

This line has just an open curly brace on it. Java, in contrast to Python, has no format requirement. This freedom is dangerous. We will adopt certain formatting conventions. Use them, or develop your own and *be very consistent*. Consistent formatting makes mistakes easy to see and correct. It is unwise consistency that is the "hobgoblin of small minds;" wise consistency is a great virtue in computing.

The single curly brace acts like tabbing in in Python: it is a delimiter. It demarcates the beginning of the `Hello` class's code. The next line

```
    public void go()
```

is a function header. In Python you would write

```
def go():
```

There are some differences. The Python method header is a boss statement so it has a colon (:) at the end. Remember, boss statements in Java have no mark at the end. There is an access specifier `public` listed first. This says, "other classes can see this function." The `jshell` session behaves like a different class, so `go()` can be seen by `jshell` when we create a `Hello` object there. The other new element is the keyword `void`. A function in Java must specify the type of object it returns. If a function returns nothing, its return type *must* be marked `void`. Next you see the line

```
    {
```

This open curly brace says, "This is the beginning of the code for the function `go`." It serves as a delimiter marking the beginning of the function's body. On the next line we see the ungainly command

```
    System.out.println("Hello, World!");
```

The `System.out.println` command puts the string `"Hello, World!"` to standard output and then it adds a newline. The semicolon is required for this statement because it is a worker statement. Try removing the semicolon and recompiling. You will get a compiler error

```
> javac Hello.java
Hello.java:5: error: ';' expected
```

```
        System.out.println("Hello, World!")
                                           ^
1 error
```

A semicolon must be present at the end of all worker statements in Java. It is a mistake to put a semicolon at the end of a boss statement. In Java, the compiler can sometimes fail to notice this and your program will have a strange logic error.

The next line

```
    }
```

signifies the end of the `go` method. Finally,

```
}
```

ends the class.

In summary, all Java code is packaged into classes. What we have seen here is that we can put functions (which we call *methods*) in classes. The methods placed in classes give objects created from classes behaviors. We shall turn next to looking at Java's type system so we can write a greater variety of methods.

Programming Exercises Add new methods to our `Hello` class with these features.

1. Have a method use `System.out.print()` twice. How is it different from `System.out.println()`?
2. Have a method do this.

   ```
   System.out.printf("<tr><td>%s</td><td>%s</td></tr>",
       16, 256);
   ```

   Experiment with this `printf` construct. Note its similarities to Python's formatting `%` construct.
3. Enter this into `jshell`.

   ```
   String s = String.format(("<tr><td>%s</td><td>%s</td></tr>",
       16, 256);
   ```

   You can, in effect "print to a String" and save yourself a lot of annoying concatenation.
4. Create the `Hello` class using Python, giving it a `go` method.

6.5 Java's Integer Types

Java does something that Python does not do; it has eight types that are termed to be *primitive types*. We will meet all of these shortly. All of them are small pieces of data; none is larger than 8 bytes.

Recall that there are two important areas of memory in programs, the stack and the heap. The heap is a data warehouse where objects are kept. The heap can be expanded by the operating system at runtime; memory in the heap is fairly plentiful.

In Python, if you make the assignment `x = 5`, the variable `x` stores the location of the value `5` on the heap. All Python objects are stored on the heap.

The stack is a fixed-sized portion of memory that manages all function and method calls. If it runs out of room, you have a dreaded stack overflow on your hands and your program will careen inexorably to its death.

The stack holds all of the variables in your program. Recall that is organized into frames, which are constructed when a function is called, and which evanesce when a function returns.

In Python the only things stored on the stack are variables and the memory addresses they point a. Creating a new class allows you to create new types. Every time you create a class, you are actually extending the Java language. Like all things that are built out of other things, there must be some "atoms" at the bottom. Said atoms are called *primitive types* in Java. Java has eight primitive types. All other types in Java are *object types*; this includes such things as strings, lists, and graphical widgets. Note that Python does not have this distinction; in Python these sorts of simple types are just immutable objects.

We will begin by discussing Java's four integer types. These are all primitive types. Let us begin by studying these in `jshell`. Note that the sizes are standard and do not vary with the system. This stands in contrast to C/C++, if you have studied them before.

Type	Size	Explanation
`long`	8 bytes	This is the double–wide 8 byte integer type. It stores a value between -9223372036854775808 and 9223372036854775807. These are 64-bit two's complement integers
`int`	4 bytes	This is the standard two's complement 4 byte integer type, and the most commonly used integer type. It stores a value between -2147483648 and 2147483647.
`short`	2 bytes	This is the standard 2 byte integer type. It stores a value between -32768 and 32767 in two's complement notation.
`byte`	1 byte	This is a one–byte integer that stores an integer between -128 and 127 in two's complement notation.

You should note that Python 3 has one integer type and that Python 2 has two: `int` and `long`.

Now create an integer variable named `x` and initialize it to 5 as follows.

```
jshell> int x = 5
x ==> 5
```

You are seeing the assignment operator `=` at work here. This works just as it does in Python; you should read it as, "x gets 5" and not "x equals 5." As is true in Python, it is a worker statement. Consequently in a compiled program, it must be ended with a semicolon. Observe that the `jshell` program is lenient about semicolons

The expression on the right–hand side is evaluated and stored into the variable on the left–hand side. Now enter `x` into `jshell`. It fetches the value of `x`, which is 5.

```
jshell> x
x ==> 5
```

Now enter the `/vars` command. You will see the visible symbol table.

```
jshell> /vars
|    int x = 5
```

To create a variable in Java, you need to specify its type and an identifier (variable name). This is because Java is a *statically compiled language*; the types of all variables must be known at compile time. In general a variable is created in a declaration of the form

```
type variableName;
```

You can initialize the variable when you create it like so.

```
type variableName = value;
```

When you create local variables inside of methods you should always initialize them, or the compiler will growl at you.

Now let us deliberately do something illegal. We will set a variable of type `byte` equal to 675. Watch Java rebel.

```
jshell> byte b = 675
|  Error:
|  incompatible types: possible lossy conversion from int to byte
|  byte b = 675;
|           ^-^
```

This would attract the compiler's attention in a compiled program and cause compilation to abort and for your program to error out. You should create a little class with a method doing this and see for yourself.

6.5.1 Using Java Integer Types in Java Code

So far we have seen Java's four integer types: `int`, `byte`, `short`, and `long`. To see them in code, begin by creating this file.

```
public class Example
{
    public void go()
    {
    }
}
```

Once you enter the code in the code window, compile and save it. It now does nothing. Now we will create some variables in the method `go` and do some experiments. Modify your code to look like this and compile.

```
public class Example
{
    public void go()
    {
        int x = 5;
        System.out.println("x = " + x);
    }
}
```

Can you make the same output using `System.out.printf`? You should try this before moving on.

Compile the program. Now open it in `jshell` using the `/open` command Then enterthe code `e = new Example()` into `jshell`. Then use the `/types` command as follows.

```
jshell> /open Example.java

jshell> /types
|    class Example

jshell> Example e = new Example();
e ==> Example@26653222

jshell>
```

Notice the mysterious item `Example@(some gibberish)`. If you noticed the gibberish looks like hex code, you are right. All Java objects can print themselves, what they print by default is not very useful. Later we will learn how to change that. Now let us send the message "`go()`" to our `Example` object `e`.

```
jshell> e.go()
x = 5
```

Recall from the `Hello` class that `System.out.println` puts things to standard output with a newline at the end.

Inside the `System.out.println()` command, we see the strange sequence `"x = " + 5`. Java has a built–in string type `String`, which is akin to Python's `str`. In Python, you would have written

```
print("x = " + str(x))
```

Java is something of a stringophile language, like its distant cousin JavaScript. Once Java knows that an expression is to be a string, any other objects concatenated to the expression are automatically converted into strings and are added to the concatenation. That is why you see

```
x = 5
```

printed to `stdout`. Note that Python is very strict in this matter and requires you to explicitly convert objects to string before they can be concatenated to a string.

Now let us add some more code to our `Example` class so we can see how these integer types work together.

```
public class Example
{
    public void go()
    {
        int x = 5;
        System.out.println("x = " + x);
        byte b = x;
        System.out.println("b = " + b);
    }
}
```

Now we compile our masterpiece and we get these scoldings from Java.

```
unix> javac Example.java
Example.java:7: error: incompatible types:
            possible lossy conversion from int to byte
        byte b = x;
                 ^
1 error
unix>
```

Indeed, line 7 contains the offending code

```
byte b = x;
```

To fully understand what is happening, let's do a quick comparison with Python and explain a few differences with Java.

Types: Java vs. Python Python is a strongly, dynamically typed language. This means that objects are aware of their type and that decisions about type are made at run time. Variables in Python are merely names; they have no type.

In contrast, Java is a strongly, statically typed language. In the symbol table, Java keeps the each variable's name, the object the variable points at and the variable's type. Types are assigned to variables at *compile time*. In Python a variable may point at an object of any type. In Java, variables have type and may only point at objects of their own type.

Now let's return to the example. The value being pointed at by `x` is 5. This is a legitimate value for a variable of type `byte`. However, `x` is an integer variable and knows it is an integer. The variable `b` is a `byte` and it is aware of its `byte`ness. When you perform the assignment

```
b = x;
```

Java sees that `x` is an integer. An integer is a bigger variable type than a byte. The variable `b` says, "How dare you try to stuff that 4–byte integer into my one–byte capacity!" Java responds chivalrously to this plea and the compiler calls the proceedings to a halt.

In this case, you can cast a variable just as you did in Python. Modify the program as follows to cast the integer `x` to a `byte`.

```
public class Example
{
    public void go()
    {
        int x = 5;
        System.out.println("x = " + x);
        byte b = (byte) x;
        System.out.println("b = " + b);

    }
}
```

Your program will now run happily.

```
jshell> /open Example.java
jshell> Example e = new Example();
e ==> Example@26653222
jshell> e.go();
x = 5
b = 5
jshell>
```

Now let's play with fire. Change the value you assign `x` to 675.

```
public class Example
{
    public void go()
    {
        int x = 675;
        System.out.println("x = " + x);
        byte b = (byte) x;
        System.out.println("b = " + b);

    }
}
```

This compiles very happily. It runs, too!

```
jshell> Example e = new Example();
e ==> Example@26653222

jshell> e.go()
x = 675
b = -93

jshell>
```

Whoa! When casting, you can see that the doctrine of *caveat emptor* applies. If we depended upon the value of `b` for anything critical, we can see we might be headed for a nasty logic error in our code. When you cast, you are telling Java, "I know what I am doing." With that right, comes the responsibility for dealing with the consequences.

Challenge How did the -93 come about? Think about doughnutting!

Notice that you are casting from a larger type to a smaller type. This is a type of *downcasting*, and it can indeed cause errors that will leave you downcast. Since we discussed downcasting, let's look at the idea of upcasting that should easily spring to mind. For this purpose, we have created a new program that upcasts a byte to an integer

```
public class UpCast
{
    public void go()
    {
        byte b = 122;
        System.out.println("b = " + b);
        int x =  b;
        System.out.println("x = " + x);

    }
}
```

This compiles and runs without comment.

```
jshell> /open UpCast.java

jshell> UpCast u = new UpCast();
u ==> UpCast@26653222

jshell> u.go()
b = 122
x = 122
```

The four integer types are just four integers with different sizes. Be careful if casting down, as you can encounter problems. Upcasting occurs without comment. Think of this situation like a home relocation. Moving into a smaller house can be difficult. Moving into a larger one (theoretically) presents no problem with accommodating your stuff.

Important! If you use the arithmetic operators `+`, `-`, `*` or `/` on the short integral types byte and short, they are automatically upcast to integers as are their results. The compound assignment operators such as `+=` which work as they do in Python. One nice feature of these is that you can use the them on these shorter types and not get the "possibly lossy conversion" error. See this session here.

```
jshell> byte b = 1;
b ==> 1
jshell> byte b = 1;
b ==> 1

jshell> b = b + 5
|  Error:
|  incompatible types: possible lossy conversion from int to byte
|  b = b + 5
|      ^---^
jshell> byte b = 1;
b ==> 1

jshell> b = b + 5
|  Error:
|  incompatible types: possible lossy conversion from int to byte
|  b = b + 5
|      ^---^

jshell> b += 5
$2 ==> 6

jshell> b += 100
$3 ==> 106

jshell> b
b ==> 106

jshell> b += 21
$5 ==> 127

jshell> b += 1
```

```
$6 ==> -128
```

Oops, maybe it's time to discuss the problem of type overflow and "doughnutting." Since the `byte` type is the smallest integer type, we will demonstrate these phenomena on this type. Observe that the binary operators `+`, `-`, `*`,, and `%` work in java just as they do in Python. However, `/` in Java is integer division and works like Python's `//`.

Open `jshell` and run these commands. By saying `int b = 2147483647`, we guarantee that Java will regard `b` as a regular integer.

```
jshell> int b = 2147483647;
b ==> 2147483647

jshell> b += 1;
$5 ==> -2147483648

jshell> b
b ==> -2147483648

jshell>
```

The last command `b += 1` triggered an unexpected result. This phenomenon called *type overflow*. As you saw in the table at the beginning of the section, a byte can only hold values between -2147483648 and 2147483647. Adding 1 to 2147483647 yields -2147483648; this phenomenon is called *doughnutting*. It is an artifact of the workings of two's complement notation. You can see that this occurs in C/C++ as well.

This is caused by the fact that integers in Java are stored in two's complement notation.

6.6 Four More Primitive Types

The table below shows the rest of Java's primitive types. We see there are eight primitive types, four of which are integer types.

Type	Size	Explanation
`boolean`	1 byte	This is just like Python's `bool` type. It holds a `true` or `false`. Notice that the boolean constants `true` and `false` are not capitalized as they are in Python.
`float`	4 bytes	This is an IEEE 754 floating point number It stores a value between -3.4028235E38 and 3.4028235E38.
`double`	8 bytes	This is an IEEE 754 double precision number. It stores a value between -1.7976931348623157E308 and 1.7976931348623157E308. This is the same as Python's `float` type. It is the type we will use for representing floating–point decimal numbers.
`char`	2 byte	This is a two–byte Unicode character. In contrast to Python, Java has a separate character type.

6.6.1 The `boolean` Type

Let us now explore booleans. Java has three boolean operations which we will show in a table

Operator	Role	Explanation
`&&`	and	This is the boolean operator $\wedge$. It is a binary infix operator and the usage is `P && Q`, where `P` and `Q` are boolean expressions. If `P` evaluates to `false`, the expression `Q` is ignored.
`||`	or	This is the boolean operator $\vee$. It is a binary infix operator and the usage is `P || Q`, where `P` and `Q` are boolean expressions. If `P` evaluates to `true`, the expression `Q` is ignored.
!	not	This negates a boolean expression. It is a unary prefix operator. Be careful to use parentheses to enforce your intent!

Hand–in–hand with booleans go the *relational operators*. These work just as they do in Python on primitive types. The operator `==` checks for equality, `!=` checks for inequality and the operators `<`, `>`, `<=` and `>=` act as expected on the various primitive types. Numbers (integer types and floating point types) have their usual orderings. Characters are ordered by their ASCII values. It is an error to use inequality comparison operators on boolean expressions.

Now let us do a little interactive session to see all this at work. You are encouraged to experiment on your own as well and to try to break things so you

better understand them.

```
jshell> 5 < 7
$7 ==> true

jshell> 5 + 6 == 11   // == tests for equality
$8 ==> true

jshell> !(4 < 5)     //  ! is not
$9 ==> false

jshell> (2 < 3) && (1 + 2 == 5)  // and at work
$10 ==> false

jshell> (2 < 3) || (1 + 2 == 5)   // or at work
$11 ==> true

jshell> 100 %7 == 4    //  % is mod
$12 ==> false
```

6.6.2 Floating–Point Types

When dealing with floating–point numbers we will only use the `double` type. Do not test floating–point numbers for equality. Since they are stored inexactly in memory, comparing them exactly is a dangerous hit–or–miss proposition. Instead, you can check and see if two floating–point numbers are within some tolerance of one another. Here is a little lesson for the impudent to ponder. Be chastened!

```
jshell> double q = .3
q ==> 0.3

jshell> double r = .1 + .1 + .1
r ==> 0.30000000000000004

jshell> q == r
$25 ==> false
```

All integer types will upcast to the `double` type when you combine them with `double`s in an expression. You can also downcast doubles to integer types; you should experiment and see what kinds of truncation occur. You should experiment with this in `jshell`. Remember, downcasting can be hazardous and ... leave you downcast. Pay especial attention to negative numbers. If you are a number theory geek, you will have a negative reaction.

6.6.3 The `char` type

In Python, characters are just one–character strings. Java works differently and is more like C/C++ in this regard. It has a separate type for characters, `char`.

Recall that Western characters are really just bytes. Java uses the *unicode* scheme for encoding characters. All unicode characters are two bytes. The ASCII characters are prepended with a zero byte to make them into unicode characters. You can learn more about unicode at `http://www.unicode.org`.

Integers can be cast to characters, and the unicode value of that character will appear.

Here is a sample interactive session. Notice that the integer 945 in unicode translates into the Greek letter α.

```
> (char) 65
'A'
> (char) 97
'a'
> (char)945
'α'
> (char)946
'β'
```

Similarly, you can cast an integer to a character to determine its ASCII (or unicode) value.

The relational operators may be used on characters. Just remember that characters are ordered by their Unicode values. The numerical value for the 8 bit characters are the same in Unicode. Remember, Unicode characters are two bytes; all of the 8 bit characters begin with the byte `00000000`.

6.7 More Java Class Examples

Now let us develop more examples of Java classes. Since we have the primitive types in hand, we have some grist for producing useful and realistic examples. Let us recall the basics. All Java code must be enclosed in a class. So far, we have seen that classes contain methods, which behave somewhat like Python functions.

Open an editor session and place this code in a file named `MyMethods.java`.

```
public class MyMethods
{
    public double square(double x)
```

```
    {
        return x*x;
    }
}
```

Compile this program. Once it compiles, open it and `jshell` and create an instance of it.

```
jsyell> MyMethods m = new MyMethods();
```

Recall that `new` tells Java, "make a new `MyMethods` object." Furthermore, we have assigned this to the variable `m`. Now type `m.getClass()` and see `m`'s class.

```
jshell> m.getClass()
class MyMethods
```

Every Java object is born with a `getClass()` method. It behaves much like Python's `type()` function. For any object, it tells you the class that created the object. In this case, `m` is an instance of the `MyMethods` class, so `m.getClass()` returns `class MyMethods`.

We endowed our class with a `square` method; here we call it.

```
jshell> m.square(5)
25.0
```

The name of the method leaves us no surprise as to its result. Now let us look inside the method and learn its anatomy.

```
    public double square(double x)
    {
        return x*x;
    }
```

In Python, you would make this function inside of a class by doing the following.

```
class MyMethods:
    def square(self, x):
        return x*x
```

in both the top line is called the *function header*. Notice that in Python, you must use the `self` variable in the argument list for any methods you create. Python functions begin with the `def` statement; this tells Python we are defining a function. Java methods begin with an *access specifier* and then a *return type*.

The access specifier controls visibility of the method. The access specifier `public` says that the `square` method is visible outside of the class `MyMethods`. The return type says that the `square` method will return a datum of type `double`.

In both Python and Java, the next thing you see is the function's name, which we have made `square`. The rules for naming methods in Java are the same as those for naming variables. To review, an identifier name may start with an alpha character or an underscore. The remaining characters may be numerals, alpha characters or underscores.

Inside the parentheses, we see different things in Java and Python. In Python, we see a lone `x`. In Java, we see `double x`. Since Java is statically typed, it requires all arguments to specify the type of the argument as well as the argument's name. This restriction is enforced *at compile time*. In contrast, Python makes these and all decisions at run time.

In general every Java method's header has the following form.

```
returnType functionName(type1 arg1, type2 arg2,  ...  , typen argn)
```

The list

```
[type1, type2, ... typen]
```

of a Java method is called the method's *signature*, or "sig" for short. Notice that the argument names are not a part of the signature of a method. Remember, such names are really just dummy placeholders. Methods in Java may have zero or more arguments, just as functions and methods do in Python.

Try entering `m.square('a')` into `jshell`.

```
jshell> m.square('a')
Error: No 'square' method in 'MyMethods' with arguments: (char)
```

Compilation would fail for this program, and , `jshell` objects by saying that a character is an illegal argument for your method `square`. Java methods have type restrictions in their arguments. Users who attempt to pass data of illegal type to these methods are rewarded with compiler errors. This sort of protection is a two–edged sword. Add this method to your `MyMethods` class.

```
    public double dublin(double x)
    {
        return x*2;
    }
```

Now let us do a little experiment.

```
jshell> /open MyMethods.java
jshell> MyMethods m = new MyMethods();
jshell> m.dublin(5)
$1 ==> 10.0
jshell> m.dublin("string")
Error: No 'dublin' method in 'MyMethods'
with arguments: (java.lang.String)
```

What have we seen? The `dublin` method belonging to the `MyMethods` class will accept integer types, which upcast to doubles, or doubles, but it rejects a string. (More about Java's string type later)

We will now write the analogous function in Python; notice what happens. Place this Python code in a file named `method.py`.

```
def dublin(x):
    return x*2
x = 5
print (f"dublin({x}) = {dublin(x)}")
x = "string"
print (f"dublin({x}) = {dublin(x)}")
```

Now let us run it at the command line..

```
unix> python method.py
dublin(5) = 10
dublin(string) = stringstring
unix>
```

Python makes decisions about objects at run time. The call `dublin(5)` is fine because it makes sense to take the number 5 and multiply it by the number 2. The call `dublin("string")` is fine for two reasons. First, multiplication of a string by an integer yields repetition, so the return statement in the function `dublin` makes sense to Python at run time. Secondly, variables in Python have no type, so there is no type restriction in `dublin`'s argument list. You will notice that static typing makes the business of methods more restrictive. However, compiler errors are better than run time errors, which can conceal ugly errors in your program's logic and which can cause surprisingly unappealing behavior from your function.

Just as in Python, you may have functions that produce no output and whose action is all side–effect. To do this, just use the `void` return type, as we did in the `Hello` class.

Programming Exercises

1. Add a method `double cube(double x)` to the class `MyMethods.java`. Can you use the `square` method to do part of the computation?
2. Modify the `square` method as follows.

   ```
   public double square(double x)
   {
       double y = x*x;
       return y;
   }
   ```

   Create a main method for this class and try placing this command in it.

   ```
   System.out.println(y);
   ```

   What happens? Why is this like Python?

change

Chapter 7

Classes and Objects

7.0 Java Object Types

Let us recall a few ideas from our Python experience. There are two areas of memory important to programs, the stack and the heap. The stack manages all function calls. Each function call puts a stack frame on the stack which contains the function's local variables and a return address that allows it to go back to its caller right where the caller left off when it returns. Any time a function returns its stack frame is exposeed to be overwritten; its contents become inaccessible. We saw that this is how local variables are managed.

We have seen that Java has eight primitive types: the four integer types, the floating–point types `double` and `float`, the `boolean` type and the `char` type. These variables point directly at their datum. As a result, the values of local variables are stored directly on the stack and are never seen by the heap.

Another important difference is that Java does not have global variables. All variables must be declared and initialized inside of a class.

Strings in Java are an *object type*. Object types in Java work in a manner similar to Python objects. Varibles of object type store a memory address; this is an address on the heap. The heap, as is is in Python, is the data warehouse of a Java program.

Only memory addresses (which are basically integers) and primitive types are ever stored by the stack.

You will see that Java strings have many capabilities. You can slice them as you can Python strings, they know their lengths, and you have access to all characters. You will learn how to use the Java API guide to learn more about any class's capabilities, including those of `String`.

7.1 Java Strings

Java handles strings in a manner similar to that of Python. Strings in Java are immutable. Java has an enormous *standard library* containing thousands of classes. The string type is a part of this vast library, an it is implemented in a class called `String`. We will explore it in a fair amount of detail here.

Because strings are so endemic in computing, the creators of Java gave Java's string type some features atypical of Java classes, which we shall point out as we progress.

Let us begin by working interactively. Here we see how to read individual characters in a string by their indices. Notice that the square-brackets operator does not work here.

```
jshell> String x = "abcdefghijklmnopqrstuvwxyz"
x ==> "abcdefghijklmnopqrstuvwxyz"

jshell> x.charAt(0)
$2 ==> 'a'

jshell> x[0]
  Error:
  array required, but java.lang.String found
  x[0]
  ^--^

jshell> x.charAt(25)
$3 ==> 'z'

jshell> x.length()
$4 ==> 26
```

Now let us deconstruct all of this. Strings in Java enjoy an exalted position. The line

```
String x = "abcdefghijklmnopqrstuvwxyz"
```

makes a *tacit* call to `new` and it creates a new `String` object in memory. Only a few other Java class enjoys the privilege of making tacit calls to `new`; these are the *wrapper classes*. Let us take a brief detour to see one of them, `Integer`, which is the wrapper type for the primitive type `int`. You can create an `Integer` either by saying

```
Integer n = 5;
```

or by saying

```
Integer n = new Integer(5);
```

This tacit call to `new` is enabled by a feature called *autoboxing*. We will meet the wrapper classes in full after we finish discussing the string class.

Coming back to our main thread, you can create a string using `new` as well.

```
String x = new String("abcdefghijklmnopqrstuvwxyz");
```

Here we made an *explicit* call to `new`. This is not done very often in practice, as it is excessively verbose, and it can create duplicate copies of immutable objects.

Access to the individual characters within a string is granted via the `charAt` string method. The expression

```
x.charAt(25)
```

can be read as "`x`'s character at index `25`." Just as in Python, the dot (.) indicates the genitive case. The nastygram you got before,

```
|  Error:
|  array required, but java.lang.String found
|  x[0]
|  ^--
```

arises because the square–bracket operator, which exists in Python, is only used to extract array entries in Java. Arrays are a Java data structure, which we will learn about in the next chapter. Finally, we see that a string knows its length; to get it we invoke the `length()` method.

`String` has another atypical feature not found in other Java classes. The operators `+` and `+=` are implemented for Strings. The `+` operator concatenates two strings, just as it does in Python. The `+=` operator works for strings just as it does in Python.

```
jshell> String x = "abc"
x ==> "abc"

jshell> x += "def"
$2 ==> "abcdef"

jshell> x
x ==> "abcdef"
```

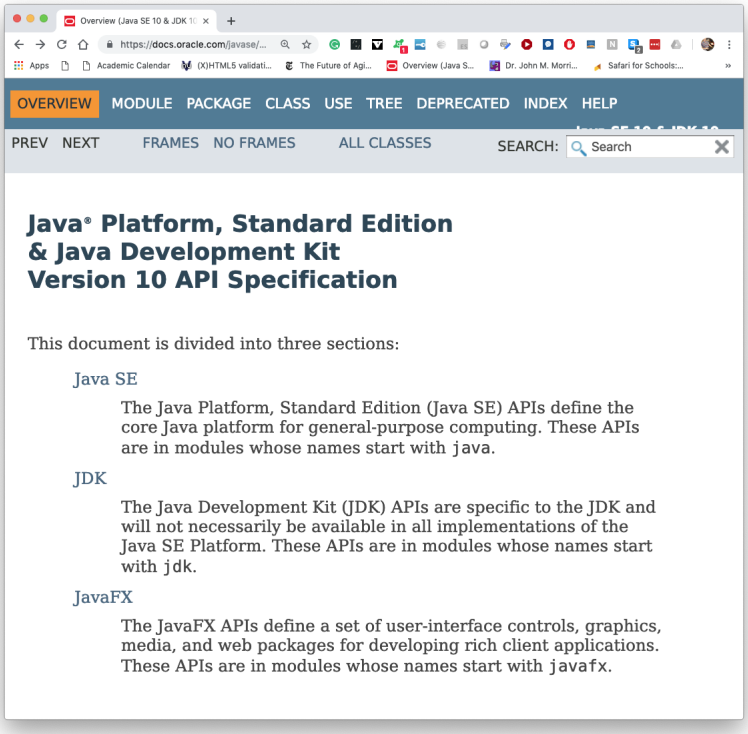
Note, however, that the string `"abc"` is not changed. It is orphaned and the String variable `x` now points at `"abcdef"`.

The mechanism of orphaning objects in Java works much as it does in Python. Both Python and Java are garbage-collected languages.

7.1.1 But is there More?

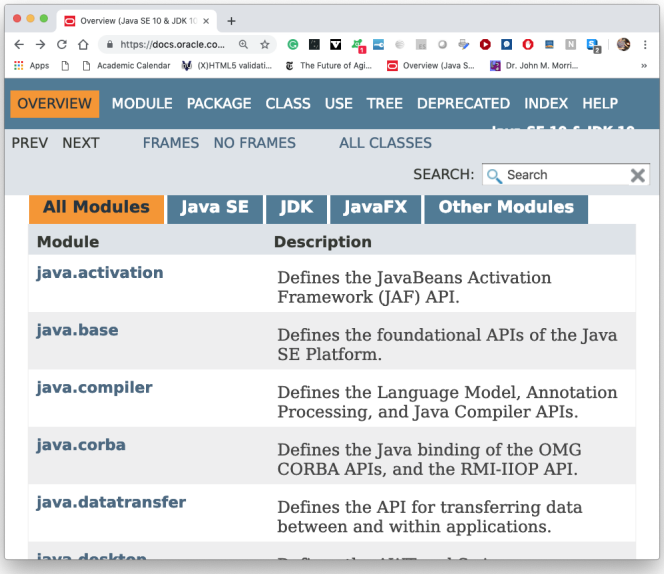
You might be asking now, "Can I learn more about the Java String class?" Fortunately, the answer is "yes;" it is to found in the Java API (Applications Programming Interface) guide. This is a freely available online comprehensive encyclopedia of all of the classes in the Java Standard Library. Go to this site, `https://docs.oracle.com/javase/15/docs/api/overview-summary.html` and you will see this. Note that you can go to the Java install site and obtain the entire documentation set for your computer.

When you go to the Java documentation, here is what you see.



Classes in Java have two levels of organization. One is modules. We are going to explore the module `java.base`.

Scroll down a little more and you will see the Modules area; it is tabbed. Click on the link for `java.base`.



When you open this link, you will see the packages inside of it. Here are some we will use in this book.

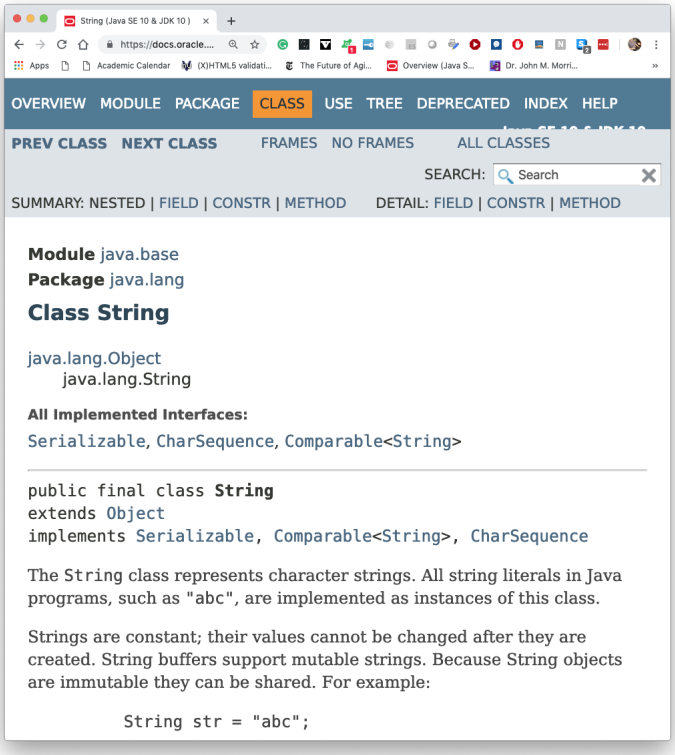
`java.lang`	This is the core of the Java language.
`java.io`	This is where Java's fileIO facilities live.
`java.nio`	This is where Java's newer fileIO facilities live.
`java.math`	This is the home of `BigInteger`, a class for arbitrary-precision integer arithmetic.
`java.util`	This is where Java's data structures live, along with some other useful stuff.

Now click on `java.lang` This page has several segments. Here they are.

- Interfaces
- Classses
- Enums
- Exceptions
- Errors

- Annotation Types

Scroll down the Classes segment. In it you will find a link for the class `String`. Click on it. This is the top of the page.



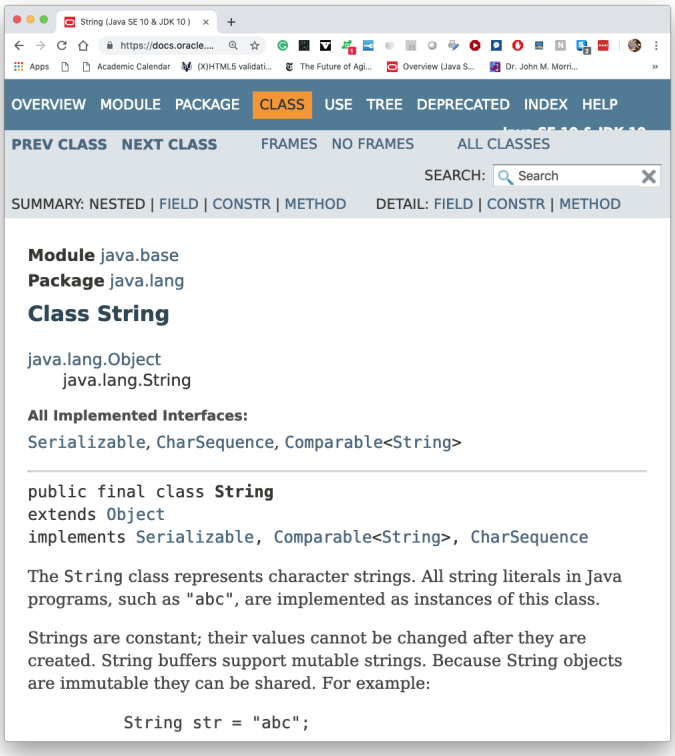
At the very top, the module and the package are identified for you. You can also see the *fully qualified name* of the `String` class, `java.lang.String`. Further down, you see this.

> The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.
>
> Strings are constant; their values cannot be changed after they are created.

In the beginning you will see much that you will not understand. For example, there is this.

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

We will learn about that stuff later. For now, you will learn how to pick out what you need to know. Now scroll down to the Method Summary area. The top of it looks like this.



There are three columns in this table. The first column contains the return type of the method and any modifier (later...). The second shows the name of the method and its argument list. The third describes the method briefly.

To learn more, click on `charAt`. You will see the *method detail*, which looks like this.

```
public char charAt(int index)
```

Returns the char value at the specified index. An index ranges from `0` to `length() - 1`. The first char value of the sequence is at index `0`, the next at index `1`, and so on, as for array indexing. If the char value specified by the index is a surrogate, the surrogate value is returned.

Specified by:
`charAt` in interface `CharSequence`

Parameters:
`index` - the index of the char value.

Returns:
the char value at the specified index of this string. The first char value is at index `0`.

Throws:
`IndexOutOfBoundsException` - if the index argument is negative or not less than the length of this string.

Right after the heading it says

```
public char charAt(int index)
```

This is the method header that appears in the actual `String` class. It then goes on to say the following.

> Returns the `char` value at the specified index. An index ranges from `0` to `length() - 1`. The first char value of the sequence is at index `0`, the next at index `1`, and so on, as for array indexing.

The following paragraph

> If the `char` value specified by the index is a surrogate, the surrogate value is returned.

looks pretty mysterious, so we will ignore it for now. The **Parameters:** heading describes the argument list. **Returns:** heading describes the return value. There are no surprises here.

What is interesting is the **Throws:** heading. This describes run time errors that can be caused by misuse of this method. These errors are not found by the compiler. If you trigger one, your program dies gracelessly and you get great list of nastiness put to your screen. You have observed similar tantrums thrown by Python when you give it an index that is out of bounds in a string, list or tuple.

We shall use this web page in the next sections so keep it open. First it will be necessary to understand a fundamental difference between Java object types and Java primitive types.

Programming Exercises Write a class called `Exercises11` and place the following methods in it.

1. Write the method

   ```
   public boolean isASubstringOf(String quarry, String field)
   {
   }
   ```

   It should return true when `quarry` is a contiguous substring of `field`. (Think Python `in` construct.)

2. Suppose you have declared the string `cat` as follows.

   ```
   String cat = "abcdefghijklmonopqrstuvwxyz";
   ```

   Find at least two ways in the API guide to obtain the string `"xyz"`. You may use no string literals (stuff inside of `"..."`), just methods applied to the object `cat`. There are at least three ways. Can you find them all?

3. The Python repetition operator `*`, which takes as operands a string and an integer, and which repeats the string the integer number of times does not work in Java. Write a method

   ```
   String repeat(String s, int n)
   ```

   that replicates the action of the Python repetition operator. And yes, there is recursion in Java.

7.2 Primitive vs. Object: A Case of `equals` Rights

We will study the equality of string objects. A big surprise lies ahead so pay close attention. Create this interactive session in Python. All is reassuringly familiar.

```
>>> good = "yankees"
>>> evil = "redsox"
>>> copy = "yankees"
>>> good == copy
True
>>> good == evil
False
```

No surprises greet us here. Now let us try the same stuff in Java.

```
jshell> String good = "yankees"
good ==> "yankees"

jshell> String evil = "redsox"
evil ==> "redsox"

String copy = new String("yankees")
```

```
copy ==> "yankees"

jshell> good == evil
$4 ==> false

jshell> good == good
$5 ==> true

jshell> good == copy
$7 ==> false
```

Beelzebub! Some evil conspiracy appears to be afoot! Despite the fact that both `good` and `copy` point to a common value of `"yankees"`, the equality test returns a `false`. Now we need to take a look under the hood and see what is happening.

First of all, let's repeat this experiment using integers.

```
jshell> int Good = 5;
Good ==> 5

jshell> int Evil = 4;
Evil ==> 4

jshell> int Copy = 5;
Copy ==> 5

jshell> Good == Evil
$12 ==> false

jshell> Good == Good
$13 ==> true

jshell> Good == Copy
$14 ==> true
```

This seems to be at odds with our result with strings. This phenomenon occurs because primitive and class types work differently.

Without exception, when you use `==` on two variables, you are asking if they store the same value. The value stored by a variable of object type is the memory address of its object. The value stored by a primitive type is the actual value it is storing; *a primitive type variable points directly at its datum.*

So if you are using `==` on variables of object type, you are comparing memory addresses. In our case, Strings do not directly store their object; they store its memory address. This is true of all object types in Java. What a java object

knows is a how to find where where the string is stored in memory. For objects, the == operator is a test for equality of identity. It checks if two objects are in fact one and the same. This behavior is identical to that of the Python `is` keyword, which checks for equality of identity.

In Python, objects *never* point directly at their datum. Python types such as `bool`, `float` and `int` are actually immutable objects. This phenomenon is a major difference between Python and Java. Python has no primitive types.

We saw `good == good` evaluate to true because `good` points to the same actual object in memory as itself. We saw `good == copy` evaluate to false, because `good` and `copy` point to separate copies of the string `"yankees"` stored in memory. Therefore the test for equality evaluates to false.

What do we do about the equality of strings? Fortunately, the `equals` method comes to the rescue.

```
jshell> good.equals(good)
$15 ==> true

jshell> good.equals(evil)
$16 ==> false

jshell> good.equals(copy)
$17 ==> true
```

Ah, paradise restored... Just remember to use the `equals` method to check for equality of Java objects. This method for strings has a close friend `equalsIgnoreCase` that will do a case-insensitive check for equality of two strings. These comments also apply to the inequality operator `!=`. This operator checks for inequality of identity. To check and see if two strings have unequal values use ! in conjunction with `equals`. Here is an example

```
jshell > !(good.equals(copy))
$ 18 ==> false
>
jshell> !(good.equals(evil))
$ 19 ==> true
```

Finally, notice that Python compares strings lexicographically according to Unicode value by using inequality comparison operators. These do not work in Java. It makes no sense to compare memory addresses. However, the string class has the method

```
int compareTo(String anotherString)
```

We show it at work here.

```
jshell> String little = "aardvark";
little ==> "aardvark"

jshell> String big = "zebra";
big ==> "zebra"

jshell> little <= big
|  Error:
|  bad operand types for binary operator '<='
|    first type:  java.lang.String
|    second type: java.lang.String
|  little <= big
|  ^-----------^

jshell> little.compareTo(big) < 0
$3 ==> true

jshell> little.compareTo(big) == 0
$4 ==> false

jshell> little.compareTo(big) > 0
$5 ==> false
```

You may be surprised `compareTo` returns an integer. However, alphabetical string examples can be done as in the example presented here. This method's sibling method, `compareToIgnoreCase` that does case–insensitive comparisons and works pretty much the same way.

7.2.1 Aliasing

Consider the following interactive session.

```
jshell> String smith = "granny";
smith ==> "granny"

jshell> String jones = smith;
jones ==> "granny"

jshell> smith == jones
$3 ==> true
```

Here we performed an assignment, `jones = smith`. What happens in an assignment is that the right–hand side is evaluated and then stored in the left. Remember, the string `smith` stores a memory address describing the location

where the string "granny" is actually stored in memory. So, this memory address is given to jones; both jones and smith hold the same memory address and therefore both point at the one copy of "granny" that we created in memory.

This situation is called *aliasing*. Since strings are immutable, aliasing can cause no harm. We saw in the Python book that aliasing can create surprises if the objects involved are mutable.

First we will need to be introduced a property of objects we have omitted heretofore in our discussion: *state*. The state of a string is given by the character sequence it contains. How these are stored is not now known to us, and we really do not need to know or care. We shall tour the rest of the string class in the Java API guide, then turn to the matter of state. Just know that a String's state is specified by the character sequence it contains.

7.3 More String Methods

Python's slicing facilities are implemented in Java using substring. Here is an example of substring at work.

```
jshell> x = "abcdefghijklmnopqrstuvwxyz"
jshell> x.substring(5)
"fghijklmnopqrstuvwxyz"
jshell> x.substring(3,5)
"de"
jshell> x.substring(0,5)
"abcde"
jshell> x
"abcdefghijklmnopqrstuvwxyz"
```

Notice that the original string is not modified by any of these calls; copies are the advertised items are returned by these calls. The endsWith method seems pretty self–explanatory.

```
jshell> x.endsWith("xyz")
true
jshell> x.endsWith("XYZ")
false
```

The indexOf method allows you to search a string for a character or a substring. In all cases, it returns a -1 if the string or character you are seeking is not present.

```
jshell> x.indexOf('a')
0
```

```
jshell> x.indexOf('z')
25
jshell> x.indexOf('A')
-1
jshell> x.indexOf("bc")
1
```

You can pass an optional second argument to the `indexOf` method to start searching at a specified index. For example, since the only instance of the character `'a'` in the string `x` is at index 0, we have t

```
jshell> x.indexOf('a', 1)
-1
```

You are encouraged to further explore the `String` API. It contains many useful methods that make strings a useful andx powerful programming tool. The programming exercises here will give you an opportunity to do this.

Programming Exercises Fill in the methods in this class. Copy it and ocmpile it; it will compile in this state. When you are done, it should print all `true`s. Do not worry about the use of the `static` keyword. It makes everything work and it will be explained later. Use the `String` API page to help you.

```
public class Exercises
{
    public static void main(String[] args)
    {
        System.out.println(between("catamaran", 'a').equals("tamar"));
        System.out.println(between("catamaran", 'c').equals(""));
        System.out.println(between("catamaran", 'q').equals(""));
        System.out.println(laxEquals("    boot", "boot"));
        System.out.println(laxEquals("boot    ", "boot"));
        System.out.println(laxEquals("     boot    ", "boot"));
        System.out.println(laxEquals(" \t\n    boot \n\t   ", "boot"));
        System.out.println(getExtension("wd2.tex").equals("tex"));
        System.out.println(getExtension("hello.py").equals("py"));
        System.out.println(getExtension("Hello.java").equals("java"));
        System.out.println(getExtension("tossMeNow").equals(""));
        System.out.println(getExtension(".").equals(""));
        System.out.println(isUpperCaseOnly("EAT NOW 123"));
        System.out.println(!isUpperCaseOnly("eat NOW 123"));
        System.out.println(isUpperCaseOnly(""));
    }
    /*
```

```
     * This returns an empty string if q is not present in
     * s or if it only appears once.  Otherwise, it returns the
     * substring between the first and last instances of q in s.
     */
    public static String between(String s, char q)
    {
        return "";
    }
    /*
     * This returns true if the only difference between s1 and s2
     * is leading or trailing whitespace.
     */
    public static boolean laxEquals(String s1, String s2)
    {
        return false
    }
    /*
     * this returns an empty string if the fileName is empty
     * or has no extension. Otherwise, it returns the extension
     * without the dot.
     */
    public static String getExtension(String fileName)
    {
        return "";
    }
    /*
     * this returns true if the String contains only uppercase
     * or non-alpha characters.
     */
    public static boolean isUpperCaseOnly(String s)
    {
        return false;
    }
}
```

and put these methods in it.

1. Write the method `public Stirng bewteen(String s, char q)` that returns an empty string if `q` occurs once or not at all inside of `s`; otherwise it reteurns the substring in between the first and last instances of `q` in `s`. Examples

7.4 The Wrapper Classes

Every primitive type in Java has a corresponding *wrapper class*. Such a class "wraps" the primitive object. These classes also supply various useful methods associated with each primitive type. Here is a table showing the wrapper classes.

	Wrapper Classes
Primitive	Wrapper
byte	Byte
short	Short
int	Integer
long	Long
boolean	Boolean
float	Float
double	Double
char	Character

All of these classes have certain common features. You should explore the API guide for each wrapper. They have many helpful features that will save you from reinventing a host of wheels.

One thing you will notice in the wrapper classes is the presence of methods marked `static`. We will discuss this later in more detail, but for now, just know that `static` methods can be called directly using the class name. If you have a class `Foo` and a static method named `cling`, you call it by using `Foo.cling()`. What you don't need to do is this.

```
Foo f = new Foo();
f.cling();
```

You can operate in this way, but calling a static method via the class name is the *preferred* way of calling a static method, and it saves the Java Virtual Machine work.

- **Immutability** Instances of these classes are immutable. You may not change the datum. You may only orphan the object by pointing at a new one with a different datum. This should remind you of Python, because this is how Python treats these types types such at `int`, `bool`, and `float`. It's just like good old Python.
- **A `toString()` method**, which returns a string representation of the datum.
- **A static `toString(primitiveType p)` Method** This method will convert the primitive passed it into a string. For example, `Integer.toString(45)` returns the string `"45"`.

- **A static parsePrimitive(String s) Method** This method converts a `String` into a value of the primitive type. For example,

  ```
  Integer.parseInt("455")}
  ```

  converts the string `"455"` into the integer `455`. For numerical types, a `NumberFormatException` is thrown if an malformed numerical string is passed them. The `Character` wrapper does not have this feature. You should take note of how this method works in a `Boolean`.

- **Membership in `java.lang`** All of these classes belong to the package `java.lang`; you do not need to import anything to use these classes.

Programming Exercises

1. Write an expression to see if a character is an upper-case letter.
2. Write an expression to see if a character is a digit.

7.4.1 Autoboxing and Autounboxing

These features make using the wrapper classes simple. Autoboxing automatically promotes a primitive to a wrapper class type where appropriate. Here is an example. The command

```
Integer i = 5;
```

really amounts to

```
Integer i = new Integer(5);
```

This call to `new` "boxes" the primitive 5 into an object of type `Integer`. The command `Integer i = 5;` automatically boxes the primitive 5 inside an object of type `Integer`. This results in a pleasing syntactical economy.

Autounboxing allows the following sensible–looking code.

```
Integer i = 5;
int x = i;
```

Here, the datum is automatically "unboxed" from the wrapper `Integer` type and it is handed to the primitive type variable `x`.

This is the smart way to compare a primitive with a boxed primitive. Direct comparison can be dangerous and result in errors.

```
jshell> Integer i = 5;
jshell> int y = i;
jshell> int x = 5;
jshell> i == y
true
jshell>
```

Autoboxing and autounboxing eliminate a lot of verbosity from Java; we no longer need to make most `valueOf` and `equals` calls.

7.5 Two Caveats

Do not box primitive types gratuitously. If you can keep variables primitive without a sacrifice of clarity or functionality, do so. Here is an example of a big mistake caused by seemingly innocuous choice. Although we have not met loops yet, you can easily figure out what is happening here. You are doing a million boxings and unboxings.

```
for(Integer i = 0; i < 1000000; i++)
{
    //code
}
```

This will be a significant performance hit. This is much better.

```
for(int i = 0; i < 1000000; i++)
{
    //code
}
```

It is best to prefer the use of primitive types, and to use the wrapper types when you need their helpful methods. We mention them because you will need to use them in conjunction with collections.

Using == on autoboxed primitives is almost always wrong You must use the `equals` method in this case.

7.6 Classes Know Things: State

So far, we have seen that objects have identity and that they have behavior, which is reflected by a class's methods.

We then saw that a string "knows" the character sequence it contains. We do not know how that sequence is stored, and we do not need to know that. The character sequence held by a string is reflective of its *state*. The state of an object is what an object "knows." Observe that the outcome of a Java method on an object can, and often does, depend upon its state. This works just as it does in Python.

To give you a look behind the scenes, we shall now produce a simple class which will provides a blueprint for making objects having state, identity and behavior. To do this, it will necessary to introduce some new ideas in Java, the *constructor* and *method overloading*.

Place the following code in a file, compile and save it with the name `Point.java`. We are going to create a simple class for representing points in the plane with integer coördinates.

```
public class Point
{
}
```

What does such a point need to know? It needs to know its `x` and `y` coördinates. Here is how to make it aware of these.

```
public class Point
{
    private int x;
    private int y;
}
```

You will see a new keyword: `private`. This says that the variables `x` and `y` are not visible outside of the class. These variables are called *instance variables* or *state variables*. We shall see that they specify the state of a `Point`.

Why this excessive modesty? Have you ever bought some electronic trinket, turned it upside–down and seen "No user serviceable parts inside" emblazoned on the bottom? The product–liability lawyers of the trinket's manufacturer figure that an ignorant user might bring harm to himself whilst fiddling with the entrails of his device. Said fiddling could result in a monster lawsuit that leaves the principles of the manufacturer living in penury.

Likewise, we want to protect the integrity of our class; we will not allow the user of our class to monkey with the internal elements of our program. We will permit the client programmer access to these elements by creating methods that give access. This is a hard-and-fast rule of Java programming: *Always declare your state variables private*.

Additionally, if we decide later that it is better to implement the class in a newer and better way, we can do this and we can keep the interface the same. This allows us to do an "engine upgrade" but have the operation of the car be the same. It will just have a little more pep when you step on the gas.

Now compile your class. Let us make an instance of this class and deliberately get in trouble.

```
jshell> Point p = new Point();
p ==> Point@26653222

jshell> p.x
|  Error:
|  x has private access in Point
|  p.x
|  ^-^
```

We have debarred ourselves from having any access to the state variables of an instance of the `Point` class. This makes our class pretty useless. How do we get out of this pickle?

7.6.1 Quick! Call the OBGYN! And get a load of `this`!

Clearly a `Point` needs help initializing its coördinates. For this purpose we use a special method called a *constructor*; this is how Java achieves the effect of Python's `__init__` method. A constructor has no return type; in fact it is the only method in a class which can lack a return type. When the constructor is finished, good programming practice dictates that all state variables should be initialized. Constructors are OBGYNs: they oversee the birth of objects.

Now for some grammatical ground rules. The constructor for a class must have the same name as the class. In fact, only constructors in a class may have the same name as the class. We now write a constructor for our `Point` class.

```
public class Point
{
    private int x;
    private int y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

When you are programming in a class, you *are* that object. The keyword `this`

refers to "me." The dot construct is the genetive case, so `this.x` is "my `x`." It should remind you of Python's `self`.

Now compile your class. To make a point at (3,4), call the constructor by using `new`. The `new` keyword calls the class's constructor and oversees the birth of an object.

```
jshell> Point p = new Point(3,4);
p ==> Point@26653222

jshell> Point q = new Point();
|  Error:
|  constructor Point in class Point cannot be applied to given types;
|    required: int,int
|    found: no arguments
|    reason: actual and formal argument lists differ in length
|  Point q = new Point();
|            ^---------^
```

The `Point p` is storing the point (3,4). Remember, the variable `p` itself only stored a memory address. The point (3,4) is stored at that address.

One other thing we see is that once we create a constructor the *default constructor*, which has an empty signature, no longer exists.

7.6.2 Now Let's do the Same Thing in Python

Python has special methods called *hooks* or *dunder methods* that do special jobs. These methods get this name from the fact that they are surrounded by double-underscores. We begin by meeting the the Python `__init__` dunder method that method behaves much like a Java constructor; it is called every time a new Python object of type `Point` is created.

```
class Point(object):
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

p = Point()
print (f"p = ({p.x}, {p.y})")
q = Point(3,4)
print (f"q = ({q.x}, {q.y})")
```

Now go back to the `String` class in the API guide. Scroll down to the constructor summary; this has a blue header on it and it is just above the

method summary. You will see that the string class has many constructors. How is this possible? We faked in in Python by using default arguments. Can we do this for our point class in Java?

Happily, the answer is "yes".

7.6.3 Method and Constructor Overloading

The *signature* of a Java method is an ordered list of the types of its arguments. Java supports *method overloading*: you may have several methods bearing the same name, provided they have different signatures. This is why you see several versions of `indexOf` in the `String` class. Java resolves the ambiguity caused by overloading at compile time by looking at the types of arguments given in the signature. It looks for the method with the right signature and it then calls it.

Notice that the static typing of Java allows it to support method overloading. Python achieves a similar effect using the facilities of default and keyword arguments. Here is an example of Python default arguments at work.

```
def f(x = 0, y = 0, z = 0):
        return x + y + z
print "f() = ", f()
print "f(3) = ", f(3)
print "f(3, 4) = ", f(3, 4)
print "f(3, 4, 5) = ", f(3, 4, 5)
```

```
unix> python overload.py
f() =  0
f(3) =  3
f(3, 4) =  7
f(3, 4, 5) =  12
unix>
```

You can use this principle on constructors, too. Let us now go back to our `Point` class. We will make the default constructor (sensibly enough) initialize our point to the origin.

```
public class Point
{
    private int x;
    private int y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
```

```
    }
    public Point()
    {
        this.x = 0;
        this.y = 0;
    }
}
```

Compile this class. Then type in this interactive session.

```
jshell> Point p = new Point(3,4);
p ==> Point@26653222

jshell> Point q = new Point();
q ==> Point@68c4039c
```

Voila! The default constructor is now working.

7.6.4 Get a load of `this` again!

The eleventh commandment reads, "Thou shalt not maintain duplicate code." This sounds Draconian, but it is for reasons of convenience and sanity. If you want to modify your program, you want to do the modifications in ONE place. Having duplicate code forces you to ferret out every copy of a duplicated piece of code you wish to modify. You should strive to avoid this.

One way to avoid it is to write separate methods to perform tasks you do frequently. Here, however, we are looking at our constructor. You see duplicate code in the constructors. To eliminate it, you may use the `this` keyword to call one constructor from another. We shall apply `this` here.

```
public class Point
{
    private int x;
    private int y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public Point()
    {
        this(0,0);
    }
}
```

Note that our Python class the same functionality with the aid of default arguments.

```
class Point(object):
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
```

7.6.5 Now Let Us Make this Class DO Something

So far, our `Point` class is devoid of features. We can create points, but we cannot see what their coördinates are. Now we shall provide *accessor methods* that give access to the cöordinates. While we are in here we will also write a special method called `toString`, which will allow our points to print nicely to the screen.

First we create the accessor methods. Here is how they should work.

```
jshell> Point p = new Point(3,4);
p ==> Point@26653222

jshell> Point q = new Point();
q ==> Point@68c4039c

jshell> p.getX()
$4 ==> 3

jshell> p.getY()
$5 ==> 4

jshell> q.getX()
$6 ==> 0

jshell> q.getY()
$7 ==> 0
```

Making them is easy. Just add these methods to your `Point` class.

```
public int getX()
{
    return x;
}
public int getY()
{
```

```
        return y;
    }
```

These accessor or "getter" methods allow the user of your class to see the coördinates of your `Point` but the user cannot use the getter methods to change the state of the `Point`. So far, our point class is immutable. There is no way to change its state variables, only a way to read their values.

To get your points to print nicely, create a `toString` method. Its header *must* be

```
public String toString()
```

In this method we will return a string representation for a point. Place this method inside of your `Point` class.

```
    public String toString()
    {
        return String.format("(%s, %s)", x, y);
    }
```

Compile and run. The `toString()` method of an object's class is called *automatically* whenever you print an object. Every Java object is born with a `toString()` method. We saw that this built–in method for our point class was basically useless. By implementing the `toString` method in our class, we are customizing it for our purposes. Here we see our nice representation of a `Point`.

```
jshell> Point p = new Point(3,4)
(3, 4)
jshell> System.out.println(p)
(3, 4)
jshell>
```

You will see that many classes in the standard library customize this method.

To get the same functionality in Python, we use the `str` dunder method. You can skip down to the programming exercises and do this now.

Now let us write a method that allows us to compute the distance between two points. To do this we will need to calculate a square-root. Fortunately, Java has a scientific calculator. Go to the API guide and look up the class `Math`. To use a `Math` function, just prepend its name with `Math.`; for example `Math.sqrt(5)` computes $\sqrt{5}$. All of the methods of this class are static. Many of the names of these functions are the same as they are in Python's `math` library and in and C/C++'s `cmath` and `math.h` libraries.

Now add this method to your class. You will see that it is just carrying out the distance formula between your point `(x,y)` and the point `q`.

```
    public double distanceTo(Point q)
    {
        return Math.sqrt( (x - q.x)*(x - q.x) + (y - q.y)*(y - q.y));
    }
```

Compile and run.

```
jshell> Point origin = new Point()
(0, 0)
jshell> Point p = new Point(5,12);
jshell> origin.distanceTo(p)
13.0
jshell> p.distanceTo(origin)
13.0
jshell>
```

Programming Exercises Here is a chance to try out some new territory in Python. We have already met the `init` dunder.

1. The Python dunder `__str__` tells Python to represent a Python object as a string. Its method header is

   ```
   def __str__(self):
   ```

   Make a method for the Python Point class that represents a Point as a string.

2. The Python hook `__eq__` can check for equality of objects. You can cause to points to be compared using `==` with this hook. Its header is

   ```
   def __eq__(self, other):
   ```

   Implement this for our Python `Point` class.

3. Look up the `hypot()` method in the `Math` class. How can you use it in our `Point` class? Make that method. Use snake notation for the name.

7.6.6 Who am I?

In object-oriented programming are two roles: that of the class developer and that of the client programmer. You assume both roles at once in Java, as all code in Java lives inside of classes. You are the class developer of the class you are writing, and the client programmer of the classes you are using. Any nontrivial Java program involves at least two classes, the class itself and often the class `String` or `System.out`. In practice, as you produce Java classes, you will often use several different classes that you have produced or from the Java Standard Library.

In both Python and java, while we are creating the `Point` class, we should think of ourselves as *being* `Point`s. A `Point`s knows its coördinates. Since you are a point when programming in the class `Point`, you have access to your private variables. You also have access to the private variables of any instance of class `Point`. This is why in the `distanceTo` method, we could use `q.x` and `q.y`.

In the last interactive session we made two points with the calls

```
Point origin = new Point();
Point p = new Point(5,12)
```

This resulted in the creation of two points. The call

```
p.distanceTo(origin)
```

returned 13.0. What is says is "Call `p`'s `distanceTo` method using the argument `origin`." In this case, you should think of "you" as `p`. The point `origin` is playing the role of the point `q` in the method header. Likewise, the call

```
origin.distanceTo(p)
```

is calling `p`'s distance to `origin`. In the first case, "I" is `origin`, in the second, "I" is `p`.

7.6.7 Mutator Methods

So far, all of our class methods have only looked at, but have not changed, the state of a point object. Now we will make our points mutable. To this end, create two "setter" methods and place them in your `Point` class.

```
    public void setX(int a)
    {
        x = a;
    }
    public void setY(int b)
    {
        y = b;
    }
```

Now compile and type in the following.

```
jshell> p = new Point()
(0, 0)
```

```
jshell> p.setX(5)
jshell> p
(5, 0)
jshell> p.setY(12)
jshell> p
(5, 12)
```

Our point class is now mutable: We are now giving client programmers permission to reset each of the coördinates. These new methods are called "mutator" methods, because they change the state of a `Point` object. Instances of our `Point` class are mutable, much as are Python lists. Mutability can be convenient, but it can be dangerous, too. Watch us get an ugly little surprise from aliasing.

To this end, let us continue the interactive session we started above.

```
jshell> q = p;
jshell> q
(5, 12)
jshell> q.setX(0)
jshell> p
(0, 12)
jshell> q
(0, 12)
```

Both `p` and `q` point at the same object in memory, which is initially storing the point `(5,12)`. Now we say, "`q`, set the x–coördinate of the point you are pointing at to 0. Well, `p` happens to be pointing at precisely the same object. In this case `p` and `q` are aliases of one another. If you call a mutator method from either variable, it changes the value pointed at by the other!

If we wanted `p` and `q` to be independent copies of one another, a different procedure is required. Let us now create a method called `clone`, which will return an independent copy of a point.

```
public Point clone()
{
    return new Point(x,y);
}
```

Compile and fire up a new `jshell` session. Now we will test–drive our new `clone` method. We will make a point `p`, an alias for the point `alias`, and a copy of the point `copy`.

```
jshell> Point p = new Point(3,4)
(3, 4)
```

```
jshell> Point alias = p
(3, 4)
jshell> Point copy = p.clone();
jshell> p
(3, 4)
```

Continuing, let us check all possible equalities.

```
jshell> p == alias
true
jshell> copy == alias
false
jshell> p == copy
false
```

We can see that `p` and `q` are in fact aliases the same object, but that `alias` is not synonymous with either `p` or `q`.

```
jshell> p
(3, 4)
jshell> alias
(3, 4)
jshell> copy
(3, 4)
```

All three point at a point stored in memory that is `(3,4)`. Now let us call the mutator `setX` on `p`; we shall then inspect all three.

```
jshell> p.setX(0)
jshell> p
(0, 4)
jshell> alias
(0, 4)
jshell> copy
(3, 4)
```

The object pointed to by both `p` and `q` was changed to `(0,4)`. The copy, however, was untouched.

Look at the body of the `clone` method. It says

```
return new Point(x,y);
```

This tells Java to make an entirely new point with coördinates `x` and `y`. The call to `new` causes the constructor to spring into action and stamp out a fresh, new `Point`.

7.7 Scope of Java Variables

In this section, we shall describe the lifetime and visibility of Java variables. The rules differ somewhat from Python, and you will need to be aware of these differences to avoid unpleasant surprises.

So fare, we have met two kinds of variables in object-oriented programming, state variables and *local* variables. Local variables are variables created inside of any method or function. State variables are visible anywhere in a class; in Python they are visible via the `self` reference. It is best in Python to initialize all state inside of the `init` dunder method.

In Java, where they are declared in a class is immaterial, but you should declare them at the top of your class. This makes them easy to find and manage. You could move them to the end of the class with no effect.

The rest of our discussion pertains to local variables. All local variables in Java have a *block*; this is delimited by the closest pair of matching curly braces containing the variable's declaration. The first rule is that *no local variable is visible outside of its block.* The second rule is that *a local variable is not visible until it is created.* You will notice that these rules are stricter than those of Python. As in Python, variables in Java are not visible prior to their creation; this rule is exactly the same.

Here is an important difference. Variables created inside of Python functions are visible from their creation to the end of the function, even if they are declared inside of a block in that function. Here is a quick example in a file named `laxPyScope.py`.

```
def artificialExample(x):
    k = 0
    while k < len(x):
        lastSeen = x[k]
         k += 1
    return lastSeen
x = "parfait"
print "artificialExample(" + x + ") = ", artificialExample(x)
```

It is easy to see that the function `artificialExample` simply returns the last letter in a nonempty string. We run it here.

```
$ python laxPyScope.py
artificialExample(parfait) =  t
$
```

Observe that the variable `lastSeen` was created inside a block belonging to a `while` loop. In Java's scoping rules, this variable would no longer be visible (it

would be destroyed) as soon as the loop's block ends. This happens because, when a block closes, all varables inside of it are popped off the stack.

There are some immediate implications of this rule. Any variable declared inside of a method in a class can only be seen inside of that method. That works out the same as in Python. Let us code up exactly the same thing in Java in a class `StrictJavaScope`. In this little demonstration, you will see Java's `while` loop at work.

```
public class StrictJavaScope
{
    public char artificialExample(String x)
    {
        int k = 0;
        while( k < x.length())
        {
             char lastSeen = x.charAt(k);
             k += 1;
        }
        return lastSeen;
    }
}
```

Now compile and brace yourself for compiler grumblings.

```
 javac StrictJavaScope.java
StrictJavaScope.java:11: error: cannot find symbol
        return lastSeen;
               ^
  symbol:   variable lastSeen
  location: class StrictJavaScope
1 error
```

Your symbol `lastSeen` died when the `while` loop ended. Even worse, it got declared each time the loop was entered and died on each completion of the loop.

How do we fix this? We should declare the `lastSeen` variable before the loop. Then its block is the entire function body, and it will still exist when we need it. Here is the class with repairs effected.

```
public class StrictJavaScope
{
    public char artificialExample(String x)
    {
        int k = 0;
```

```
        char lastSeen = ' ';
        while( k < x.length())
        {
             lastSeen = x.charAt(k);
             k += 1;
        }
        return lastSeen;
    }
}
```

Peace now reigns in the valley.

```
jshell> s = new StrictJavaScope();
jshell> s.artificialExample("parfait")
't'
jshell>
```

`while` We are at it The use of the `while` loop is entirely natural to us and it looks a lot like Python. There are some differences and similarities. The differences are largely cosmetic and syntactical. The semantics are the same, save of this issue of scope we just discussed.

- **similarity** The `while` statement is a boss statement. No mark occurs in Java at the end of a boss statement.
- **difference** Notice that there is NO colon or semicolon at the end of the `while` statement. Go ahead, place a semicolon at the end of the `while` statement in the example class. It compiles. Run it. Now figure out what you did, Henry VIII.
- **difference** Notice that predicate for the `while` statement is enclosed in parentheses. This is required in Java; in Python it is optional.
- **similarity** The `while` statement owns a block of code. This block can be empty; just put an empty pair of curly braces after the loop header.

The scoping for methods and state variables is similar. State variables have *class scope* and they are visible from anywhere inside of the class. They may be modified by any of the methods of the class. Any method modifying a state variable is a mutator method for the class. Be careful when using mutator methods, as we have discussed some of their perils when we talked about aliasing. A good general rule is that if a class creates small objects, give it no mutator methods. For our `Point` class, we could just create new `Point`s, rather than resetting coördinates. Then you do not have to think about aliasing. In fact, it allows you to share objects among variables freely and it can save space. It also eliminates the need for copying objects.

Later, we will deal with larger objects, like graphics windows and displays. We do not want to be unnecessarily calling constructors for these large objects and we will see that these objects in the standard library have a lot of mutator methods.

All methods are visible inside of the class. To get to methods outside of the class, you create an instance of the class using `new` and call the method via the instance. Even if your state variables are (foolishly) `public`, you must refer to them via an instance of the class. Let us discuss a brief example to make this clear.

Suppose you have a class `Foo` with a method called `doStuff()` and public a public state variable `x`. Then to get at `doStuff` or `x` we must first create a new `Foo` by calling a constructor. In this example we will use the default.

```
Foo f = new Foo();
```

Then you can call `doStuff` by making the call

```
f.doStuff();
```

Here you are calling `doStuff` via the instance `f` of the class `Foo`. To make `f`'s `x` be 5, we enter the code

```
f.x = 5;
```

Notice that the "naked" method name and the naked variable name are not visible outside of the class. In practice, since all of our state variables will be marked `private`, no evidence of state variables is generally visible outside of any class.

7.8 The Object-Oriented Weltanschauung

Much emphasis has been placed here on classes and objects. In this section we will have a discussion of programming using objects. We will begin by discussing the procedural programming methods we developed in Chapters 0-7 of the Python book.

7.8.1 Procedural Programming

When we first started to program in Python, we wrote very simple programs that consisted only of a main routine. These programs carried out small tasks and were short so there was little risk of confusion or of getting lost in the code.

As we got more sophisticated, we began using functions as a means to break down, or modularize, our program into manageable pieces. We would then code each part and integrate the functions into a program that became a coherent whole. Good design of programs is "top down." You should nail down what you are trying to accomplish with your program. Then you should break the program down into components. Each component could then be broken into smaller components. When the components are of a manageable size, you then code them up as functions.

To make this concrete, let us examine the case of writing a program that accepts a string and which looks through an English word list, and which shows all anagrams of the string you gave as input which appear in the word list.

To do this, you could write one monolithic procedure. However, the procedure would get pretty long and it would be trying to accomplish many things at once. Instead we might look at this and see we want to do the following

- Obtain the word from the user.
- Lower-case the word and permute the letters so they are in alphabetical order
- Open a word list file.
- for each word in the list:
  - Lower-case each word in the wordlist and put it in alphabetical order.
  - If you get a match, put the word on an output list
- Return the list of words we obtained to the user.

Not all the tasks here are of the same difficulty. The first one, obtain the word from the user, is quite easy to do. We, however have to make a design decision and decide how to get the word from the user. This is a matter of deciding the program's *user interface*.

Python is an object-oriented language like Java with a library of classes. Many of the Python classes can save you great gobs of work; the same is true in Java. Always look for a solution to your problem in the standard library before trying to solve it yourself! If you create classes intelligently, you will see that you will be able to reuse a lot of code you create.

Returning to our problem, you would revisit each sub-problem you have found. If the sub-problem is simple enough to write a function for it, code it. Otherwise, break it down further.

This is an example of top–down design for a *procedural* program. We keep simplifying procedures until they become tractable enough to code in a single function. We program with *verbs*.

The creation of functions gives us a layer of abstraction. Once we test our functions, we use them and we do not concern ourselves with their internal details (unless a function goes buggy on us), we use them for their *behavior*. Once a function is created and its behavior is known, we no longer concern ourselves with its local variables and the details of its implementation.

This is an example of *encapsulation*; we are thinking of a function in terms of its behavior and not in terms of its inner workings.

7.8.2 Object–Oriented Programming

In object–oriented programming, we program with *nouns*. A class is a sophisticated creature. It creates *objects*, which are computational creatures that have state, identity and behavior. We shall see here that encapsulation plays a large role in object-oriented programming. Good encapsulation dictates that we hide things from client programmers they do not need to see. This is one reason we make our state variables private. We may even choose to make certain methods private, if the do think they are of real use other than that of a service role the other methods of the class.

You still do top-down design, but you begin by thinking about what kind of objects the task at hand entails. This prompts you to think about the classes you can use from the standard libraries and those you need to write yourself. You must think about the ways in which they interact.

For each class, you have to think about what state it needs to maintain, and what methods it should have so it can do its job properly. We arrive here in much more complex world than that of procedural programming.

When you used the `String` class, you did not need to know how the characters of a `String` are stored. You do not need to know how the `substring()` method works: you merely know the behavior it embodies and you use it. What you can see in the API guide is the *interface* of a class; this is a class's public portion. You are a client programmer for the entire standard library.

What you do not see is the class's *implementation*. You do not know how the `String` class works internally. You could make a good guess. It looks as if a list of characters that make up the string is stored somewhere. That probably reflects the state of a `String` object.

A string, however, is a fairly simple object. The contents of a window in a graphical user interface (GUI) in Java is stored in an object that is an instance of the class `Stage`. How do you store such a thing? Is it different on different platforms? All of a sudden we feel the icy breath of the possibly unknown....

However, there is nothing to fear! In Java the `Stage` class has behaviors that allow you to work with a frame in a GUI and you do not have to know *how* the internal details of the `Stage` work. This is the beauty of encapsulation. Those details are thankfully hidden from us!

For an everyday example let us think about driving a car. You stick in the key, turn it, and the ignition fires the engine. You then put the car in gear and drive. Your car has an interface. There is the shifter, steering wheel, gas pedal, the music and climate controls, the brakes and the parking brake. There are other interface items such as door locks, seat adjusters, and the dome light switch.

These constitute the "public" face of your car. You work with this familiar interface when driving. It is similar across cars. Even if your car runs on diesel, the interface you use to drive is not very different from that of a gasoline–fueled car.

You know your car has "private parts" to which you do not have direct access when driving. Your gas pedal acts as a mutator method does; when you depress it, it causes more gas to flow to the fuel injection system and causes the RPM of the engine to increase. The RPM of the engine is a state variable for your car. Your tachometer (if your car has one) is a getter method for the RPM of your engine. You affect the private parts (the implementation) of your car only indirectly. Your actions occur through the interface.

Let's not encapsulate things for a moment. Imagine if you had to think about *everything* your car does to run. When you stick your key in the ignition, if you drive a newer car, an electronic system analyzes your key fob to see if your key is genuine. That then triggers a switch that allows the ignition to start the car. Then power flows to the starter motor..... As you are tooling down the highway, it is a safe bet you are not thinking about the intricacies of your car's fuel injection system and the reactions occurring in its catalytic converter. You get the idea. Encapsulation in classes simplifies things to a manageable essence and allows us to think about the problem (driving here) at hand. You use the car's interface to control it on the road. This frees your mind to think about your actual driving.

So, a Java program is one or more classes working together. We create instance of these classes and call methods via these instances to get things done. In the balance of this book, you will gain skill using the standard library classes. You will learn how to create new classes and to create extensions of existing ones. This will give you a rich palette from which to create programs.

Exercises

1. Think about your bicycle. What constitutes its interface?
2. What is the interface to your computer? How do you interact with it and

control it? What are some of its "private parts"?

3. How about a takeout pizza joint? How do you interact with it? What are some of its public and private parts?

Chapter 8

Python $\longrightarrow$ Java

8.0 Introduction

We are going to frame the concepts we learned in Python in Java. During this chapter, we will do a comparison of the design and mechanics of the two languages.

8.1 Java Data Structures

Recall that a data structure is a container in which we store a collection of related objects under a single name. Different data structures have different organizations and different rules for accessing and manipulating their contents. In Python, we met the data structures `list`, `tuple`, and `dict`. Python lists are mutable heterogeneous sequences; they can contain objects of any type as entries. Python tuples are like lists, but they are immutable; list methods that change list state cannot be used on tuples. Python dictionaries allow us to store key-value pairs. Python data structures grow according to our needs and they shrink when we delete items from them.

Java has two data types comparable to Python lists. We begin by learning about the *array*; it is a *homogeneous* mutable sequence type of *fixed* size. When you create an array you specify its size and the type of entries it contains. If you run out of room and wish to add more entries to an array, you must create a new, bigger array, copy your array into its new home, and then abandon the old array. Before abandoning the old array, you will have do certain housekeeping chores so that all abandoned objects get garbage-collected. Arrays can be of primitive or object type. An array itself is an object. The syntax for declaring an array of type `type` is

```
type[] identifier;
```

Open a new `jshell` session. We will use our first `import` statement here. The `import` statement works much as it does in Python. Importing the class `java.util.Arrays` will give us a convenient way to print the contents of an array; the built-in string representation of an array is useless.

Let us begin by declaring a variable of integer array type.

```
jshell> import java.util.Arrays;
jshell> int[] x;
x ==> null
```

Now let's try to assign something to an entry.

```
jshell> x[0] = 1
|  java.lang.NullPointerException thrown
|        at (#2:1)
```

We are greeted by a surly error message. Here is one sure reason why.

```
jshell> Arrays.toString(x)
"null"
```

Right now, the array variable is pointing at Java's "graveyard state" `null`. If you attempt to use a method on a object pointing at `null`, you will get a run time error called a `NullPointerException`. We need to give the array some actual memory to point to; this is where we indicate the array's size.

We call the special array constructor to attach an actual array to the array pointer `x`. After we attach the array, notice how we obtain the array's length.

```
jshell> x = new int[10];
jshell> Arrays.toString(x)
"[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]"
jshell> x.length
10
```

Observe also that Java politely placed a zero in each entry in this integer array. This will happen for any primitive numerical type. If you make an array of booleans, it will be populated with the value `false`. In a character array, watch what happens.

```
jshell> char[] y = new char[10];
jshell> Arrays.toString(y)
```

```
"[, , , , , , , , , ]"
jshell> (int) y[0]
0
```

The array is filled with the *null character* which has ASCII value 0. It is not a printable character. An array of object type is filled with `null`s. Typically, you will need to loop through the array to attach an object or primitive to each entry.

Arrays have indices, just as lists do in Python. Remember, you should think of the indices as living *between* the array entries. Arrays know their length, too; just use `.length`. Notice that this is *not* a method and it is an error to use parentheses at the end of it.

8.1.1 `java.util.Arrays`

The convenience class `java.util.Arrays` is a "service class" that has useful methods for working with arrays. We will demonstrate some of its methods here. If you are working on arrays, look to it first as as a means of doing routine chores with arrays. Its methods are fast, efficient and tested. It is a nice exercise to re-create some of them, but don't needlessly reinvent the wheel.

Go to the API page; you will see some useful items there. Here is a summary of the most important ones for us. We will use `Type` to stand for a primitive or object type. Hence `Type[]` means an array of type `Type`. You call all of these methods by using `Arrays.`*method(arg(s))*.

Header	Action
`Type[] copyOf(Type[] original, int newLength)`	This copies your array and returns the copy. If the newLength is shorter than your array, your array is truncated. Otherwise, it is lengthened and the extra entries are padded with the default value of `Type`.
`Type[] copyOfRange(Type[] original, int from, int to)`	This returns a copy of a slice of your original array, between indices `from` and `to`. Using illegal entries generates a run time error.
boolean equals(Type[] array1, Type[] array2)	This returns `true` if the two arrays have the same length and contain the same values in the same order. It works just like Python's == on lists.
`void fill(Type[] array, Type value)`	This will replace all of the entries in the array `array` with the value `value`.
`void fill(Type[] array, int from, int to, Type value)`	This will replace the entries between indices `from` and `to` with the value `value`.
`String toString(Type[] array`	This pretty-prints your array as a string. You have seen this used.

8.1.2 Fixed Size? C'mon!

We now introduce a new class and a new piece of Java syntax. An `ArrayList` is a variable-size array. There are two ways to work with `ArrayList`s and we will show them both.

Let us create an `ArrayList` and put some items in it. To work with an `ArrayList` you will need to import the class `java.util.ArrayList`. The `import` statement in the interactive session below shows how to make the class visible. `java.util.ArrayList` is the fully–qualified name of this class. The import statement puts us on a "first-name" basis with the class.

How do I know what to import? Look the class `ArrayList` up in the API guide. Near the top of the page, you will see this.

> java.util
> ArrayList<E>

This tells you that the `ArrayList` class lives in the package `java.util`. Therefore you should place this at the top of your code.

```
import java.util.ArrayList;
```

Do not put the `<E>` in the `import` statement.

We will use the `ArrayList`'s `add` method to place new items on the list we create.

```
jshell> import java.util.ArrayList;
jshell> ArrayList pool = new ArrayList();
jshell> pool.add("noodle")
|  Warning:
|  unchecked call to add(E) as a member of the raw type java.util.ArrayList
|  pool.add("noodle")
|  ^----------------^
$5 ==> true

jshell> pool.add("chlorine")
|  Warning:
|  unchecked call to add(E) as a member of the raw type java.util.ArrayList
|  pool.add("chlorine")
|  ^------------------^
$6 ==> true

jshell> pool.add("algicide")
|  Warning:
|  unchecked call to add(E) as a member of the raw type java.util.ArrayList
|  pool.add("algicide")
|  ^------------------^
$7 ==> true

jshell> pool
pool ==> [noodle, chlorine, algicide]

jshell> pool.get(0)
$9 ==> "noodle"
```

All looks pretty good here, save for the surly warnings. But then there is an irritating snag.

```
jshell> pool.get(0).charAt(0)
|  Error:
|  cannot find symbol
|    symbol:   method charAt(int)
|  pool.get(0).charAt(0)
|  ^----------------^
```

8.1.3 What is this `Object`?

To explain what just happened to us properly, we will take a look into the near future that lurks in Chapter 5. Every object of object type in Java, logically enough, is an `Object`. Go into the API guide and look up the class `Object`.

Every Java class has a place in the Java class hierarchy, including the ones you create. What is different from human family trees is that a Java class has *one* parent class. A Java class can have any number of children. This hierarchy is independent of the hierarchical structure imposed on the Java Standard Library by packages or modules.

The Java class hierarchy is an Australian (upside–down) tree, just like your file system. In LINUX, your file system has a root directory called `/`. In the Java class hierarchy, the class `Object` is the root class.

Heretofore, we have created seemingly stand–alone classes. Our classes, in fact have not really been "stand–alone." Automatically, Java enrolls them into the class hierarchy and makes them children of the `Object` class. This is why every object has `toString()` and `equals()` methods, even if you never created them.

The only stand–alone types are the primitive types. They are entirely outside of the Java class hierarchy. However, we have seen that these too, have `Object` analogs in the form of the eight wrapper classes.

What is entailed in this parent–child relationship? The child *inherits* the public portion of the parent class. In a human inheritance, the heirs can decide what to do with the property they receive. They can use the property for its original purpose or redirect it to a new purpose. In Java, the same rule applies. When we made a `toString()` method for our `Point` class, we decided to redirect our inheritance. Every Java object is born with a `toString()` method. Unfortunately the base `toString()` method gives us a default string representation of our object that looks like this.

```
ClassName@ABunchOfHexDigits
```

We decided this is not terribly useful so we *overrode* the *base* `toString()` method and replaced it with our own. To override a method in a parent class, just re–implement the method, with exactly the same signature, in the child class. We also overrode the `clone()` method in the parent class. If you intend to copy objects, do not trust the `clone()` you inherit from `Object`.

This table describes the methods in the `Object` class and the relevance of each of them to us now.

Object Method	Description
`clone()`	This method creates and returns a copy of an object. You should override this if you intend to use independent copies of instance of your class. Do not implement this if your class creates immutabled objects.
`finalize()`	This method is automatically called when the garbage collector arrives to reclaim the object's memory. We will rarely if ever use it. Its explicit use is now deprecated.
`getClass()`	This method tells you the class that an object was created from.
`notify()`	This method is used in threaded programs. We will deal with this much later

The three `wait` methods and the `notifyAll` methods all apply in threaded programming. Threads allow our programs to spawn sub-processes that run independently of our main program. Since these are a part of `Object`, this tells you that threading is built right into the core of the Java language. We will develop threading much later.

8.1.4 Back to the Matter at Hand

Everything is returned from an `ArrayList` is an `Object`. Strings have a `charAt()` method, but an `Object` does not. As a result you must perform a cast to use things you get from an `ArrayList`. Here is the (ugly) syntax. Ugh. It's as ugly as Scheme or Lisp.

```
jshell> ((String)pool).get(0).charAt(0)
'n'
jshell>
```

This is the way things were until Java 5. Now we have *generics* that allow us to specify the type of object to go in an `ArrayList`. Generics make a lot of the ugliness go away. The small price you pay is you must specify a type of object you are placing in the `ArrayList`. The type you specify is placed in the *type parameter* that goes inside the angle brackets `< .... >`. You may use any object type as a type parameter; you may not do this for primitive types.

```
ArrayList<String> farm = new ArrayList<String>();
jshell> farm.add("cow")
true
jshell> farm.get(0).charAt(0)
'c'
jshell>
```

Here is something new to Java 7. Generics now have a feature called *type inference* that, among other things, makes creating array lists simpler. We now show the Java 7 way to do what we just did above.

```
ArrayList<String> farm = new ArrayList<>();
jshell> farm.add("cow")
true
jshell> farm.get(0).charAt(0)
'c'
jshell>
```

Notice you do not have to specify the type parameter on the right hand side. Java infers it for you.

Warning: Deception Reigns King Here! All here has a pleasing cosmetic appearance. However, it's time to take a peek behind the scenes and see the real way that generics work.

What happens behind the scenes is that the *compiler* enforces the type restriction. It also automatically inserts the needed casts for the get() method. Java then erases all evidence of generics prior to run time.

1. You make an `ArrayList` of some type, say `String` by using the `ArrayList<String>` syntax.
2. You put things on the list with `add` and friends and gain access to them with the `get()` method.
3. The compiler will add the necessary casts to `String` type when you refer to the entries of the `ArrayList` using `get()`, removing this annoyance from your code.
4. The compiler then performs *type erasure*; it eliminates all mention of the type parameter from the code, so to the run time environment, `ArrayLists` look like old–style `ArrayLists` at run time.

This is a smart decision for two reasons. One reason is that it prevents legacy code from breaking. That code will get compiler growlings and warnings about "raw types" but it will continue to work.

Secondly, if you declare `ArrayLists` of various type, each type of `ArrayList` does not generate new byte code. If you are familiar with C++, you may have heard that C++'s version of generics, *templates*, causes "code bloat;" each new type declared using a C++ template creates new object code in your executable file. Because of type erasure, Java does not do this.

Let us now make a sample class that takes full advantage of generics. First, let us make a version without generics and see something go wrong.

```
import java.util.ArrayList;
public class StringList
{
    private ArrayList theList;
    public StringList()
    {
        theList = new ArrayList();
    }
    public boolean add(String newItem)
    {
        return theList.add(newItem);
    }
    public String get(int k)
    {
        return theList.get(k);
    }
}
```

Compile this program and you will get a nastygram like this.

```
javac StringList.java
StringList.java:15: error: incompatible types:
    Object cannot be converted to String
        return theList.get(k);
                           ^
Note: StringList.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
1 error
```

The error is that we are advertising that `get` returns a `String`; the `ArrayList`'s `get()` only returns an `Object`. Now let us add the type parameter `<String>` to the code. Your code compiles. Let us now inspect our class interactively. We can now cast aside our worries about casts.

```
jshell> StringList greats = new StringList();
jshell> greats.add("Babe Ruth")
true
jshell> greats.add("Mickey Mantle")
true
jshell> greats.add("Lou Gehrig")
true
jshell> greats.get(0)
"Babe Ruth"
jshell> greats.get(0).charAt(0)
'B'
jshell>
```

You can see that `greats.get(0)` in fact returns a `String`, not just an `Object`, since it accepts the `charAt()` message.

Programming Exercises These exercises will help familiarize you with the `ArrayList` API page. This class offers an abundance of useful services. Try these in a `jshell` session.

1. Make a new `ArrayList` of strings named `roster`.
2. Add several lower-case words to the `ArrayList`; view its contents as you add them.
3. How do you compute the number of elements of an `ArrayList`?
4. How can you determine if a given string is in your `ArrayList`?
5. Enter this import statement: `import java.util.Collections`.
6. Type this command `Collections.sort(roster);` Tell what happens.
7. Type this command `Collections.shuffle(roster);` Tell what happens.
8. Add some upper-case words. Add some strings with numbers and symbols. How do they behave when you use `Collections.sort()`? What definitive conclusion can you surmise?
9. Another generic class in Java is `HashSet<E>`. Create a `HashSet<String>`. Add the same string twice. What got returned? What happened? Add more items. What do you notice? Look this class up in the API guide and do some spelunking.

8.2 Conditional Execution

Java, like Python or any other self-respecting computer language, supports conditional execution. Python has `if`, `elif` and `else` statements. These are all boss statements. All of this is the works the same way in Java, but the appearance is a little different. Here is a comparison method called `ticketTaker` in Python and Java. First we show the Python version.

```
def ticketTaker(age):
    if age < 13:
        print("You may only see G movies.")
    elif age < 17:
        print("You may only see PG or G movies.")
    elif age < 18:
        print("You may only see R, PG, or G-rated movies.")
    else:
        print("You may see any movie.")
```

The Java version is quite similar. The keywords change a bit. Notice that the predicates are enclosed in parentheses. This is required. Observe in this example that you can put a one–line statement after an `if`, `else if` or `else` without using curly braces. If you want one more than one line or an empty block attached to any of these, you must use curly braces. It is best to always use curly braces for bodies of boss statement in Java; this eliminates a lot of frustrating error messages from the compiler and a lot of irksome logic errors in your code.

```
public void ticketTaker(int age)
{
    if (age < 13)
    {
        System.out.println("You may only see G movies.");
    }
    else if (age < 17)
    {
        System.out.println("You may only see PG or G- movies.");
    }
    else if (age < 18)
    {
        System.out.println("You may only see R, PG or G movies.");
    }
    else
    {
        System.out.println("You may see any movie.");
    }
}
```

Both languages support a ternary statement. We shall illustrate it in an absolute value function for both languages. First here is the Python version.

```
def abs(x):
    return x if x >= 0 else -x
```

Now we show Java's ternary operator at work.

```
public int abs(int x)
{
    return x >= 0 ? x : -x;
}
```

Use parentheses to keep the order of operations from producing undesired results where necessary.

Java supports an additional mechanism, the `switch` statement for conditional execution. We show an example of this statement and then explain its action.

```
public class Stand
{
    public String fruit(char c)
    {
        String out = "";
        switch(c)
        {
            case 'a': case 'A':
                out = "apple";
                break;
            case 'b': case 'B':
                out = "blueberry";
                break;
            case 'c': case 'C':
                out = "cherry";
                break;
            default:
                out = "No fruit with this letter";
        }
        return out;
    }
}
```

Let us now instantiate the `Stand` class and test its `fruit` method.

```
jshell> s = new Stand()
Stand@6504bc
jshell> s.fruit('A')
"apple"
jshell> s.fruit('b')
"blueberry"
jshell> s.fruit('z')
"No fruit with this letter"
jshell>
```

The `switch`–`case` statement only allows you to switch on a variable of *integral type*, i.e. an integer or character type. Java 7 or later additionally allows you to switch on a `String`.

8.2.1 The New `switch` Statement

Starting in Java14 (fall 2020), you can use the new `switch` statement. It is simpler, has a clean appearance, and it dispenes with the `break`s. It is good to know about the old one since it will be all over OPC.

public class Stand public String fruit(char c) String out = ""; switch(c) case 'a', 'A': -> out = "apple"; case 'b', 'B': -> out = "blueberry"; case 'c', 'C': -> out = "cherry"; default -> out = "No fruit with this letter"; return out;

The `switch` construct cannot be used on variables of floating-point type. Clearly this is a consequence of the fact that floating-point numbers are not stored exactly and that equality comparisons between them are not at all recommended. Do not use it on a `boolean` variable; for these, we use the `if` machinery.

In the old `switch`, at the end of each row of one or more `case`s, you place zero or more lines of code followed by a `break` statement. Remove various `break` statements and note the behavior of the function. You will see that they play an important role. If you do not like `switch–case`, you can live without it with little or no deleterious effect. This new `switch` statement is much more appealing than the old one.

8.3 Big Integers

We shall introduce a new class, `BigInteger`, which does extended–precision integer arithmetic. Go into the Java API guide and bring up the page for `BigInteger`. Just under the main heading

> java.math
> Class BigInteger

you well see this class's family tree. Its parent is `java.lang.Number` and its grandparent is `java.lang.Object`. The fully–qualified name of the class is `java.math.BigInteger`. To use the class, you will need to put the import statement

```
import java.math.BigInteger;
```

at the top of your program. You can always look at the bottom of the family tree to see what import statement is needed.

Remember that you never need to import any class that is in `java.lang`, such as `java.lang.String`. These are automatically imported for you. Python seamlessly integrates super-long integers into the language. This is not so in

Java. Java class developers cannot override the basic operators like +, -, * and /.

Begin by looking the Constructor summary. The most useful constructor to us seems to be

```
BigInteger(String val)
```

Now we shall experiment with this in an interactive session.

```
jshell> import java.math.BigInteger;
jshell> p = new BigInteger("1");
jshell> p
1
jshell>
```

We now have the number 1 stored as a `BigInteger`. Continuing our session, we attempt to compute 1 + 1.

```
jshell> p + p
Error: Bad type in addition
jshell>
```

In a program this would be a compiler error. Now go into the method summary and look for `add`.

```
jshell> p.add(p)
2
jshell> p
1
jshell>
```

The `add` method computes `1 + 1` in `BigInteger` world and comes up with `2`. Notice that the value of `p` did not change. This is no surprise, because `BigInteger`s are immutable.

To find out if a class makes immutable objects, look in the preface on its page in the API guide. First you see the header on this page, then the family tree. Then there is a horizontal rule, and you see the text

```
public class BigInteger
extends Number
implements Comparable<BigInteger>
```

The phrase "`extends Number`" just means that the `Number` class is the parent of `BigInteger`. We will learn what "implements" means when we deal with interfaces; we do not need it now.

Next you see the preamble, which briefly describes the class. Here it says "Immutable arbitrary–precision integers." So, as with strings, you must orphan what a variable points at to get the variable to point at anything new. Now let us see exponentiation, multiplication, subtraction and division at work.

```
jshell> import java.math.BigInteger;
jshell> a = new BigInteger("1341121");
jshell> BigInteger b = a.pow(5);
jshell> a
1341121
jshell> b
4338502129107268229778644529601
jshell> BigInteger c = b.multiply(new BigInteger("100"))
433850212910726822977864452960100
jshell> BigInteger d = a.subtract(new BigInteger("1121"));
jshell> d
1340000
jshell> d.divide(new BigInteger("1000"))
1340
jshell>
```

It would be convenient to have a way to convert a regular integer to a big integer. There is a method

```
static BigInteger valueOf(long val)
```

To call this (static) method, the usage is

```
BigInteger.valueof(whateverIntegerYouWantConverted)
```

The `BigInteger.valueOf()` method is called a *static factory method*; it is a "factory" that converts regular integers into their bigger brethren.

We now show an example or two. Be reminded of the need to use the `equals` method when working with variables pointing at objects, so you do not get a surprise.

```
jshell> import java.math.BigInteger;
jshell> p  = BigInteger.valueOf(3)
3
jshell> q = new BigInteger("3")
3
jshell> p == q
false
jshell> p.equals(q)
true
jshell>
```

8.4 Recursion in Java

Java supports recursion, and subject to the new syntax you have learned, it works nearly the same way as it did in Python. All of the pitfalls and benefits you learned about in Python apply in Java. Let us write a factorial function using the `BigInteger` class Recall the structure of the factorial function in Python.

```
def factorial(n):
    return 1 if n <= 0 else n*factorial(n - 1)
```

Everything was so simple and snappy.

Now we have to convert this to Java using the operations provided by `BigInteger`. We do have some tools at hand. `BigInteger.valueOf()` converts regular integers into their bigger brethren. We also have to deal with the `.multiply` syntax to multiply. Finally, we must remember, *we are returning a* `BigInteger`. Bearing all those consideration in mind, you should get something like this. If the ternary operators is not quite to your taste, use an `if` statement instead. We have broken the big line here solely for typographical convenience.

```
import java.math.BigInteger;

public class Recursion
{
    public BigInteger factorial(int n)
    {
        return n > 0 ?
            factorial(n - 1).multiply(BigInteger.valueOf(n)):
            BigInteger.valueOf(1);
    }
}
```

Now let us test our function.

```
jshell> r = new Recursion();
jshell> r.factorial(6)
720
jshell> r.factorial(100)
933262154439441526816992388562667004907159682
643816214685929638952175999932299156089414639
761565182862536979208272237582511852109168640
000000000000000000000
jshell> r.factorial(1000)
40238726007709 ... (scads of digits)  ...00000
jshell>
```

Recursion can be used as a repetition mechanism. We add a second method `repeat` to our class to character or string is passed it any specified integer number of times to imitate Python's `string * int` repeat mechanism. This will serve as a nice example of method overloading. First let us work with the `String` case. Let us call the String `s` and the integer `n`. If `n <= 0`, we should return an empty string. Otherwise, let us glue a copy of `s` to the string `repeat(s, n - 1)`

```
public String repeat(String s, int n)
{
    String out = "";
    if(n > 0)
    {
        out += s + repeat(s, n - 1);
    }
    return out;
}
```

Now we get the character case with very little work.

```
    public String repeat(char ch, int n)
    {
        return repeat("" + ch, n)
    }
```

Now our `repeat` method will repeat a character or a string. We do not need to worry about the character or string we need to repeat. Method overloading makes sure the right method is called.

8.5 Looping in Java

We have already seen the `while` loop in Java. It works in a manner entirely similar to Python's `while` loop. For your convenience, here is a quick comparison

```
while predicate:
    bodyOfLoop

while(predicate)
{
    bodyOfloop
}
```

It looks pretty much the same. All of the same warnings (beware of hanging and spewing) apply for both languages. Note that the predicate of a `while` loop is enclosed in parentheses.

Java also offers a second version of the `while` loop, the `do-while` loop. Such a loop looks like this.

```
do
{
    bodyOfloop
}
while(predicate);
```

The body of the loop executes unconditionally the first time, then the predicate is checked. What is important to realize is that the predicate is checked *after* each execution of the body of the loop. When the predicate evaluates to `false`, the loop's execution ends. Almost always, you should prefer the `while` loop over the `do-while` loop. When using this loop, take note of the semicolon; you will get a nastygram from Mr. Compiler if you omit it.

Java has two versions of the `for` loop. One behaves somewhat like a variant of the `while` loop and comes to us from C/C++. The other is a definite loop for iterating through a collection.

First let us look at the C/C++ `for` loop; its syntax is

```
for(initializer; test; between)
{
    loopBody
}
```

This loop works as follows. The `initializer` runs once at when the loop is first encountered. The initializer may contain variable declarations or initializations. Any variable declared here has scope only in the loop.

The `test` is a predicate. Before each repetition of the loop, the `test` is run. If the `test` fails (evaluates to `false`), the loop is done and control passes beyond the end of the loop. If the `test` passes, the code represented by `loopBody` is executed. The `between` code now executes. The `test` predicate is evaluated, if it is true, the `loopBody` executes. This process continues until the `test` fails, at which time the loop ends and control passes to the line of code immediately beyond the loop. This loop is basically a modified `while` loop.

Java also has a `for` loop for collections that works similarly to Python's `for` loop. Observe that the loop variable `k` is an iterator, just as it is in Python's `for` loop. It has a look-but-don't-touch relationship with the entries of the array, just as Python does. It grants access but does not allow mutation. This works for both class and primitive types.

```
import java.util.ArrayList;
jshell> ArrayList<String> cats = new ArrayList<String> ();
```

```
jshell> cats.add("siamese")
true
jshell> cats.add("javanese")
true
jshell> cats.add("manx")
true
jshell> for(String k : cats){System.out.println(k);}
siamese
javanese
manx
jshell> for(String k : cats){k = "";}//Look, but don't touch!
jshell> for(String k : cats){System.out.println(k);}
siamese
javanese
manx
```

8.6 Starguments for Java

Recall that Python has starguments; these must go after positional arguments. Here is a quick example

```
def product(*factors)
    out = 1
    for k in factors:
        out *= k
    return out
```

Python treats ths starred argument (stargument) like a list. Java has a similar feature.

We have seen several methods with a mysterious ellipsis (...) in their argument lists. Such arguments are called *variable-length arguments*, or *varargs*, for short. Two familiar examples come to mind. The method `System.out.printf` has signature `[String, Object...]` and `Arrays.asList` has the header

```
static <T> List<T> Arrays.asList(T...)
```

Use of these is similar. The call

```
System.out.printf("The square of %s is %s.\n", 5, 5*5);
```

puts

```
The square of 5 is 25.
```

to `stdout`. As you are familiar with this, you need one object to be printed for each format specifier in the format string. The ellipsis is indicative of a variable size list of arguments of the indicated type. Similarly if you make the call

```
List<String> foo = Arrays.asList("cow", "horse", "pig");
```

creates an object of type `List<String>` and populates it with the listed elements. Again, we see a variable number of items in the list. Varargs can be of any primitive or object type.

Now let us discuss writing methods with varargs in them. We begin by showing an example.

```
public class VarargsExample
{
    public static void main(String[] args)
    {
        System.out.println(product(3,4,7,8));
    }
    public static int product(int... numbers)
    {
        int out = 1;
        for(int k: numbers)
        {
            out *= k;
        }
        return out;
    }
}
```

Here are some things to note. Run this; it prints out 672, the product of the four numbers passed `product`.

1. The parameter `numbers` is treated as if were an array.
2. You can access the entries of the array using `[]` and you can use the collections `for` loop on the array.
3. This construct can be used on both object and primitive types.

There is a catch. You can only have one vararg in a method and it must occur at the end of your method's signature. Think about why this must be required! As another example, we can write a variant of `String`'s `join` method.

```
import java.util.Arrays;
public class VarargsExample
```

```
{
    public static void main(String[] args)
    {
        System.out.println(product(3,4,7,8));
        System.out.println(myJoin("|", "a", "b", "c", "d"));
    }
    public static int product(int... numbers)
    {
        int out = 1;
        for(int k: numbers)
        {
            out *= k;
        }
        return out;
    }
    // this is the same as mortar.join(bricks)
    public static String myJoin(String mortar, String... bricks)
    {
        int n = bricks.length;
        if(n == 0)
        {
            return "";
        }
        StringBuffer sb = new StringBuffer();
        for(int k = 0; k < n; k++)
        {
            sb.append(bricks[k] + mortar);
        }
        sb.append(bricks[n - 1]);
        return sb.toString();
    }
}
```

Run this and see `a|b|c|d` printed.

Finally, notice that you can pass an array to a varargs argument and it will behave as expected. Let us demonstrate this on `product`

```
import java.util.Arrays;
public class VarargsExample
{
    public static void main(String[] args)
    {
        System.out.println(product(3,4,7,8));
        System.out.println(myJoin("|", "a", "b", "c", "d"));
        int [] nums = {6,7,8,9};
```

```
        System.out.printf("product(%s) = %s\n",
            Arrays.toString(nums), product(nums));
    }
    public static int product(int... numbers)
    {
        int out = 1;
        for(int k: numbers)
        {
            out *= k;
        }
        return out;
    }
    public static String myJoin(String mortar, String... bricks)
    {
        int n = bricks.length;
        if(n == 0)
        {
            return "";
        }
        StringBuffer sb = new StringBuffer();
        for(int k = 0; k < n; k++)
        {
            sb.append(bricks[k] + mortar);
        }
        sb.append(bricks[n - 1]);
        return sb.toString();
    }
}
```

Can I require a minimum number of arguments in a vararg function? With a little trickery, yes. Suppose you want to make a function that requires at least two integers, and which returns an array containing the products of adjacent neighbors. You can do this.

```
public int[] multipliedNeighbors(int a, int b, int... rest)
{
    int[] out = new int[rest.length + 1]
    out[0] = a*b;
    if(rest.length == 0)
        return out;
    out[1] = b*rest[0]
    for(int k = 2; k < rest.length; k++}
    out[k] = rest[k-1]*rest[k]);
    return out;
}
```

Programming Exercises

1. Write a varargs function that accepts strings and which returns the first string in the list alphabetically, ignoring case.
2. Write a varargs function that accepts doubles and which returns the mean of the numbers in the list.
3. Write a varargs function `subset(String[] foo, int... indices)` which returns an array of strings at the indicated indices. If an index is out of bounds, throw an `IndexOutOfBoundsException`. Example:

```
String[] foo = {"a", "b", "c", "d"}
subset(foo, 2,1,3,1) -> {"c", "b", "d", "b"}.
```

8.7 `static` and `final`

You have noticed that the `static` keyword appears sometimes in the API guide. In Java, `static` means "shared." Static portions of your class are shared by all instances of the class. They must, therefore, be independent of any instance of the class, or *instance-invariant*.

When you first instantiate a class in a program, the *Java class loader* first sets up housekeeping. It loads the byte code for the class into RAM.

Before the constructor is called, any static items go in a special part of memory that is visible to all instances of the class. Think of this portion of memory as being a bulletin board visible to all instances of the class. You may make static items public or private, as you see fit. Static items that are public are visible outside and inside of the class.

When a variable or method is static, it can, and should, be called by the class's name. For instance, `BigInteger.valueOf()` is a static method that converts any `long` into a `BigInteger`. Recall we called this method a static factory method; it is static and behaves as a "factory" that accepts `longs` and converts the to `BigIntegers`. Static factory methods are often a superior alternative to a long list of constructors. See [2], pp. 5 ff.

Two other familiar examples are the `Math` and `Arrays` classes. In the `Math` class, recall you find a square-root by using `Math.sqrt()`, in `Arrays`, the static method `toString(T[])` creates a string representation of the array passed it. All of `Math`'s and `Arrays` methods are static. Neither has a public constructor. Both are called *convenience* or `service` classes that exist as containers for related methods.

You can also have variables that are declared static. In the `Math` library, there are `Math.PI` and `Math.E`. These variables are static. They are also `final`;

they are are variables that cannot be reassigned. In general, any variable you mark `final` cannot be reassigned after it is initialized.

Constness vs. `final` Variables anywhere in Java can be marked `final`; this means you cannot reassign the variable once it is initialized. However, you can call mutator methods on that datum and change the state of the object a `final` variable points to. In this case you do not have constness; the object being pointed at by a `final` variable can have its state changed.

Since `Math.PI` and `Math.E` are primitives, they are, in fact, constants.Since immutable objects and primitives lack mutator methods, these are rendered constant by declaring them `final`.

If you create static variables, you should also have a `static` block in your class. Code inside this block is run when the class is first loaded. Use it to intialize static data members. In fact, it is a desirable postcondition of your `static` block running that all static state variables are explicitly initialized. Remember *"Explicit is better than implicit"*, quoth the Zen of Python. Now let us put `final` and `static` to work.

The `Minter` class shown here gives each new instance an ID number, starting with 1. The static variable `nextID` acts as a "well" from which ID numbers are drawn. The `IDNumber` instance variable is marked `final`, so the `ID` number cannot be changed throughout any given `Minter`'s lifetime.

```
public class Minter
{
    private static int nextID;
    final private int ID;
    static
    {
        nextID = 1;
    }
    public Minter()
    {
        ID = nextID;
        nextID++;
    }
    public String toString()
    {
        return "Minter, ID = " + ID;
    }
}
```

8.7.1 Etiquette for Static Members

Since the Java class loader creates the static data for a class before any instance of the class is created, there is a separation between static and non–static portions of a class.

Non–static methods and state variables may access static portions of a class. This works because the static portion of the class is created before any instance of the class is created, so everything needed is in place. Outside of your class, other classes may see and use the static portions of your class that are marked `public`. These client programmers do not need to instantiate your class. They can gain access to any static class member, be it a method or a state variable by using the

```
ClassName.staticMember
```

construct.

Now consider the reverse case. Things in a class that are static must be instance-invariant. This means you cannot access the state variables or non-static methods of an object directly from a static method.

However, you can create an instance of your class and call non-static methods on the instance. What you cannot do is have *direct* access to non-static data or methods in a class.

The key to understanding why is to know that *static data is shared by all instances of the class*. Hence, to be well-defined, *static data must be instance–invariant*. Since your class methods can, and more often than not, do depend on the state variables in your class, they in general are not instance–invariant. Static methods and variables belong to the class as a whole, not any one instance. This restriction will be enforced by the compiler. Even if a method does not depend upon a class's state, unless you declare it `static`, it is not static and static methods may not call it.

To use any class method in the non-static portion of your class, you must first instantiate the class and call the methods via that instance. We will see an example this at work in the following subsection

For a simple example, place this method in the `Minter` class we just studied.

```
public static void main(String[] args)
{
    Minter m = new Minter();
    System.out.println(m);
}
```

Run the class and you will see it is now executable.

```
$ java Minter
Minter, IDNumber = 1
```

Observe that we made a tacit call to a method of the class `Minter`. To use the class, we had to create an instance `m` of `Minter` first. When we called `System.out.println`, we made a tacit call to `m.toString()`. You cannot make naked (no-instance) methods calls to non-static methods in `main`. You can, however, see the private parts of instances of the class.

Really, it is best to think of `main` as being outside the class and just use instance of your or other classes and use their (public) interface.

Finally, notice that `main` has an argument list with one argument, `String[] args`. This argument is an array of `String`s. This is how command–line arguments are implemented in Java. We now show a class that demonstrates this feature.

```
public class CommandLineDemo
{
    public static void main(String[] args)
    {
        int num = args.length;
        System.out.println("You entered " + num + " arguments.");
        int count = 0;

        for (String k: args)
        {
            System.out.println("args[" + count + "] = " + k);
            count ++;
        }
    }
}
```

Now we shall run our program with some command–line arguments. You need to type in the `java` command yourself rather than just hitting F2.

```
> java CommandLineDemo one two three
You entered 3 arguments.
args[0] = one
args[1] = two
args[2] = three
>
```

Even if you do not intend for your class to be executable, the `main` method is an excellent place for putting test code for your class. Making your class executable can save typing into the interactions pane. It is also necessary if you ever want to distribute your application in an *Java archive*, which is an executable file.

Chapter 9

BigFraction

9.0 Case Study: An Extended-Precision Fraction Class

We have achieved several goals so far, the most important of which are understanding what make up a Java and Python classes and understanding the core both languages so as to be Turing-complete.

To tie everything together, we will do a case study of creating two classes class `BigFraction.java` and `BigFraction.py`, which will impleement extended-precison rational arithmetic in both languagess. This class will have a professional appearance and will have full documentation.

9.0.1 A Brief Orientation

Before we begin let us remind ourselves of some basic mathematical facts and provide a rationale for what we are about to do. We are all familiar with the *natural* (counting) numbers

$$\mathbb{N} = \{1, 2, 3, 4, ...\}.$$

We can also start counting at zero because we are C family language geeks with

$$\mathbb{N}_0 = \{0, 1, 2, 3, ...\}.$$

The set of all integers (with signs) is often denoted by $\mathbb{Z}$. Why the letter Z? This comes from the German word *zahlen*, meaning "to count."

The `BigInteger` class in Java and the `int` type in Python create computational environments for computing in $\mathbb{Z}$ without danger of overflow, unless you really go bananas.

The *rational numbers* consist of all numbers that can be represented as a ratio of integers; the symbol used for them is $\mathbb{Q}$. The 'Q' is for "quotient." So,

$$\mathbb{Q} = \{m/n : m \in \mathbb{Z}, n \in \mathbb{N}, n \neq 0\}.$$

The `BigFraction` classes will create an environment for computing in $\mathbb{Q}$ similar to that which `BigInteger` provides for $\mathbb{Z}$.

9.1 Starting `BigFraction.py`

We begin by creating a class `BigFraction` in a file `BigFraction.py`.

```
class BigFraction:
    pass
```

What does a fraction need to know? It needs to know its numerator and denominator.

```
class BigFraction:
    def __init__(self, num, denom):
        this.num = num
        this.denom = denom
```

The other thing we should do is to give our class a string representatin.

```
class BigFraction:
    def __init__(self, num, denom):
        this.num = num
        this.denom = denom
    def __str__(self):
        return f"{self.num}/{self.denom}"
    def __repr__(self):
        return f"BigFraction({self.num}, {self.denom})"
```

Now let's test our new class in the interactive shell.

```
>>> from BigFraction import BigFraction
>>> b = BigFraction(1,2)
>>> b
BigFraction(1, 2)
>>> print(b)
1/2
>>> b = BigFraction(5,10)
```

```
>>> b
BigFraction(5, 10)
>>> print(b)
5/10
>>> b = BigFraction(-2,-4)
>>> print(b)
-2/-4
```

Uh oh. We can foresee problems here. Fractions should be stored in a fully reduced form. And, let's get the negative upstairs; this will turn out to be very beneficial down the road. So, we will put our fractions in a canonical form: fully reduced and any negative in the numerator.

Also, let us raise an error if the client programmer attempts to create a fraction with a zero denomintor.

9.1.1 Reducing Fractions

Ah, we are now back in the Miss Wormwood days of elementary school. She showed you a fraction such as

$$\frac{32}{12}$$

and you are supposed to reduce it. Well, both the top and bottom are even, so

$$\frac{32}{12} = \frac{16}{6}.$$

Hey we can do that again, and the result is

$$\frac{16}{6} = \frac{8}{3}.$$

Since 8 and 3 have no common factors, we are done.

This, however, is not going to cut the computational mustard. What if you have a fraction with 1000 digits on the top and bottom? Are you going to hunt for the common factors one by one and keep reducing? That would be a joyless slog to code as well as being baronially wasteful. We need a better way.

Suppose that a and b are integers that are not both zero. We define the *greatest common divisor* of a and b to be the largest positive integer dividing both evenly. Such a thing exsts, because 1 is a divisor of every integer. We will denote this function by $\gcd(a, b)$. The two quantities $a/\gcd(a, b)$ and $b/\gcd(a, b)$ are both integers. Also, they have no other common factor than 1. Now you see the *raison d'etre* for our interest in the greatest common divisor. If we have a fraction and we compute the greatest common divisor of its numerator and denominator, the result is a fully-reduced fraction.

Being avid Pythonista, we might take this approach. Suppose we have a and b. We could start at 2 and see if 2 is a common divisor of a and b. If it is, we can keep track of that fact in a variable. We then go to 3 and repeat this procedure. We stop when we get to the smaller of a and b. This is a pretty hackish approach, but let's give it a whirl.

Programming Exercise Implement this function in Python. For what size of numbers does this really begin to bog down?

Where's the catch? Again, consider the case of a couple of integers, each having hundreds of digits. This method could take an eternity. It's time for a little math! We are going to state and prove a simple little theorem that is the key to a fast gcd calculation.

First, let's talk a little bit about divisibility. We will use the notation $a \mid b$ to indicate that a divides b evenly; equivalently $b\%a = 0$. This means that there is some integer q so that $aq = b$.

Suppose that d is a common divisor of a and b You can choose integers s and t so that $ds = a$ and $dt = b$. Now suppose x and y are any integers. Then

$$ax + by = ads + bty = d(as + by);$$

since $as + by$ is an integer, we have $d|as + by$. The denouement: Any common divisor of a and b will also be a divisor of $ax + by$ for any integers x and y.

Let's call such things as $ax + by$ *integer combinations* of a and b. What we have show is that if d is a common divisor of a and b, it is a divisor of any integer combination of a and b.

Theorem. *Suppose that a, b, q, and r are integers and that $b = aq + r$. Then* $\gcd(b, a) = \gcd(a, r)$.

Proof. Suppose that d is a common divisor of a and r. Since $b = aq + r$, we have represented b as an integer combination of a and r. Therefore $d \mid b$. We conclude that d is a common divisor of a and b. We have just shown that every common divisor of a and r is a common divisor of a and b.

Now suppose d is common divisor of a and b. Since $b = aq + r$, $r = b - aq$. The integer r is an integer combination of a and b; therefore $d \mid r$. We have just shown that d is a common divisor of a and r. We have shown that every common divisor of a and b is a common divisor of a and r.

This tells us that a and b have exactly the same common divisors as a and r. We conclude $\gcd(b, a) = \gcd(a, r)$.

9.1.2 Speeding things up

One thing we know is that for any integer b and and non-zero integer a,

$$b = a * (b//a) + b\%a.$$

Here is another useful fact, $\gcd(a, 0) = |a|$, provided that $a \neq 0$. Let's compute $\gcd(1048576, 7776)$. We will use Python as our calculator. You should try this on another pair of big numbers.

```
>>> a = 7776
>>> b = 1048576
>>> remainder = 1048576 % 7776
>>> remainder
6592
>>> b, a = a, remainder
>>> b
7776
>>> a
6592
>>> remainder = b%a
>>> b, a = a, remainder
>>> a
1184
>>> b
6592
>>> remainder = b%a
>>> b, a = a, remainder
>>> b
1184
>>> a
672
>>> remainder = b%a
>>> b, a = a, remainder
>>> b
672
>>> a
512
>>> remainder = b%a
>>> b, a = a, remainder
>>> b
512
>>> a
160
>>> remainder = b%a
>>> b, a = a, remainder
```

```
>>> b
160
>>> a
32
>>> remainder = b%a
>>> b, a = a, remainder
>>> a
0
>>> b
32
```

Thhis gives us a chain of equalities.

$$\gcd(1048576, 7776) = \gcd(7776, 6592) = \cdots = \gcd(32, 0).$$

Since $\gcd(32, 0) = 32$, we are done. It seems we have a loop here

```
while a > 0:
        b, a = a, b%a
```

Changing signs does not change the gcd; to wit $\gcd(\pm a, \pm b) = \gcd(a, b)$, so we can strip off any negative signs. Also, let's raise an error if some reckless client tries to compute $\gcd(0, 0)$. Now for the coup d'grace.

```
def gcd(a,b):
        if a == 0 and b ==  0:
                raise ValueError
        if a < 0:
                a = -a
        if b < 0:
                b = -b
        while a > 0:
                b, a = a, b%a
        return b
```

Place this in a file named `number_theory.py`. Import it and test it. Ooh, it's quick.

```
>>> from number_theory import gcd
>>> gcd(1048576, 7776)
32
>>> gcd(323980490348, 32980398423123456)
4
```

This algorithm is called *Euclid's Algorithm*. The slowest it can work is the case of two adjacent fibonacci numbers. Since these grow exponentially, the number of iterations is at worst proportional to $\log(n)$, where n is the larger of the two numbers.

9.1.3 Finishing __init__

Now let us write this method. We will take the addional step of kicking any negative upstars. We will rase an error if a zero denominator gets passed in.

```
def __init__(self, num, denom)
    if denom == 0:
        raise ValueError
    if denom < 0:
        num = -num
        denom = -denom
    d = gcd(num, denom)
    self.num = num//d
    self.denom = denom//d
```

This method has the desirable property of storing a fraction in the canonical form we specified. The fraction is stored fully reduced. Any negative is in the denominator. You will see that this design decision will pay dividendds down the road. The care we took here will benefit us very soon. Let us lay out the whole class that we have so far.

```
from number_theory import gcd
class BigFraction:
    def __init__(self, num, denom):
        if denom == 0:
            raise ValueError
        if denom < 0:
            num = -num
            denom = -denom
        d = gcd(num, denom)
        self.num = num//d
        self.denom = denom//d
    def __str__(self):
        return f"{self.num}/{self.denom}"
    def __repr__(self):
        return f"BigFraction({self.num}, {self.denom})"
```

```
>>> from BigFraction import BigFraction
>>> b = BigFraction(7776, 1048576)
>>> b
BigFraction(243, 32768)
>>> print(b)
243/32768
```

9.2 Starting BigFraction.java

In this section, we will build a big fraction class with the same capabilities as the Python class. Note that we will be using `BigInteger`s as numerator and denominator. So, let's get started. We will rough in some items.

```
import java.math.BigInteger;
public class BigFraction
{
    private BigInteger num;
    private BigInteger denom;
    public BigFraction(BigInteger num, BigInteger denom)
    {
    }
    public String toString()
    {
        return String.format("%s/%s", num, denom);
    }
}
```

Looking at the `BigInteger` docs, we notice several things. One is that `BigIntegerss` are immutable. We can ensure this by making our state variables `final`.

```
import java.math.BigInteger;
public class BigFraction
{
    private final BigInteger num;
    private final BigInteger denom;
    public BigFraction(BigInteger num, BigInteger denom)
    {
    }
    public String toString()
    {
        return String.format("%s/%s", num, denom);
    }
}
```

Another is that `BigInteger` has a `gcd` method. We can avail ourselves of that. We also have to do the correct things to negate a `BigInteger` and to checks its positivity or negativity.

Changing sign is easy; just use the `negate()` method. To check sign, it is handy to compare with the static constant `BigInteger.ZERO`. Dividing is done with the `divide` method.

Let us go to work on the constructor. We show the lines of Python and use them as a guide, translating into Java as we progress.

```
    public BigFraction(BigInteger num, BigInteger denom)
    {
        //if denom == 0:
            //raise ValueError
        if(denom.equals(BigInteger.ZERO))
        {
            throw new IllegalArgumentException();
        }
        //if denom < 0:
            //num = -num
            //denom = -denom
        if(denom.compareTo(BigInteger.ZERO) < 0)
        {
            num = num.negate();
            denom = denom.negate();
        }
        //d = gcd(num, denom)
        BigInteger d = num.gcd(denom);
        //self.num = num//d
        //self.denom = denom//d
        this.num = num.divide(d);
        this.denom = denom.divide(d);
    }
```

Observe that we can only initialize state once because the state variables are `final`. We worked with the constructor's parameters as local variables until the very end.

Now let's assemble our effort.

```
import java.math.BigInteger;
public class BigFraction
{
    private final BigInteger num;
    private final BigInteger denom;
    public BigFraction(BigInteger num, BigInteger denom)
    {
        if(denom.equals(BigInteger.ZERO))
        {
            throw new IllegalArgumentException();
        }
        if(denom.compareTo(BigInteger.ZERO) < 0)
        {
            num = num.negate();
            denom = denom.negate();
        }
```

```
        BigInteger d = num.gcd(denom);
        this.num = num.divide(d);
        this.denom = denom.divide(d);
    }
    @Override
    public String toString()
    {
        return String.format("%s/%s", num, denom);
    }
}
```

We are thinking that there might be a constructor that will take a integer and convert it into a `BigInteger`, but there isn't. However, if we look in the docs, we wil see this.

`public static BigInteger valueOf(long val)`
Returns a BigInteger whose value is equal to that of the specified long.
API Note:
This static factory method is provided in preference to a (long) constructor because it allows for reuse of frequently used BigIntegers.
Parameters:
`val` - value of the BigInteger to return.
Returns:
a BigInteger with the specified value.

We will crib from this strategy and make our own static factory method `BigFraction.valueOf(long num, long denom)`. This will save us work down the road.

```
import java.math.BigInteger;
public class BigFraction
{
    private final BigInteger num;
    private final BigInteger denom;
    public BigFraction(BigInteger num, BigInteger denom)
    {
        if(denom.equals(BigInteger.ZERO))
        {
            throw new IllegalArgumentException();
        }
        if(denom.compareTo(BigInteger.ZERO) < 0)
        {
            num = num.negate();
            denom = denom.negate();
        }
        BigInteger d = num.gcd(denom);
```

```
        this.num = num.divide(d);
        this.denom = denom.divide(d);
    }
    @Override
    public String toString()
    {
        return String.format("%s/%s", num, denom);
    }
    public static BigFraction valueOf(long num, long denom)
    {
        return new BigFraction(BigInteger.valueOf(num),
            BigInteger.valueOf(denom));
    }
}
```

Et voila! It works!

```
jshell> /open BigFraction.java

jshell> BigFraction b = BigFraction.valueOf(7776, 1048576);
b ==> 243/32768
```

Our two classes are at the same level of progress.

9.3 Look out Miss Wormwood! Arithmetic!

The aim of this section is to endoow our big fractions with the ability to do arithmetic. Let us focus on these five operations: +, -, *, / and exponentiation.

9.3.1 Addition

Recall that

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}.$$

Let's start on the Python side. To redefine addition, there is the dunder method `__add__`. So in our class we make the header as follows

```
    def __add__(self, other):
```

Both `self` and `other` will be `BigFractions` so each has a numerator and a denominator. We use the addition formula for fractions and compute the numerator for the sum as follows.

```
    top = self.num*other.denom + self.denom*other.num
```

Now let's get the denominator

```
    bottom = self.denom*other.denom
```

Now let's put the whole thing together and ship out a `BigFraction`.

```
    def __add__(self, other):
        top = self.num*other.denom + self.denom*other.num
        bottom = self.denom*other.denom
        return BigFraction(top, bottom)
```

Now we test our work.

```
>>> from BigFraction import BigFraction
>>> a = BigFraction(1,3)
>>> b = BigFraction(1,2)
>>> a + b
BigFraction(5, 6)
>>> print(a + b)
5/6
```

Now it's Java's turn. You cannot redefine operators in Java, so we will do as they did in `BigInteger` and create an `add` method. The lines of Python are show here.

```
    public BigFraction add(BigFraction that)
    {

        //top = self.num*other.denom + self.denom*other.num
        //bottom = self.denom*other.denom
        //return BigFraction(top, bottom)
    }
```

We now translate them.

```
    public BigFraction add(BigFraction that)
    {
        //top = self.num*other.denom + self.denom*other.num
        BigInteger top = num.multiply(that.denom).add(
            denom.multiply(that.num));
        //bottom = self.denom*other.denom
        BigInteger bottom = denom.multiply(that.denom);
```

```
        //return BigFraction(top, bottom)
        return new BigFraction(top, bottom);
    }
```

Now test it. You can delete the python comments from your `add` method.

```
jshell> /open BigFraction.java

jshell> BigFraction a = BigFraction.valueOf(1,2);
a ==> 1/2

jshell> BigFraction b = BigFraction.valueOf(1,3);
b ==> 1/3

jshell> a.add(b)
$5 ==> 5/6
```

9.3.2 Subtraction

This is fish in a barrel since

$$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}.$$

We just change the addition in the middle to a subtraction. Here is Python

```
    def __sub__(self, other):
        top = self.num*other.denom - self.denom*other.num
        bottom = self.denom*other.denom
        return BigFraction(top, bottom)
```

And here is Java.

```
    public BigFraction subtract(BigFraction that)
    {

        BigInteger top = num.multiply(that.denom).subtract(
            denom.multiply(that.num));
        BigInteger bottom = denom.multiply(that.denom);
        return new BigFraction(top, bottom);
    }
```

9.3.3 Multiplication

We all know that

$$\frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd}.$$

We therefore proceed as follows in Python.

```
    def __mul__(self, other):
        top = self.num*other.num
        bottom = self.denom*other.denom
        return BigFraction(top, bottom)
```

Now for Java.

```
    public BigFraction multiply(BigFraction that)
    {
        BigInteger top = num.multiply(that.num);
        BigInteger bottom = denom.multiply(that.denom);
        return new BigFraction(top, bottom);
    }
```

9.3.4 Division

This is just "invert and multiply."

```
    def __truediv__(self, other):
        top = self.num*other.denom
        bottom = self.denom*other.num
        return BigFraction(top, bottom)
```

```
    public BigFraction divide(BigFraction that)
    {

        BigInteger top = num.multiply(that.denom);
        BigInteger bottom = denom.multiply(that.num);
        return new BigFraction(top, bottom);
    }
```

Let's test-drive the Python.

```
>>> b = BigFraction(1,4);
>>> c = BigFraction(1,5);
>>> b + c
BigFraction(9, 20)
>>> b - c
BigFraction(1, 20)
>>> b*c
BigFraction(1, 20)
>>> b/c
```

```
BigFraction(5, 4)
>>>
```

And now the Java.

```
jshell> /open BigFraction.java

jshell> BigFraction b = BigFraction.valueOf(1,4);
b ==> 1/4

jshell> BigFraction c = BigFraction.valueOf(1,5);
c ==> 1/5

jshell> b.add(c)
$5 ==> 9/20

jshell> b.subtract(c)
$6 ==> 1/20

jshell> b.divide(c)
$7 ==> 5/4

jshell> b.multiply(c)
$8 ==> 1/20
```

9.3.5 Pow!

Holy exponent, Batman! It's time to compute powers! Note the argument is an integer.

```
    def __pow__(self, n):
        if n == 0:
            return BigFraction(1,1)
        if n > 0:
            return BigFraction(self.num**n, self.denom**n)
        n = -n
            return BigFraction(self.denom**n, self.num**n)


    public BigFraction pow(int n)
    {
        if n == 0:
        {
            return BigFraction.valueOf(1,1);
        }
```

```
        if n > 0:
        {
            return BigFraction(num.pow(n), denom.pow(n));
        }
        n = -n;
        return BigFraction(denom.pow(n), num.pow(n));
    }
```

Programming Exercises Add these methods to our existing `BigFraction` class. These will make our `BigFraction`s more resemble `BigIntegers`.

1. Write a method `public BigInteger bigIntValue()` that divides the denominator into the numerator and which truncates towards zero.
2. Write the method `public BigFraction abs()` which returns the absolute value of this `BigFraction`.
3. Write a method `public BigFraction negate()` which returns a copy of this `BigFraction` with its sign changed.
4. Write the method `public BigFraction max(BigFraction)` which returns the larger of this `BigFraction` and `that`.
5. Write the method `public BigFraction min(BigFraction)` which returns the smaller of this `BigFraction` and `that`.
6. Write the method `public int signum()` which returns +1 if this `BigFraction` is positive, -1 if it is negative and 0 if it is zero.
7. Write the method `public int compareTo(BigFraction that)` which returns +1 if this `BigFraction` is larger than `that`, -1 if `that` is larger than this `BigFraction` and 0 if this `BigFraction` equals `that`.
8. Add a static method public BigFraction harmonic(int n) which computes the nth harmonic number. Throw an `IllegalArgumentException` if the client passes an `n` that is negative.
9. When should division throw an `IllegalArgumentException`? Add this feature to the class.
10. (Quite Challenging) Write the method `public double doubleValue()` which returns a floating point value for this `BigFraction`. It should return `Double.NEGATIVE_INFINITY` or `Double.POSITIVE_INFINITY` where appropriate. Test this very carefully; it is not easy to get it right.

9.4 Adding Static Constants

In Java, we have a notion of constness; this consists of a `final` variable pointing at an immutable object. Let's give our `BigFraction` class static constants `ZERO`,

`ONE`, and `HALF`. We will create them at the top of the class and initialize them in a `static` block.

To do this just insert this right at the top of your class

```
public static final BigFraction ZERO;
public static final BigFraction HALF;
public static final BigFraction ONE;
static
{
    ZERO = BigFraction.valueOf(0,1);
    HALF = BigFraction.valueOf(1,2);
    ONE = BigFraction.valueOf(1,1);
}
```

9.5 Documenting Your Code

These two classes could be quite useful to others. Now we need to give our users documentation so they can learn how to use our classes effectively.

We will document our Java code using the `javadoc` system. This system is easy to use and it creates a professional-looking API page that is similar in a appearance to the ones you see for the standard libraries.

Javadoc comments are delimited by the starting token `/**` and the ending token `*/`. C/C++ style comments delimited by `//` and `/* ..... */` do not appear on Javadoc pages. You may use HTML markup in your javadoc where needed.

Use Javadoc to document your *interface*, the public portion of your class. Do not javadoc `private` methods or state variables. We will produce a full javadoc page for our `BigFraction` class.

We will use docstings to document our Python class. These will be displayed in response to the `help` command or with the command

```
unix> python -m pydoc YourModule.py
```

9.5.1 Documenting `BigFraction.py`

Begin by describing the class right after the class header like so.

```
from number_theory import gcd
class BigFraction:
    """This is a class for performing extended-precision rational
```

```
arithmetic.  It includes a full suite of operators for arithmetic
and it produces a sortable objects, ordered by their numerical
values.

All BigFractions are stored in a canonical form: they are fully
reduced and any negative is stored in the numerator.
"""
```

Now for the `__init__` method.

```
    def __init__(self, num, denom):
        """This accepts two integer argments, a numerator
and a denominator.  A zero denominator will trigger a ValueError."""
        if denom == 0:
            raise ValueError
        if denom < 0:
            num = -num
            denom = -denom
        d = gcd(num, denom)
        self.num = num//d
        self.denom = denom//d
```

If you type the command `python -m pydoc BigFraction`, in a command window you will see this. Users can also see this documentaton in an interactive session using the `help` command.

```
NAME
    BigFraction

CLASSES
    builtins.object
        BigFraction

    class BigFraction(builtins.object)
     |  BigFraction(num, denom)
     |
     |  This is a class for performing extended-precision rational
     |  arithmetic.  It includes a full suite of operators for arithmetic
     |  and it produces a sortable objects, ordered by their numerical
     |  values.
     |
     |  All BigFractions are stored in a canonical form: they are fully
     |  reduced and any negative is stored in the numerator.
     |
     |  Methods defined here:
```

```
 |
 |  __add__(self, other)
 |
 |  __init__(self, num, denom)
 |      This accepts two integer argments, a numerator
 |      and a denominator.  A zero denominator will trigger a ValueError.
```

To exit, type q. Now we document the rest of the class

```
from number_theory import gcd
class BigFraction:
    """This is a class for performing extended-precision rational
arithmetic.  It includes a full suite of operators for arithmetic
and it produces a sortable objects, ordered by their numerical
values.

All BigFractions are stored in a canonical form: they are fully
reduced and any negative is stored in the numerator.

This class has three static constants

ZERO, the BigFraction representing 0
ONE, the BigFraction representing 1
HALF, the BigFraction representing 1/2
"""
    ZERO = None
    ONE = None
    HALF = None
    def __init__(self, num, denom):
        """This accepts two integer argments, a numerator
and a denominator.  A zero denominator will trigger a ValueError."""
        if denom == 0:
            raise ValueError
        if denom < 0:
            num = -num
            denom = -denom
        d = gcd(num, denom)
        self.num = num//d
        self.denom = denom//d
    def __str__(self):
        """This returns a string represenation for this
        BigFraction of the form numerator/denominator."""
        return f"{self.num}/{self.denom}"
    def __repr__(self):
        """This returns a string representation of the form
        BigFraction(numerator, denominator) suitable for the Python
```

```
        REPL"""
        return f"BigFraction({self.num}, {self.denom})"
    def __add__(self, other):
        """This defines + and  returns the sum of two BigFractions."""
        top = self.num*other.denom + self.denom*other.num
        bottom = self.denom*other.denom
        return BigFraction(top, bottom)
    def __sub__(self, other):
        """This defines - and  returns the difference of two BigFractions."""
        top = self.num*other.denom - self.denom*other.num
        bottom = self.denom*other.denom
        return BigFraction(top, bottom)
    def __mul__(self, other):
        """This defines * and  returns the product of two BigFractions."""
        top = self.num*other.num
        bottom = self.denom*other.denom
        return BigFraction(top, bottom)
    def __truediv__(self, other):
        """This defines / and  returns the quotient of two BigFractions."""
        top = self.num*other.denom
        bottom = self.denom*other.num
        return BigFraction(top, bottom)
    def __pow__(self, n):
        """This defines ** and returns this BigFraction raised to the
        nth power. This works for both positive and negative integers."""
        if n == 0:
            return BigFraction(1,1)
        if n > 0:
            return BigFraction(self.num**n, self.denom**n)
        n = -n
        return BigFraction(self.denom**n, self.num**n)
    @staticmethod
    def init_static():
        """initializes the static constants ZERO, HALF, and ONE."""
        BigFraction.ZERO = BigFraction(0,1)
        """The Big Fraction 0/1"""
        BigFraction.ONE  = BigFraction(1,1)
        """The Big Fraction 1/1"""
        BigFraction.HALF = BigFraction(1,2)
        """The Big Fraction 1/2"""
BigFraction.init_static()
```

9.5.2 Documenting `BigFraction.java`

The kind of class we have created represents a real extension of the Java language that could be useful to others. Now we need to give our class an API page so it has a professional appearance and so it can easily be used by others.

Javadoc comments are delimited by the starting token `/**` and the ending token `*/`. C/C++ style comments delimited by `//` and `/* ..... */` do not appear on Javadoc pages.

You may use HTML markup in your javadoc where needed.

Use Javadoc to document your *interface*, the public portion of your class. Do not javadoc `private` methods or state variables.

We will produce a full javadoc page for our `BigFraction` class. Let us begin with the constructors.

```
 /**
 * This constructor stores a <code>BigFraction</code> in
 * reduced form, with any negative factor appearing in
 * the numerator.
 * @param num the numerator of this <code>BigFraction</code>
 * @param denom the denommator of this <code>BigFraction</code>
 * @throws IllegalArgumentException if a zero
 * denominator is passed in
 */
public BigFraction(BigInteger num, BigInteger denom)
{
    if(denom.equals(BigInteger.ZERO))
        throw new IllegalArgumentException();

    BigInteger d = num.gcd(denom);
    if(denom.compareTo(BigInteger.ZERO) < 0)
    {
        num = num.negate();
        denom = denom.negate();
    }
    this.num = num.divide(d);
    this.denom = denom.divide(d);
}
/**
 * This default constructor produces BigFraction 0/1.
 */
public BigFraction()
{
    this(BigInteger.ZERO,BigInteger.ONE);
}
```

We see the special markup `@param`; this is the description given for each parameter. The markup `@throws` warns the client that an exception can be thrown by a method. You should always tell exactly what triggers the throwing of an exception, as the penalty for an exception is program death.

9.5.3 Triggering Javadoc

First we give instructions for DrJava. Bring up the Preferences by hitting control-; or by selecting the Preferences item from the bottom of the Edit menu. Under Web browser put the path to your web browser. An example of a valid path is

```
/usr/lib/firefox/firefox.sh
```

If you use Windoze, your path should begin with `\tt C:\`. If you use a Mac, it will be in your Applications folder. You can browse for it by hitting the `...` button just to the right of the Web Browser text field.

The javadoc will be saved in a directory called `doc` that is created in same directory as your class's code. Allow the javadoc to be saved in that folder, or files will "spray" all over your directory and make a big mess.

You can also javadoc at the command line with

```
unix> javadoc -d someDirectory BigFraction.java
```

The javadoc output will be placed in the directory `someDirectory` that you specify. Make sure you use the `-d` option to avoid spraying. To see your objet d'art, select File Open... in your browser and then navigate to the file `index.html` in your `doc` directory and open it.

Note that yoiur program *must* compile before any javadoc will be generated.

I don't see my javadoc! Make sure you are using the javadoc comment tokens like so.

```
/**
*     stuff
*/
```

and not regular multiline comment token that look like this.

```
/*
*     stuff
*/
```

9.5.4 Documenting `toString()` and `equals()`

You will see a new markup device `@return` and `overrides` which tells you what these methods override. You will notice if you look in the javadoc you generated, that an `overrides` tag is already in the method detail.

```
/**
 * @return a string representing this BigFraction of  the form
 * numerator/denominator.
 */
@Override
public String toString()
{
    return "" + num + "/" + denom;
}
```

Note the use of the `@Override` construct just after our javadoc markup. This is called an *annotation*, and the compiler checks that you have used the right signature to actual override the method. If you don't it will be flagged as a compiler error. Always use this annotation if you are implementing the methods `public boolean equals(Object o)` or `public String toString()`.

Now we deal similarly with the `equals` method.

```
/**
 * @param o an Object we are comparing this BigFraction to
 * @return true iff this BigFraction and that are equal numerically.
 * A value of <tt>false</tt> will be returned if the Object o is not
 * a BigFraction.
 */
@Override
public boolean equals(Object o)
{
    if(! (o instanceof BigFraction))
        return false;
    BigFraction that = (BigFraction) o;
    return num.equals(that.num) && denom.equals(that.denom);
}
```

9.5.5 Putting in a Preamble and Documenting the Static Constants

We show where to preamble goes, after the imports and before the head for the class. Place a succinct description of your class here to let your clients know what it does.

```
import java.math.BigInteger
/**
 * This is a class of immutable arbitrary-precision
 * rational numbers.  BigFraction provides
 * extended-precision fractional arithmetic
 * operations, including + with the <code>add</code> method,
 * - with the <code>subtract</code>
 * method, * with the <code>multiply</code> method,
 * and / with the <code>divide</code> method.
 * It computes integer powers
 * of fractions using the <code>pow</code> method.
 */
public class BigFraction
{
    //code
}
```

Documenting the static constants is very straightforward.

```
    /**
     * This is the BigFraction constant 0, which is 0/1.
     */
    public static final BigFraction ZERO;
    /**
     * This is the BigFraction constant 1, which is 1/1.
     */
    public static final BigFraction ONE;
```

9.5.6 Documenting Arithmetic

Next we javadoc all of the arithmetic operations we have provided the client. Notice how we add an exception if the client attempts to divide by zero.

```
    /**
     * This add BigFractions.
     * @param that a BigFraction we are adding to this BigFraction
     * @return <code>this</code> + <code>that</code>
     */
    public BigFraction add(BigFraction that)
    {
        BigInteger term1 = num.multiply(that.denom);
        BigInteger term2 = denom.multiply(that.num);
        BigInteger bottom = denom.multiply(that.denom);
        return new BigFraction(term1.add(term2), bottom);
```

```
}
/**
 * This subtracts BigFractions.
 * @param that a BigFraction we are adding to this BigFraction
 * @return <code>this</code> - <code>that</code>
 */
public BigFraction subtract(BigFraction that)
{
    BigInteger term1 = num.multiply(that.denom);
    BigInteger term2 = denom.multiply(that.num);
    BigInteger bottom = denom.multiply(that.denom);
    return new BigFraction(term1.subtract(term2), bottom);
}
/**
 * This multiplies BigFractions.
 * @param that a BigFraction we are adding to this BigFraction
 * @return <code>this</code> * <code>that</code>
 */
public BigFraction multiply(BigFraction that)
{
    BigInteger top = num.multiply(that.num);
    BigInteger bottom = denom.multiply(that.denom);
    return new BigFraction(top, bottom);
}
/**
 * This divides BigFractions.
 * @param that a BigFraction we are adding to this BigFraction
 * @return <code>this</code>/<code>that</code>
 * @throws <code>IllegalArgumentException</code> if division by
 * 0 is attempted.
 */
public BigFraction divide(BigFraction that)
{
    if(that.equals(BigFraction.ZERO))
        throw new IllegalArgumentException();
    BigInteger top = num.multiply(that.denom);
    BigInteger bottom = denom.multiply(that.num);
    return new BigFraction(top, bottom);
}
 /**
 * This computes an integer power of BigFraction.
 * @param n an integer power
 * @return <code>this</code><sup><code>n</code></sup>
 */
public BigFraction pow(int n)
{
```

```
        if(n > 0)
            return new BigFraction(num.pow(n), denom.pow(n));
        if(n == 0)
            return new BigFraction(1,1);
        else
        {
            n = -n;   //strip sign
            return new BigFraction(denom.pow(n), num.pow(n));
        }
    }
```

Finally, we will take care of our two `valueOf` methods.

```
    /**
     * @param n a long we wish to promote to a BigFraction.
     * @return A BigFraction object wrapping n
     */
    public static BigFraction valueOf(long n)
    {
        return new BigFraction(n, 1);
    }
    /**
     * @param num a BigInteger we wish to promote to a BigFraction.
     * @return A BigFraction object wrapping num
     */
    public static BigFraction valueOf(BigInteger num)
    {
        return new BigFraction(num, BigInteger.ONE);
    }
```

9.5.7 The Complete Code

Here it is! We have dropped in javadoc for our stateic factory method as well.

```
import java.math.BigInteger;
/**
 * This is a class of immutable arbitrary-precision
 * rational numbers.  BigFraction provides
 * extended-precision fractional arithmetic
 * operations, including + with the <code>add</code> method,
 * - with the <code>subtract</code>
 * method, * with the <code>multiply</code> method,
 * and / with the <code>divide</code> method.
 * It computes integer powers
 * of fractions using the <code>pow</code> method.
```

```
 */
public class BigFraction
{
    /**
     * This is the BigFraction constant 0, which is 0/1.
     */
    public static final BigFraction ZERO;
    /**
     * This is the BigFraction constant 1, which is 1/1.
     */
    public static final BigFraction ONE;

    static
    {
        ZERO = new BigFraction();
        ONE = new BigFraction(1,1);
    }
    private final BigInteger num;
    private final BigInteger denom;
    /**
     * This constructor stores a <code>BigFraction</code> in
     * reduced form, with any negative factor appearing in
     * the numerator.
     * @param num the numerator of the <code>BigFraction</code>
     * @param denom the denominator of the <code>BigFraction</code>
     * @throws <code>IllegalArgumentException</code> if the creation
     * of a zero-denominator <code>BigFraction</code> is attempted.
     */
    public BigFraction(BigInteger num, BigInteger denom)
    {
        if(denom.equals(BigInteger.ZERO))
            throw new IllegalArgumentException();

        BigInteger d = num.gcd(denom);
        if(denom.compareTo(BigInteger.ZERO) < 0)
        {
            num = num.negate();
            denom = denom.negate();
        }
        num = num.divide(d);
        denom = denom.divide(d);
    }
    /**
     * This default constructor produces BigFraction 0/1.
     */
    public BigFraction()
```

```
{
    this(BigInteger.ZERO,BigInteger.ONE);
}
/**
 * @return a string representing this BigFraction of  the form
 * numerator/denominator.
 */
@Override
public String toString()
{
    return String.format("%s/%s", num, denom);
}
/**
 * @param o an Object we are comparing this BigFraction to
 * @return true iff this BigFraction and that are equal numerically.
 * A value of <code>false</code> will be returned if the Object o is not
 * a BigFraction.
 */
@Override
public boolean equals(Object o)
{
    if(! (o instanceof BigFraction))
        return false;
    BigFraction that = (BigFraction) o;
    return num.equals(that.num) && denom.equals(that.denom);
}
/**
 * This static factory produces num/denom as a BigFraction.
 * @param num the numerator for this BigFraction
 * @param denom the denominator for this BigFraction
 * @return A <code>BigFraction</code> representing num/denom.
 */
public static BigFraction valueOf(long num, long denom)
{
    return new BigFraction(BigInteger.valueOf(num),
        BigInteger.valueOf(denom));
}
/**
 * This add BigFractions.
 * @param that a BigFraction we are adding to this BigFraction
 * @return <code>this</code> + <code>that</code>
 */
public BigFraction add(BigFraction that)
{
    BigInteger term1 = num.multiply(that.denom);
    BigInteger term2 = denom.multiply(that.num);
```

```
        BigInteger bottom = denom.multiply(that.denom);
        return new BigFraction(term1.add(term2), bottom);
    }
    /**
     * This subtracts BigFractions.
     * @param that a BigFraction we are adding to this BigFraction
     * @return <code>this</code> - <code>that</code>
     */
    public BigFraction subtract(BigFraction that)
    {
        BigInteger term1 = num.multiply(that.denom);
        BigInteger term2 = denom.multiply(that.num);
        BigInteger bottom = denom.multiply(that.denom);
        return new BigFraction(term1.subtract(term2), bottom);
    }
    /**
     * This multiplies BigFractions.
     * @param that a BigFraction we are adding to this BigFraction
     * @return <code>this</code> * <code>that</code>
     */
    public BigFraction multiply(BigFraction that)
    {
        BigInteger top = num.multiply(that.num);
        BigInteger bottom = denom.multiply(that.denom);
        return new BigFraction(top, bottom);
    }
    /**
     * This divides BigFractions.
     * @param that a BigFraction we are adding to this BigFraction
     * @return <code>this</code>/<code>that</code>
     * @throws <code>IllegalArgumentException</code> if division by
     * 0 is attempted.
     */
    public BigFraction divide(BigFraction that)
    {
        if(that.equals(BigFraction.ZERO))
            throw new IllegalArgumentException();
        BigInteger top = num.multiply(that.denom);
        BigInteger bottom = denom.multiply(that.num);
        return new BigFraction(top, bottom);
    }
    /**
     * @param n a long we wish to promote to a BigFraction.
     * @return A BigFraction object wrapping n
     */
    public static BigFraction valueOf(long n)
```

```
    {
        return new BigFraction(n, 1);
    }
     /**
     * @param num a BigInteger we wish to promote to a BigFraction.
     * @return A BigFraction object wrapping num
     */
    public static BigFraction valueOf(BigInteger num)
    {
        return new BigFraction(num, BigInteger.ONE);
    }
}
```

Programming Exercises

1. Add javadoc for all of the methods you wrote in the previous set of programming exercises.
2. Write a second class called `TestBigFraction`. Place a `main` method in this class and have it test `BigFraction` and its methods. Place the classes in the same directory.

Chapter 10

Types and Subtypes

10.0 Introduction

So far, we have been programming "in the small." We have created simple classes that carry out fairly straightforward chores. Our programs have been little one or two class programs. One class has been the class you are writing, the other has been `jshell` or a simple driver class with a `main` method in it. We created the `BigFraction` API, which allows a client programmer using it to do exact, extended-precision rational arithmetic.

So far, the relationship between classes has been a "has-a relationship." For example our `BigFraction` class has two `BigInteger`s, representing the numerator and denominator of our fraction object. We often use instances of classes that we attach to local variables inside of methods. This is a "uses-a" instead of a "has-a" relationship. Both of these relationships are `compositional`, since we are using them to compose, or build, our class. The compositional relationship is the most important and most common relationship between classes.

Java programs often consist of many classes, which work together to do a job. Sometimes we will create classes from scratch, sometimes we will aggregate various types of objects in a class, and sometimes we will customize existing classes using *inheritance*. We will also draw upon Java's vast class libraries. We will also see how to tie related classes together by using *interfaces*. These are offers to sign a contract whose terms are fulfilled by implementing the methods specified by the interface. Classes accepting this contract are said to *implement* the interface. What is interesting here is that you can create variables of interface type which can point at objects of any class that implements the interface.

Once this machinery is in place, we will look at functional interfaces, which allow us to create function-like objects. These objects are absolutely key to the

building of graphical user interface programs.

10.1 Interfaces

An interface in Java is an offer to sign a contract. This is best seen via an example. We shall create an interface for shapes; for our purposes, a shape is an object that can compute its diameter, perimeter, and area. Here is our interface. Notice the use of the new keyword `interface`. You can compile this and you will get a `.class` file.

```
public interface Shape
{
    public double area();
    public double perimeter();
    public double diameter();
}
```

For your class to fulfil the terms of the contract, it must implement all of the methods specified in the `Shape` interface. Let us make a `Rectangle` class that implements the three methods. Notice the "`implements Shape`" in the header of the class. This is how we accept the offer made by the interface.

```
public class Rectangle implements Shape
{
    private double width;
    private double height;
    public Rectangle(double width, double height)
    {
        this.width = width;
        this.height = height;
    }
    public Rectangle()
    {
        this(0,0);
    }
    public double area()
    {
        return height*width;
    }
    public double perimeter()
    {
        return 2*(height + width);
    }
    public double diameter()
```

```
    {
        return Math.hypot(height, width);
    }
}
```

Next, we create a Circle class. Both these classes implement `Shape`.

```
public class Circle immplements Shape
{
    private double radius;
    public Circle(double radius)
    {
        this.radius = radius;
    }
    public Circle()
    {
        this(0);
    }
    public double area()
    {
        return Math.PI*radius*radius;
    }
    public double perimeter()
    {
        return 2*Math.PI*radius;
    }
    public double diameter()
    {
        return 2*radius;
    }
}
```

10.1.1 Pretty Polymorphism

Interfaces, like classes, are types. Since `Circle` and `Rectangle` implement `Shape`, we say that `Circle` and `Rectangle` are *subtypes* of `Shape`, and that `Shape` is a *supertype* of `Circle` and `Rectangle`.

Recall that we have previously said that both varibles and objects have type, and that variables can only point at objects of their own type. Given the tools you have had, this is true. We now reveal a little more about type rules. Let us inspect our class and interface in `jshell` Note that since we are implementing `Shape` we must open it too in the session. This is just what you expect to see.

```
jshell> /open Shape.java
```

```
jshell> /open Rectangle.java

jshell> Rectangle r = new Rectangle(6,8)
r ==> Rectangle@685f4c2e

jshell> r.area()
$5 ==> 48.0

jshell> r.diameter()
$6 ==> 10.0

jshell> r.perimeter()
$7 ==> 28.0
```

Let us continue our session. Now we create a variable of type `Shape`. Since `Rectangle` is a subtype of `Shape`, a variable of type `Shape` can point at a `Rectangle`.

```
jshell> /open Circle.java

jshell> Shape s = new Rectangle(6,8);
s ==> Rectangle@2ef1e4fa

jshell> s.diameter()
$10 ==> 10.0

jshell> s.perimeter()
$11 ==> 28.0

jshell> s.area()
$12 ==> 48.0
```

This works for circles, too.

```
jshell> s = new Circle(10);
s ==> Circle@46f7f36a

jshell> s.diameter()
$14 ==> 20.0

jshell> s.perimeter()
$15 ==> 62.83185307179586

jshell> s.area()
$16 ==> 314.1592653589793
```

Here is what you can't do.

```
jshell> s = new Shape()
|  Error:
|  Shape is abstract; cannot be instantiated
|  s = new Shape()
|      ^---------^
```

Clearly this makes no sense because none of the methods specified in the interface has any code! This is a compile-time error. You cannot create instances of an interface. You can, however, create varibles of interface type. Said variables can point at any object whose type is a subtype of the interface's type. There are two principles at work here.

- **The Visibility Principle** The type of a variable pointing at an object determines what methods are visible; this is true for variables of class or interface type. Only methods in the variable's type may be seen. This is because Java is statically typed; visible methods must be known at compile time.
- **The Delegation Principle** If a variable is pointing at an object and a visible method is called, the object is responsible for executing the method. Regardless of a variable's type, if a given method in the object is visible, the object's method will be called. Remember objects are strongly aware of their type so you can do this.

Programming Exercises

1. Create a class `Triangle.java` that has as state variables the three sides of a triangle. To do this, you need *Herron's Formula*, which goes as follows. Suppose a triangle has sides of lengths a, b, and c. Then its `semiperimeter` s is defined as
$$s = \frac{a+b+c}{2}.$$
. The area of the triangle is
$$\sqrt{s(s-a)(s-b)(s-c)}$$
If the radicand is negative, you have an "illegal" triangle; in that case throw an `IllegalArgumentException`.
2. Make your class `Triangle`, implement `Shape`.
3. Extend `Triangle` to `EqualateralTriangle`, doing as little work as possible.
4. Make `Triangle` implement `Polygon`.

5. Declare and `ArrayList<Shape>` and add shapes to it. Implement this method in a new class's `main`.

```
public static double totalArea(ArrayList<Shape> al)
{
    return 0;
}
```

Create an array list of `Shapes` and test it out.

10.2 The API Guide

The API guide documents both classes and interfaces. Open the page for `ArrayList`. Near the top of the class, you will see this.

All Implemented Interfaces: `Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess`

Interfaces, like classes, can have type parameters. What you see listed are all of the standard library interfaces implemented by `ArrayList`.

Click on the link for `List<E>`. Near the top you will see this.

All Known Implementing Classes: `AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector`

This is a complete list of standard library classes implementing this interface. The Method Summary gives the methods required for a class to implement the interface. You can see that it is a rather large list of methods.

Now look up the interface `Comparable`. It specifies one method, `int compareTo(T o)`. You will see that there are swarms of classes that implement this interface, including `String` and `BigInteger`. A comparable type consists of sortable data. Implmenting this interfaces turns on some very valuable sorting tools built into Java.

10.3 Subclasses

You might ask, "Can one class be a subtype of another?" The answer is yes, and the purpose of this section is to show you how to do it. The mechanism we shall use is called *inheritance*. Again, we will illustrate this with an example. It would seem that a square is a special type of rectangle, so if we create a class `Square` it should be a subtype of `Rectangle`. The `extends` keyword indicates that `Square` is subtype of `Rectangle`. Automatically, `Square` gets all of `Rectangle`'s public methods. However, in this state we do not have an appropriate constructor.

```
public class Square extends Rectangle
{
}
```

Note that to compute arae, diameter, and perimeter of a square, we just need to know its side length. So we we begin by adding a state variable for that.

```
public class Square extends Rectangle
{
    private double side;
}
```

Now we need a constructor. Here we introduce the keyword `super`, which calls the parent constructor.

```
public class Square extends Rectangle
{
    private double side;
    public Square(double side)
    {
        super(side, side);
        this.side = side;
    }
}
```

Warning! If you use the `super` keyword in a constructor, it must be done in the first line of the constructor's code. Failure to do this is a compile time error. You should reverse the order of the two lines of code in `Square`'s constructor and see what this error looks like.

Now, It's time for inspection.

```
jshell> /open Shape.java

jshell> /open Rectangle.java

jshell> /open Square.java

jshell> Rectangle r = new Square(10);
r ==> Square@2e5d6d97

jshell> r.area()
$5 ==> 100.0

jshell> r.diameter()
```

```
$6 ==> 14.142135623730951

jshell> r.perimeter()
$7 ==> 40.0

jshell> Shape s = new Square(10)
s ==> Square@2ef1e4fa

jshell> s.area()
$9 ==> 100.0

jshell> s.diameter()
$10 ==> 14.142135623730951

jshell> s.perimeter()
$11 ==> 40.0
```

Notice that a subtype of a subtype is a subtype! Happily, we were able to inherit all of `Rectangle`'s methods and making `Square` was easy. Also, note that every type is a subtype of the root class `Object`. Here is a way to convince you.

```
jshell> Object o = "some string"
o ==> "some string"

jshell> Object p = new ArrayList<String>();
p ==> []

jshell> o.toString()
$3 ==> "some string"

jshell> p.toString()
$4 ==> "[]"
```

An `Object` has a `toString()` method, so by the Visibility Principle, an `Object` variable can see a `toString()` method. That is why everything works here.

Now see this.

```
jshell> o.length()
|  Error:
|  cannot find symbol
|    symbol:   method length()
|  o.length()
|  ^------^
```

Strings have a `length()` method, but `Objects` do not. The error you see is caused by the fact that an `Object` variable can only see `Object` methods;

hence this compile-time error. A similar thing happens when we try to get the `ArrayList`'s size.

```
jshell> p.size()
|  Error:
|  cannot find symbol
|    symbol:   method size()
|  p.size()
|  ^----^
```

10.4 Overriding Methods

Let us now put a `toString` method in our `Rectangle` and `Circle` classes. Insert these into your classes. As it stands now, the `Object toString()` method is just the class name followed by an and some hex digits. Add these to your classes.

```
public String toString()
{
    return String.format("Rectangle(%s, %s)", width, height);
}
```

```
public String toString()
{
    return String.format("Circle(%s)", radius);
}
```

```
jshell> /open Shape.java

jshell> /open Circle.java

jshell> /open Square.java

jshell> /open Rectangle.java

jshell> Shape r = new Rectangle(6,8);
r ==> Rectangle(6.0, 8.0)

jshell> Shape s = new Circle(10)
s ==> Circle(10.0)

jshell> Shape sq = new Square(10)
sq ==> Rectangle(10.0, 10.0)
```

We have successfully overriden the `Object` `toString()` method. When you do this, it is a smart idea to use the `@Override` annotation on these methods. Here is what it looks like on `Rectangle`.

```
@Override
public String toString()
{
    return String.format("Rectangle(%s, %s)", width, height);
}
```

and here it is cor `Circle`.

```
public String toString()
{
    return String.format("Circle(%s)", radius);
}
```

Use this annotation whenever overriding the method of an ancestor class. This causes the compiler to check to see if you are overriding properly. One thing that can happen is that you use the wrong signature in the method you are intending to override. In this case, you don't override the method, you *overload* it with method name overloading. This can cause unfortunate and hard to diagnose errors.

One thing we see that is unsatisfactory is the `Rectangle` `toString()` method is not really right thing for `Square`. We fix this by overriding the parent method. Add this to your `Square` class and you will like the result (Run it!).

```
@Override
public String toString()
{
    return String.format("Square(%s)", side);
}
```

By now you have likely surmised this. If you call a method on a object an that method is implemented in the object's class, that method is called. If the method is not present, then the JVM checks for it in the parent class and runs it if it is present there. This process will continue all the way up to the root class `Object`. If it's not found by then, the compiler will issue forth with an error message.

Programming Exercises

1. Make these classes, `Mama.java`

```
public class Mama
{
    public int foo()
    {
        return 42;
    }
}
```

and `Baby.java`

```
public class Baby extends Mama
{
    public int foo(int x)
    {
        return 2*x;
    }
}
```

2. Compile them.
3. Make this clas

```
public class Main
{
    public static void main(String[] args)
    {
        Mama m = new Mama();
        Baby b = new Baby();
        System.out.println(m.foo());
        System.out.println(b.foo());
        System.out.println(b.foo(10));
    }
}
\item Explain what happened.
\item Insert an \texttt{@Override} annotation on \texttt{foo} in
    the class \texttt{Baby}.  Compile. What happened?
```

10.5 Inheritance and the API Guide

Open the page for `ArrayList`. Right at the top of the page you will see this.

Module `java.base`

Package `java.util`

Class `ArrayList<E>`

```
java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.ArrayList<E>
```

Type Parameters: E - the type of elements in this list

All Implemented Interfaces:

```
Serializable, Cloneable, Iterable<E>, Collection<E>,
List<E>, RandomAccess
```

Direct Known Subclasses:

```
AttributeList, RoleList, RoleUnresolvedList
```

You can see the "family tree" of `ArrayList` displayed going all the way back up to `Object`. You can also see the child classes of `ArrayList` in the standard libraries.

Now scroll past the method summary. You will see all of the methods inherited from ancestor classes. The method names are links; click on them to see their method detail.

Exercises These will help you to learn to use the documentation more effectively.

1. From which class does `ArrayList` inherit its `toString()` method?
2. From which class does `ArrayList` inherit its `sort()` method?
3. From which class does `ArrayList` inherit its `equals()` method?
4. From which class does `ArrayList` inherit its `wait()` method?
5. Whose `add` methood is overriden by `ArrayList`?

10.6 Subinterfaces

Interfaces can be extended to require additional methods in implementing classes. For example, we could extend our `Shape` interface as follows.

```
public interface Polygon extends Shape
{
    public int numSides();
}
```

An implementing class for this interface must implement all shape methods and implement `numSides()` as well. Let us do this to `Rectangle`.

```
public class Rectangle implements Polygon
{
    private double width;
    private double height;
    public Rectangle(double width, double height)
    {
        this.width = width;
        this.height = height;
    }
    public Rectangle()
    {
        this(0,0);
    }
    public double area()
    {
        return height*width;
    }
    public double perimeter()
    {
        return 2*(height + width);
    }
    public double diameter()
    {
        return Math.hypot(height, width);
    }
    @Override
    public String toString()
    {
        return String.format("Rectangle(%s, %s)", width, height);
    }
    public int numSides()
    {
        return 4;
    }
}
```

Notice that there is no suitable `numSides()` method for a `Circle`, so we only

implement `Shape` in that case. Observe that `Square` implements `Polygon` automatically since implemented interfaces are inherited. There is no need to override here, since squares and rectangles both have four sides. Also note this; a variable of `Shape` type cannot see `numSides`. For this, you need a variable of type `Polygon`.

However, `Square` will inherit the implementation of `Polygon` from `Rectangle`.

Programming Exercises

1. What are the superinterfaces of `java.util.Collection<E>`?
2. Is `ArrayList<E>` a subtype of `java.util.Collection<E>`?
3. Of what interface(s) is `List<E>` a direct subinterface?

10.6.1 Default Methods

Beginning in Java8, methods called `default methods` were allowed to be placed into interfaces. Here is how the work.

If you implement an interface with a default method, you automatically "inherit" the default method. However, if you don't like what that method does, you are free to override it by implementing the method in your class.

If you implement two interfaces that have the same default method, you *must* override that method to avoid the deadly diamond problem. This convention is enforced by Mean Mr. Compiler.

10.7 Functional Interfaces

An interface is a *functional interface* if it specifies exactly one method. These interfaces can have default methods as exceptions to this rule but they are the only exceptions.

We will begin by studying the `Consumer` family of interfaces. This consists of four interfaces, all of which live in package `java.util.function`.

- `Consumer<T>` This is a generic interface and specifies a function taking an object of type `T` as input and having a void return type.
- `IntConsumer` This is interface and specifies a function taking an `int` as input and having a void return type.
- `LongConsumer` This is an interface that specifies a function taking an `Long` as input and having a void return type.

- `DoubleConsumer` This is an interface that specifies a function taking an `double` as input and having a void return type.

In all cases, the name of the method is `accept`. If you look in the API Guide, you will see the default method `default Consumer<T> andThen(Consumer<? super T> after`. Since this is a default method, we do not need to worry about or implement it now. Consumers take a piece of data and do something with it, such as printing it to `stdout`.

If you want to implement this, you might at first think, "Crud. I have to make a class that implements the desired consumer, put an `accept` method in it, and then create an instancd of of it to use it. Ugh. This is utterly useless, right?

Wrong. A fiendlishly clever mechanism of type inference makes using these interfaces simple. First, realize that you can create a variable of interface type. Secondly, Java has two means of creating function-like objects. First, let us introduce the *method reference*. Bear witness to this snippet of `jshell`.

```
jshell> Consumer<String> printMe = System.out::println;
printMe ==> $Lambda$19/0x0000000800b5a440@17f052a3

jshell>
```

What in tarnation? Why is this allowed? There is a friendly ghost here; it is a class with no name. We have created an object that is an instance of a nameless class that is a subtype of `Consumer<String>`. You now ask, "This is a pretty new toy, but what can you do with it.?"

It's time for a trip to the `ArrayList` class. Look at the method detail for the `forEach` method.

```
public void forEach(Consumer<?  super E> action)
```

Description copied from interface: `Iterable` Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception. Actions are performed in the order of iteration, if that order is specified. Exceptions thrown by the action are relayed to the caller. The behavior of this method is unspecified if the action performs side-effects that modify the underlying source of elements, unless an overriding class has specified a concurrent modification policy.

Specified by:

`forEach in interface Iterable<E>`

Parameters:

`action` - The action to be performed for each element

Throws:

`NullPointerException` - if the specified action is null

You see some terrifying notation here, this `<? super E>` thing. This just means that the `forEach` method takes a `Consumer` of any type that is a supertype of the type populating the array list. In particular, our `Consumer` will take any `Consumer` of `String`s. Now watch this.

```
jshell> ArrayList<String> al = new ArrayList<>();
al ==> []

jshell> al.add("aardvark")
$4 ==> true

jshell> al.add("bear");
$5 ==> true

jshell> al.add("carical");
$6 ==> true

jshell> al.forEach(printMe)
aardvark
bear
carical

jshell> al.forEach(System.out::println)
aardvark
bear
carical
```

What's cool is this `printMe` variable can store a *method*. This eliminates the need for writing a loop to display the items in a list. but wait... there is more!

10.8 Lambdas

Is there a way to customize the function object we want to create that is more flexible than a method reference? It turns out there is; these objects are called *lambdas*, and they are functions without a name. It is easy to make a lambda that is a Consumer. Here is a very simple example.

```
jshell> Consumer<String> kappa = e -> System.out.println(e.toUpperCase());
kappa ==> $Lambda$21/0x0000000800b5c440@6433a2

jshell> al.forEach(kappa)
AARDVARK
```

```
BEAR
CARICAL
```

Java is doing a little type inference here; it knows that `kappa` is a `Consumer<String>`, so it infers that the argument `e` is a `String`. The expresson on the right-hand side of the arrow is the return value for the lambda.

The `forEach` method takes a consumer, which basically is an action, and performs it on each item in list in index order.

10.8.1 Lambda Grammar

Lambdas can come in a variety of forms. We shall present a field guide here. The simplest form looks like this for a function with one argument and a return value.

```
x -> expr
```

The item `expr` is the return value of the lambda. If you have two arguments, you must enclose them in parentheses and your lambda looks like this.

```
(x, y) -> expr
```

You can enforce a masure of type safety by specifying types for arguments; here is an example of that.

```
(String s, int y) -> s.charAt(y)
```

You can have lambdas with more than one line of code. If you want to return something from such a lambda, you must do so explicitly. Here is a simple example that computes the square root of the sum of two numbers.

```
(int x, int y) ->
{
    int z = x > 0? x: -x;
    z += y > 0? y: -y;
    return z;
}
```

10.9 Comparators

Often we have a collection of objects that are sortable in various ways and we would like a mechanism that would conveniently afford us this ability. To this end, we now undertake a brief study of the functional interface `Comparator<E>`.

```
public interface Comparator<E>
{
    public int compare(E e1, E e2);
}
```

The design contract works like this. The set of elements `E` is given an ordering $\leq$ by this `compare` method. This ordering works as follows.

$$\texttt{e1} \leq \texttt{e2} \iff \texttt{compare}(\texttt{e1}, \texttt{e2}) \leq 0.$$

This ordering should satisfy these properties. For all `e1`, `e2` and `e3`,

1. If `e1`.equals(`e2`), then `compare`(`e1`, `e2`) returns 0.
2. if $\texttt{e1} \leq \texttt{e2}$ and $\texttt{e2} \leq \texttt{e3}$, $\texttt{e1} \leq \texttt{e3}$ (transitivity).
3. $(\texttt{e1} \leq \texttt{e2}) \vee (\texttt{e1} \leq \texttt{e2})$ is always true. In other words, any two elements of `E` are "related." Mathematically, this ordering is said to be *linear*. This is exactly what we expect for objects to be deemed sortable.

We will do a modest example here, which will allow us to sort an entry from a concordance by frequency of appearance and secondarily alphabetically, or alphabetically and then by frequency. We begin by showing the basic class.

```
public class ConcordanceEntry
{
    String word;
    int times;
    public ConcordanceEntry(String word, int times)
    {
        this.word = word;
        this.times = times;
    }
    @Override
    public String toString()
    {
        return String.format("%s:  %s", word, times);
    }
}
```

Next, we add some two static elements of type `Comparator`; notice that these variable are of interface type. We have made them `final` because they are exposed to the client.

```
public class ConcordanceEntry
{
```

```
    public static final Comparator<ConcordanceEntry> byTimes;
    public static final Comparator<ConcordanceEntry> byFrequency
    static
    {
        byTimes = (e1, e2) -> {
            if(e1.times != e2.times)
            {
                return e1.times - e2.times;
            }
            return e1.word.compareTo(e2.word);
        };
        byAlpha = (e1, e2) -> {
            if(!e1.word.equals(e2.times))
            {
                return e1.word.compareTo(e2.word);
            }
            return e1.times - e2.times;
        };
    }
    String word;
    int times;

    public ConcordanceEntry(String word, int times)
    {
        this.word = word;
        this.times = times;
    }
    @Override
    public String toString()
    {
        return String.format("%s:  %s", word, times);
    }
}
```

What is happening here? How are these two new static members actual objects?

If you visit the API guide, you will see that the interface `Comparator` is a functional interface. It specifies one method,

```
public int compare(T e1, T e2);
```

Java is performing type inference here we described in the our discussion of `Consumers`. It creates an object of class type that implements the specified method and it assigns the lambdas we assigned to the static variables as the `compare` method of that object. No specific named class is ever created.

In pre-8 Java, you would need to implement these using an anonymous inner class. The code for byAlpha would look like this.

```
byAlpha = new Comparator<ConcordanceEntry>(){
    public int compareTo(Concordance e1,
        ConcordanceEntry e2){
    if(!e1.times.equals(e2.times))
    {
        return e1.word.compareTo(e2);
    }
    return e1.times - e2.times;
};
```

This new construct of lambdas eliminates a good bit of boilerplate code that adds no meaning to what we are doing. What we want to pass here is *behavior* that is to be carried out by our comparator objects.

So what is the benefit of this? How do we sort? Let us demonstrate that in a main method. To implement this method, we introduce the static method `sort` in the class `java.util.Collections`. There are two methods by this name. We use the `sort` method which has the signature `[List<E>, Comparator<E>]`; this method sorts according to our comparator.

```
public static void main(String[] args)
{
    ArrayList<ConcordanceEntry> al =
        new ArrayList<>();
    al.add(new ConcordanceEntry("cow", 5));
    al.add(new ConcordanceEntry("pig", 2));
    al.add(new ConcordanceEntry("zebra", 3));
    al.add(new ConcordanceEntry("cow", 5));
    al.add(new ConcordanceEntry("zebra", 5));
    al.add(new ConcordanceEntry("elephant", 6));
    al.add(new ConcordanceEntry("eland", 1));
    al.add(new ConcordanceEntry("coati", 2));
    System.out.println("Unsorted:");
    for(ConcordanceEntry e: al)
    {
        System.out.println(e);
    }
    Collections.sort(al, byAlpha);
    System.out.println("Sorted by word:");
    for(ConcordanceEntry e: al)
    {
        System.out.println(e);
    }
```

```
        System.out.println("Sorted by frequency:");
        Collections.sort(al, byTimes);
        for(ConcordanceEntry e: al)
        {
            System.out.println(e);
        }
    }
```

Running this, here is the original list.

```
cow:  5
pig:  2
zebra:  3
cow:  5
zebra:  5
elephant:  6
eland:  1
coati:  2
```

Here it is, sorted first by word and second by frequency.

```
Sorted by word:
eland:  1
coati:  2
pig:  2
zebra:  3
cow:  5
cow:  5
zebra:  5
elephant:  6
```

Now we sort first by frequency then by word.

```
eland:  1
coati:  2
pig:  2
zebra:  3
cow:  5
cow:  5
zebra:  5
elephant:  6
```

10.10 Can I have many parents?

One question you might ask is, "Can I implement more than one interface?" The answer is yes. You do this.

```
public class ImplementingClass implements Interface1,
    Interface2, .... InterfaceN
{
    //code
}
```

When you do this, your class, or one of its ancestors, must implement the totality of all of the methods specified in the interface. This fact has an appealing and useful feel.

10.10.1 The Deadly Diamond

The next queston you might ask is "Can I have several parent classes?" The answer here is no. Class designers often speak of the "deadly diamond;" this is a big shortcoming of multiple inheritance and can cause it to produce strange behaviors. Imagine you have these four classes, Root, Left, Right and Bottom. Suppose that `Left` and `Right` extend `Root` and that `Bottom` were allowed to extend `Left` and `Right`.

Before proceeding, draw yourself a little inheritance diagram. Graphically these four classes create a cycle in the inheritance graph (*which in Java must be a rooted tree*).

Next, imagine that both the `Left` and `Right` classes implement a method `f` with identical signature and return type. Further, suppose that `Bottom` does not have its own version of `f`; it just decides to inherit it. Now imagine seeing this code fragment

```
Bottom b = new Bottom(....);
b.f(...)
```

There is a sticky problem here: Do we call the `f` defined in the class `Left` or `Right`? If there is a conflict between these methods, the call is not well–defined in our scheme of inheritance.

10.10.2 A C++ Interlude

There is a famous example of multiple inheritance at work in C++. There is a class `ios`, with children `istream` and `ostream`. The familiar `iostream` class

inherits from both `istream` and `ostream`. Since the methods for input and output do not overlap this works well.

However, the abuse of multiple inheritance in C++ has lead to a lot of very bad errors in code. Java's creators decided this advantage was outweighed by the error vulnerabilities of multiple inheritance.

The One-Parent Rule Every class has exactly one parent, except for `Object`, which is the root class. When you inherit from a class, you "blow your inheritance." The ability to inherit is very valuable, so we should only inherit when it yields significant benefits.

10.11 Abstract Classes

Interfaces do some nifty work for us. They afford us polymorphism which gives some desirable flexibility. They allow a project manager to specify names and signatures of methods for his programmers to develop. You can even insert default methods into interfaces. You cannot, however have a constructor for an interface, nor can you give an interface state. Interfaces also enjoy the advanage that a class can implement several of them, where you are always limited to one parent class via inheritance.

Sometimes, a group of related classes have a lot in common and you want to share common state and common code. This is in accordance with the 11th Commandment: *Thou shalt not maintain duplicate code.*

Fortunately there is a construct that allows us to "partially implement" a class, prevent the class from being instantiated, and force the children of the class to implement methods in a manner similar to that of an interface. Such a beast is called an `abstract class`.

In anticipation of the future, let us think again about shapes, but this time we will create a group of shape classes that can draw themselves in some kind of graphical environment. A shape that is going to draw itself needs to know its center, its color, and its size. What is different about various shape types is how they draw themselves.

For now, we will produce some "toy" classes as a means of demonstration; in a subsequent chapter we will use the scheme we create here to build a GUI application that renders shapes on a graphics surface.

Create these classes. Here is `Color.java`

```
public class Color
{
    private final int hexCode;
```

```
    public Color(int hexCode)
    {
        this.hexCode = hexCode;
    }
    @Override
    public String toString()
    {
        return String.format("Color(#%s)",
            Integer.toString(hexCode, 16));
    }
}
```

Here is `Canvas.java`

```
public class Canvas
{
    private final Pen pen;
    private double width;
    private double height;
    public Canvas(double width, double height)
    {
        this.width = width;
        this.height = height;
        this.pen = new Pen();
    }
    public Pen getPen()
    {
        return pen;
    }
    public double getWidth()
    {
        return width;
    }
    public double getHeight()
    {
        return height;
    }
    @Override
    public String toString()
    {
        return String.format("Canvas(%s, %s)", width, height);
    }
}
```

Here is `Pen.java`.

```
public class Pen
{
    private Color color;
    private double width;
    private void setColor(Color color)
    {
        this.color = color;
    }
    private void setWidth(double width)
    {
        this.width = width;
    }
    @Override
    public String toString()
    {
        return String.format("Pen(color = %s, width = %s)", color, width);
    }
}
```

Now let us make a class for a shape, `Shape.java`.

```
public class Shape
{
    private double centerX;
    private double centerY;
    private double color;
    private double size
    public Shape(double centerX, double centerY, double color, double size)
    {
        this.centerX = centerX;
        this.centerY = centerY;
        this.size = size;
    }
    public void draw(Pen pen)
    {
        //what do we do here?
    }
```

We see a problem: how do we draw a general `Shape`? That makes no sense whatsoever! This is a use case for abstract classes. Java has a keyword `abstract` that can be applied to classes and methods. We will leave the `draw` method undefined just as we would in an interface. To do this, we must declare the method `abstract`, like so

```
public abstract draw(Pen pen);
```

If we mark any method in a class `abstract`, we must also mark the class `abstract`. Our class now looks like this.

```
public abstract class Shape
{
    private double centerX;
    private double centerY;
    private double color;
    private double size
    public Shape(double centerX, double centerY, double color, double size)
    {
        this.centerX = centerX;
        this.centerY = centerY;
        this.size = size;
    }
    public abstract void draw(Pen pen);
```

Here are the rules of the road for abstract classes.

1. If you omit a method's body in a class you *must* declare that method `abstract`.
2. If *any* method in a class is declared `abstract`, the class itself must be declared `abstract`.
3. You may declare any class you create `abstract`. By so doing, you prevent any instances of it from being created; do this if it make no sense for an instance of your class to be created.
4. You may not create instances of any abstract class.
5. You can create variables of abstract class type. They can point at any object of any descendant type. Both the visibility and delegation principles apply.
6. If you extend an abstract class and the child class is not abstract, you must implement all abstract methods in the abstract class, as well as any abstract methods in ancestor classes of the abstract class.
7. A class that is not abstract is said to be `concrete`.

A variable of type `Shape` can point at any object of a descendant type. You could now make classes for circles, rectangles, squares, and other graphical objects. These can each implement the `draw` method of `Shape`.

How do I know if a standard library class is abstract? Look in the Java API guide and find the class `AbstractList`; clearly it will be abstract. Go to the top of the page. You will see the fully-qualified name, the family tree, and then its implemented interfaces and direct descendants. Just below that you see this

```
public abstract class AbstractList<E>
extends AbstractCollection<E>
implements List<E>
```

The first line tells all: See the word `abstract`?

So, in summary, you can declare any class abstract and instances of it cannot be created. You can declare methods in a class abstract and they cannot have a method body. Any child class must override these methods unless, it too, is abstract. Any class containing an abstract method must be marked abstract. However, an abstract class is not required to have any abstract methods.

10.12 A Brief Trip into Functional Programming

If you look in the API guide for `ArrayList`, you will find a mysterious method called `stream`. This is among the default methods inherited from the interface `java.util.Collection`. Here is its method detail.

```
default Stream<E> stream()
```

Returns a sequential Stream with this collection as its source.

This method should be overridden when the spliterator() method cannot return a spliterator that is IMMUTABLE, CONCURRENT, or late-binding. (See spliterator() for details.)

Implementation Requirements: The default implementation creates a sequential Stream from the collection's Spliterator.

Returns: a sequential Stream over the elements in this collection

Since: 1.8

So, what is a `Stream<E>`? A stream is a read-only Netflix-like view of a collection of objects. So for an `ArrayList`, a stream containing the list's objects can be created and they are shown in the list's order. Creating a stream creates a `source`. Creating the stream causes little or nothing to happen. You must then take some action on the stream to consume it; consumption of a stream triggers the actual processing of the data in the list.

Here are some methods that trigger consumption.

- `forEach(Consumer<? super E> action` This works just as it does on an array list.

- `count()` This counts the number of elements in the stream and returns the count as an integer.
- `collect(Collectors.toList())` will collect the contents of the stream into a list. returns the count as an integer.

Wait a minute! This seems kind of useless! In its present form, admittedly so. However, we have only described the bread in the sandwich (a source and something that consues), but not the goodness inside. That's where this becomes a very powerful tool.

The goodies inside consist of *transformers* that transform and filter the items in your collection. Transformers are basically alimenatary canals that "eat" from streams and subsequently "excrete" a stream. Let us now meet the `Predicate` family. They reside in package `java.util.function`. They are all functional interfaces, so we can use lambdas where called for.

- `Predicate<T>` Objects of this type can be specified using a lambda whose argument is of type `T` and whose return value is a `boolean`.
- `IntPredicate`Objects of this type can be specified using a lambda whose argument is of type `int` and whose return value is a `boolean`.
- `LongPredicate`Objects of this type can be specified using a lambda whose argument is of type `long` and whose return value is a `boolean`.
- `DoublePredicate` Objects of this type can be specified using a lambda whose argument is of type `double` and whose return value is a `boolean`.

Let us see `filter` at work.

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
public class Filter
{
    public static void main(String[] args)
    {
        ArrayList<String> words = new ArrayList<>();
        words.add("aardvark");
        words.add("African gazelle");
        words.add("ostrich");
        words.add("yak");
        words.add("zebra");
        words.add("vampire bat");
        words.add("anaconda");
        words.add("tapir");
        System.out.println(words);
```

```
        words.stream()
             .filter( s -> s.compareToIgnoreCase("m") < 0)
             .forEach(System.out::println);
        List<String> out = words.stream()
                                .filter( s -> s.compareToIgnoreCase("m") < 0)
                                .collect(Collectors.toList());
        System.out.println(out);
    }
}
```

Let us point out some features of this little program. Notice the typographical convention of aligning "on the dots" where we first make a stream from the `ArrayList words`. Note that only the last line in the progression ends with a semicolon. Actually, this is a single line of code, but we do not want great long lines of code because they become unreadable. Combining a filter with a foreach statement eliminates the need for creating a for loop with an if statement inside for filtering a list.

10.12.1 Declarative vs. Imperative

The streams API and the functional programming interface in Java enable a style of programming that is *declarative*; to wit, you get exactly what you want by saying what you want. The line

```
        words.stream()
             .filter( s -> s.compareToIgnoreCase("m") < 0)
             .forEach(System.out::println);
```

says, "Make a stream from my list, filter in the first half of the alphabet, and print the entries out. The imperative form looks like this.

```
for (String s: words)
{
    if( x.compareToIgnoreCase("m") < 0)
    {
        System.out.println(s);
    }
}
```

Here we are "spoon-feeding" Java every minute step rather than telling it what we want.

10.12.2 Using `map`

This is the bread-and-butter means for making a stream of objects of one type into a stream of objects as another or the same type. Let's see how we can use it to get our animal list to print upper-case. Notice we have two transformers in our stream sandwich.

```
        words.stream()
             .filter( s -> s.compareToIgnoreCase("m") < 0)
             .map( s -> s.toUpperCase())
             .forEach(System.out::println);
```

The output looks like this.

```
AARDVARK
AFRICAN GAZELLE
ANACONDA
```

To do this imperatively, you might do this.

```
for (String s: words)
{
    if( x.compareToIgnoreCase("m") < 0)
    {
        System.out.println(s.toUpperCase());
    }
}
```

Map, like its friend filter, comes in various guises. Here they come.

1. `mapToInt` This changes an object stream into a stream of (primitive) integers, an `IntStream`.
2. `mapToDouble` This changes an object stream into a stream of (primitive) doubles, a `DoubleStream`.
3. `mapToLong` This changes an object stream into a stream of (primitive) longs, an `LongStream`.

Chapter 11

Files and Exceptions

11.0 Introduction

This chapter will introduce you interacting with the file system. At the same time, it will be necessary to learn about the concept of exception handling. The reason for this is that when you access a file and it is not present or you do not have permission to read it, an exception is generated.

These exceptions are beyond programmer and, often user, control. What you don't want them doing is crashing your program. So you will learn how to run code with these sorts of hazards and learn how to recover from the problem at hand gracefully. We will begin this chapter by discussing exceptions.

11.1 Exceptions

We have already seen these. For example, when creating `BigFraction`, we threw an `IllegalArgumentException` when a client programmer attempted to create a fraction with a zero denominator. There is little doubt you have encountered a `NullPointerException` when you forgot to use `new` to allocate memory for an object or a `IndexOutOfBounds` exception when working with a collection. Consider this unfortunate code.

```
import java.util.ArrayList;
public class DumbError
{
    private static ArrayList<String> roster;

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

```
    {
        roster.add("Doofus McDuff");
        System.out.println(roster);
    }
}
```

This code compiles.

```
$ javac DumbError.java
```

Now let us run it.

```
$ java DumbError
Exception in thread "main" java.lang.NullPointerException
    at DumbError.main(DumbError.java:8)
$
```

Doom. You see that a `NullPointerException` got thrown. The compiler did not catch the error; an exception is a run-time error. You will see that on line 8 the offending code is

```
roster.add("Doofus McDuff");
```

This occured because we never did this

```
roster = new ArrayList<>();
```

We attempted to call a method on an object pointing to the graveyard state `null`. Insert this line we just showed and your program will run without error. Other exceptions you probably have run across include these. All of these exceptions are examples of *run-time exceptions*. Most run-time exceptions are caused by programmer errors. When you encountered them, you saw that your program died on the spot and that a stack trace was generated.

There is another species of exception you are about to meet called *checked exceptions*. These exceptions occur because of events that are beyond user, or programmer, control. These sorts of exceptions crop up when handling files; being able to handle them is prerequisite to doing any interaction with the file system.

Java provides a mechanism called *exception handling* that provides a parallel track of return from functions so that you can avoid cluttering the ordinary execution of code with endless error-handling routines.

Exceptions are objects that are "thrown" by various methods or actions. In this chapter we will learn how to handle (catch) an exception. By so doing

we allow our program to recover and continue to work. Failure to catch and exception results in a flood of nasty text from Java (a so-called "exploding heart"). Crashes such as these should be extremely rare in production-quality software. We can use exceptions as a means to protect our program from such dangers as user abuse and from such misfortunes as crashing whilst attempting to gain access to an nonexistent or prohibited resource. Many of these hazards are beyond both user and programmer control.

When you program with files or with socket connections, the handling of exceptions will be mandatory; hence the need for this interlude before we begin handling files.

Let us now turn to understanding how exceptions fit into Java's class hierarchy.

Progrmming Exercise

1. What happens if you try to compute 1/0 using integers?
2. What happens if you try to compute 1/0 using doubles?
3. What happens if you make the call `Math.sqrt(-1)`?
4. What happens if you make the call `Math.log(-1)`?

11.2 The `Throwable` Subtree

Go to the Java API guide and pull up the class `Exception`. The family tree is is as follows.

```
java.lang.Object
java.lang.Throwable
java.lang.Exception
```

The class name `Throwable` is a bit strange; one would initially think it were an interface. It is, however, a class. The class `java.lang.Exception` has a sibling class `java.lang.Error`.

When objects of type `Error` are thrown, it is not reasonable to try to recover. These things come from problems in the Java Virtual Machine, bad memory problems, or problems from the underlying OS. We just accept the fact that they cause program death. Continuing to proceed would just lead to a chain of ever-escalating problems.

Objects of type `Exception` are thrown for more minor problems, such as an attempt to open a non-existent file for reading, trying to convert an unparseable

string to an integer, or trying to access an entry of a string, array, or array list that is out of bounds.

Let us show this mechanism at work. For example, if you attempt to execute the code

```
int foo = Integer.parseInt("gobbledegook");
```

you will be rewarded with an exception. To see what happens, create this program `MakeException.java`.

```
public class MakeException
{
    public static void main(String[] args)
    {
        int foo = Integer.parseInt("gobbledegook");
    }
}
```

This program compiles happily. You will see that the infraction we have here is a run-time error, as is any exception.

When you run the program you will see this acrimonious screed.

```
unix> java MakeException
Exception in thread "main" java.lang.NumberFormatException:
    For input string: "gobbledegook"
at java.base/java.lang.NumberFormatException.forInputString(
       NumberFormatException.java:68)
at java.base/java.lang.Integer.parseInt(Integer.java:652)
at java.base/java.lang.Integer.parseInt(Integer.java:770)
at MakeException.main(MakeException.java:5)
unix>
```

The exploding heart you see here shows a stack trace. This shows how the exception propagates through the various calls the program makes. To learn what you did wrong, you must look in this list for your file(s). You will see the offending line here.

```
    at MakeException.main(MakeException.java:5)
```

You are being told that the scene of the crime is on line 5, smack in the middle of your `main` method. The stack trace can yield valuable clues in tracking down and extirpating a run-time problem such as this one.

We have seen reference to "throw" and "throws" before. Go into the API guide and bring up the class `String`. Now scroll down to the method summary and click on the link for the familiar method `charAt()`. You will see this notation.

Throws:

> `IndexOutOfBoundsException` - if the `index` argument is negative or not less than the length of the string.

Let us now look up this `IndexOutOfBoundsException`. The family tree reveals that this class extends the `RuntimeException` class. The purpose of this exception is to terminate the execution of the program since you are in an error state. We can, however, run code in response to the occurrence of an exception, so our program does not crash.

11.3 Throwing an Exception

You can throw exceptions when something is about to go wrong. For example in our BigFraction class, we threw an exception if a denominator of zero was passed. Here it is.

```
public BigFraction(BigInteger num, BigInteger denom)
{
    if(denom.equals(BigInteger.ZERO)
    {
        throw new IllegalArgumentException();
    }
    (rest of constructor)
}
```

It is best, if at all possible, to use a standard library exception. You can create your own exceptions by inheriting from any class in the `Exception` subtree. If you inherit from `RuntimeException`, the caller does not have to handle the exception as shown in the next section.

11.4 Checked and Run-Time Exceptions

There are two types of exceptions that exist: runtime exceptions and all others, which are called checked exceptions. How do you know if an exception is a

`RuntimeException`? Just look up its family tree and see if it is a descendant of `RuntimeException`. So far in our study of Java, we have only seen runtime exceptions.

Checked exceptions, on the other hand, are usually caused by situations beyond programmer, or even end-user control. Suppose a user tries to get a program to open a file that does not exist, or a file for which he lacks appropriate permissions. Another similar situation is that of attempting to create a *socket*, or a connection to another computer. The host computer may not allow such connections, it could be down, or it could even be nonexistent. These situations are not necessarily the user's or programmer's fault.

Checked exceptions must be *handled*; this process entails creating code to tell your program what to do in the face of these exceptions being thrown. It is entirely optional to handle a runtime exception.

Sometimes a runtime exception will be caused by user error; in these cases it is appropriate to use exception handling to fix the problem. For example if a user is supposed to enter a number into a `TextField` the hapless fool enters a string that is not numeric, your program might try to use `Integer.parseInt` to convert it into an integer. Here we see a problem created by an end-user. This user should be protected and this error should be handled gracefully so that (bumbling) user can go about his business. You always want to protect the end-user from exceptions if it is at all feasible or reasonable. Remember: *Never reward a paying customer with death.* It's bad for business. Now let us erect some scaffolding for handling files, as we will introduce the idea of handling exceptions in the context of fileIO.

11.5 The `Path` to Perdition

We begin with the interface `java.nio.files.Path`. It has a list of methods that is specifies for handling locations in your file system. Said locations might or might not exist. Since `Path` is an interface, you cannot create instances of a `Path` using `new`. However, this interface comes equipped with a static method `of` Here is the method detail.

```
static Path of(String first, String...  more)
```

Returns a Path by converting path string, or a sequence of strings that when joined form a path string. If `more` does not specify any elements then the value of the first parameter is the path string to convert. If more specifies one or more elements then each non-empty string, including first, is considered to be a sequence of name elements and is joined to form a path string. The details as to how the Strings are joined is provider specific but typically they will be joined using the name-separator as the separator. For example, if the name separator is `"/"` and `getPath("/foo","bar","gus")` is invoked, then the path

string "/foo/bar/gus" is converted to a Path. A Path representing an empty path is returned if first is the empty string and more does not contain any non-empty strings.

The Path is obtained by invoking the getPath method of the default FileSystem.

Note that while this method is very convenient, using it will imply an assumed reference to the default FileSystem and limit the utility of the calling code. Hence it should not be used in library code intended for flexible reuse. A more flexible alternative is to use an existing Path instance as an anchor, such as:

```
Path dir = ...
Path path = dir.resolve("file");
```

Parameters:
first - the path string or initial part of the path string
more - additional strings to be joined to form the path string
Returns:
the resulting Path **Throws:**
InvalidPathException - if the path string cannot be converted to a Path
Since:
11
See Also:
FileSystem.getPath(java.lang.String, java.lang.String...)

Note that the class Paths merely implement the of method of this class. The Paths is probably a dead-end class, so you should use Path.of in preference to its get method. It will be the workhorse for us in this chapter.

Let us now take a tour of Path methods. Begin by creating a directory named zoo with these contents

```
(base) MAC:Sat Nov 28:10:33:s1> ls -Rl zoo
total 0
-rw-r--r--  1 morrison  staff     0B Nov 27 16:03 capybara
drwxr-xr-x  6 morrison  staff   192B Nov 28 10:32 cats
-rw-r--r--  1 morrison  staff     0B Nov 27 16:03 dingo
-rw-r--r--  1 morrison  staff     0B Nov 27 16:03 eland
drwxr-xr-x  6 morrison  staff   192B Nov 28 10:32 reptiles

zoo/cats:
total 0
-rw-r--r--  1 morrison  staff     0B Nov 27 16:03 bobcat
-rw-r--r--  1 morrison  staff     0B Nov 28 10:32 cheetah
-rw-r--r--  1 morrison  staff     0B Nov 28 10:32 lion
-rw-r--r--  1 morrison  staff     0B Nov 28 10:32 tiger
```

```
zoo/reptiles:
total 0
-rw-r--r--  1 morrison  staff     0B Nov 27 16:03 alligator
-rw-r--r--  1 morrison  staff     0B Nov 28 10:32 anaconda
-rw-r--r--  1 morrison  staff     0B Nov 28 10:32 cobra
-rw-r--r--  1 morrison  staff     0B Nov 28 10:32 iguana
```

Enter this directory and fire up `jshell`. We begin by getting our current working directory as a `Path` and determining its absolute position in our file system. Note the use of `isAbsolute` to tell if a path is absolute or relative.

```
jshell> Path zoo = Path.of(".")
zoo ==> .

jshell> Path absZoo = zoo.toAbsolutePath()
absZoo ==> /Users/morrison/book/S1/zoo/.

jshell> zoo.isAbsolute()
$3 ==> false

jshell> absZoo.isAbsolute()
$4 ==> true
```

Here is a look up the file family tree.

```
jshell> absZoo.getParent()
$8 ==> /Users/morrison/book/S1/zoo

jshell> absZoo.getParent().getParent()
$9 ==> /Users/morrison/book/S1
```

The result is disappointing when we do this on `zoo`, but we sho a hackish workaround here.

```
jshell> zoo.getParent()
$10 ==> null

jshell> zoo.toAbsolutePath().getParent().getFileName()
$11 ==> zoo
```

OPC Note In older OPC, you will see reference to the class `java.io.File`; this is the predecessor to the modern `Path` interface, which you should prefer. However, you can convert a `Path` object to a `File` by calling the `toFile` method on it. The older `File` class has a `toPath` method to convert from `File` to `Path`. You should prefer the `Path` interface when writing new code.

java.nio.file.Paths This static service class appears to be an abortive effort and you should avoid it. Construct new paths using the static `of` method.

Interacting with the File System The next class for you to know about is `Files`; this is the workhorse class for fileIO. The `Path` interface and `Files` jointly cover all of the territory previously covered by the old `java.io.File` class and a whole lot more.

That name `Files` should give you a hint. You have seen the pluralized class name before in `java.util.Collections` and `java.util.Arrays`.

Alex, give me "common bonds" for $800. Alex replies, "Here is the answer: this is the common bond between these three classes."

The smart contestant says, "What is being a static service class?" Bingo, add $800 to that contestant's score.

To move around in our little file tree and to manipulate it, we will use `Files`. This is a static service class that gives us access to the file system, as well as an array of other useful services..

Boola! Boola! The `Files` class provides some useful predicates to determine file attributes. As you might expect, their names begin with `is`. These four check out a file's type.

- `isDirectory(Path path)` This returns `true` if the path points at a directory.
- `isRegularFile(Path path)` This returns `true` if the path points at a at a regular file.
- `isHidden(Path path)` This returns `true` if the path points at a a hidden (in UNIX a dotfile) file.
- `isSymbolicLink(Path path)` This returns `true` if the path points at a at a symbolic link (an alias for a file or a directory).

These tell you about file permissions.

- `isReadable(Path path)` This returns `true` if the path points at a file you have read permissions for.
- `isWritable(Path path)` This returns `true` if the path points a file you have write permissions for.
- `isExecutable(Path path)` This returns `true` if the path points a file you have execute permissions for.
- `isDirectory(Path path)` This returns `true` if the path points

These two will tell you basic "identity" information.

- `exists(Path path, LinkOption... option )` This returns `true` if the path points at a file that exists. The second argument is entirely optional and we won't bother with it.
- `isSameFile(Path path1, Path path2)` This returns `true` if the both paths point at the same file.

You can also create and delete files and directories.

- `createFile(Path path, FileAttribute<?>... option )` This returns `true` if the path points at a file that exists. The second argument is entirely optional, and allows you to specify file permissions.
- `delete(Path path)` This deletes the file at the specified path. point at the same file.

11.6 Reading a Text File

Open the API page for the class `java.nio.Files`. We are going to write a class named `Cat.java` which accepts a string as a command-line argument that (should) be a name of an existing file and which puts the file to `stdout`. For the sake of simplicity, we will do this all in the `main` method. Now get the method detail for the static method `readAllLines`. This method reads the lines of the file into a `List<String>`. Let us attempt this.

```
import java.util.List;
import java.nio.file.Files;
import java.nio.file.Path;
public class Cat
{
    public static void main(String[] args)
    {
        Path path = Path.of(args[0]);
        List<String> fileContents = Files.readAllLines(path);
        for(String s: fileContents)
        {
            System.out.print(s);
        }
    }
}
```

Now we compile and see this.

```
unix> javac Cat.java
Cat.java:9: error: unreported exception IOException;
```

```
    must be caught or declared to be thrown
        List<String> fileContents = Files.readAllLines(path);
                                                       ^
1 error
```

Uh oh. Because an `IOException` is a checked exception, action on our part is required. We begin by enclosing our "dangerous code" in a `try` block. This must be followed by a `catch` block for an `IOException`. Note that a new import is needed.

```
import java.util.List;
import java.nio.file.Files;
import java.nio.file.Path;
import java.io.IOException;
public class Cat
{
    public static void main(String[] args)
    {
        try
        {
            Path path = Path.of(args[0]);
            List<String> fileContents = Files.readAllLines(path);
            for(String s: fileContents)
            {
                System.out.print(s);
            }
        }
        catch(IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

Now, create a text file to run this on.

```
This is a test
it is only a test.
Let's see if this works.
```

Let's run it.

```
unix> java Cat.java test.txt
This is a testit is only a test.Let's see if this works.unix>
```

Evidently all of the newlines get amputated. Let's put 'em back; just change `System.out.print` to `System.out.println`.

```
unix> java Cat.java test.txt
This is a test
it is only a test.
Let's see if this works.
```

Et Voila!

Can we shorten this code? Yes, if we avail ourselves of the `forEach` method for lists.

```
import java.util.List;
import java.nio.file.Files;
import java.nio.file.Path;
import java.io.IOException;
public class Cat
{
    public static void main(String[] args)
    {
        try
        {
            Path path = Path.of(args[0]);
            List<String> fileContents = Files.readAllLines(path);
            fileContents.forEach(System.out::println);
        }
        catch(IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

We are using a *method reference* in the `forEach` method.

Next, for a little perverse fun, let us run our program on a nonexistent file `heffalump.txt`. Fury is unleashed by the JVM.

```
base) MAC:Tue Dec 22:15:57:s9> java Cat.java heffalump.txt
java.nio.file.NoSuchFileException: heffalump.txt
        at java.base/sun.nio.fs.UnixException.translateToIOException(UnixException.java
        at java.base/sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:
        at java.base/sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:
        at java.base/sun.nio.fs.UnixFileSystemProvider.newByteChannel(UnixFileSystemPro
        at java.base/java.nio.file.Files.newByteChannel(Files.java:375)
        at java.base/java.nio.file.Files.newByteChannel(Files.java:426)
        at java.base/java.nio.file.spi.FileSystemProvider.newInputStream(FileSystemProv
        at java.base/java.nio.file.Files.newInputStream(Files.java:160)
        at java.base/java.nio.file.Files.newBufferedReader(Files.java:2916)
```

```
        at java.base/java.nio.file.Files.readAllLines(Files.java:3396)
        at java.base/java.nio.file.Files.readAllLines(Files.java:3436)
        at Cat.main(Cat.java:12)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImp
        at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcc
        at java.base/java.lang.reflect.Method.invoke(Method.java:564)
        at jdk.compiler/com.sun.tools.javac.launcher.Main.execute(Main.java:415)
        at jdk.compiler/com.sun.tools.javac.launcher.Main.run(Main.java:192)
        at jdk.compiler/com.sun.tools.javac.launcher.Main.main(Main.java:132)
```

Scan through this wreckage; it reveals that the source of the exception was on line 12 in our file. Let's handle it and cut down on the Wagnerian drama.

```
import java.util.List;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.NoSuchFileException;
import java.io.IOException;
public class Cat
{
    public static void main(String[] args)
    {
        try
        {
            Path path = Path.of(args[0]);
            List<String> fileContents = Files.readAllLines(path);
            fileContents.forEach(System.out::println);
        }
        catch(NoSuchFileException ex)
        {
            System.err.printf("File %s does not exist.\n", args[0]);
        }
        catch(IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

This is better.

```
unix> java Cat heffalump.txt
File heffalump.txt does not exist.
```

This reveals something else. We can have several `catch` blocks. Order is im-

portant. Put the most specific exceptions (lower on the inheritance tree) at the top and the most generals ones at the bottom. Only one `catch` block will ever by executed. Here is one other minor tweak. Dum-dum user just might forget a command-line argument. Just at this at the top of your `main` method before the `try` block.

```
if(args.length == 0)
{
    System.err.println("A command-line argument is required");
}
```

11.7 Dealing with Dyspepsia

What if we want to process a humongous file? Reading it all at once could be a huge memory hog. Can we be more efficient? Happily the tools are at hand. We will bring `newBufferedReader` to bear on the problem. During this exercise, you will learn about *try with resources* that will automatically close any file you open. Let us rewrite `Cat.java` using this method.

```
import java.util.List;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.NoSuchFileException;
import java.io.BufferedReader;
import java.io.IOException;
public class Cat
{
    public static void main(String[] args)
    {
        if(args.length == 0)
        {
            System.err.println("A command-line argument is required");
        }
        try
        {
            Path path = Path.of(args[0]);
            BufferedReader br  = Files.newBufferedReader(path);
            String line = "";
            while( (line = br.readLine()) != null)
            {
                System.out.println(line);
            }
            br.close();
        }
```

```
        catch(NoSuchFileException ex)
        {
            System.err.printf("File %s does not exist.\n", args[0]);
        }
        catch(IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

So what has happened? A `BufferedReader` creates a connection to a file. Hidden from you is the buffer, which stores a chunk of the file in your program's memory. Usually this chunk is of size 4K. In the beginning the `BufferedReader` grabs a chunk of text. We then have it reading the file a line at at time. When the buffer is empty, another chunk of file is hoovered into the buffer. So, if we are reading a large file, our memory footprint is far smaller than if we got the whole file at once using `readAllLines`. Also, we are not pestering the operating system with a zillion requests for chunks of the file.

A problem remains. If an exception occurs, the file might not close. There is a smart way to deal with this called *try with resources*. Let us see what that looks like.

```
import java.util.List;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.NoSuchFileException;
import java.io.BufferedReader;
import java.io.IOException;
public class Cat
{
    public static void main(String[] args)
    {
        if(args.length == 0)
        {
            System.err.println("A command-line argument is required");
        }
        Path path = Path.of(args[0]);
        try
        (
            BufferedReader br  = Files.newBufferedReader(path);
        )
        {
            String line = "";
            while( (line = br.readLine()) != null)
```

```
            {
                System.out.println(line);
            }
        }
        catch(NoSuchFileException ex)
        {
            System.err.printf("File %s does not exist.\n", args[0]);
        }
        catch(IOException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

Notice we moved the declaration of `path` so it is in scope throughout `main`. We no longer need to close the file, because try with resources automatically does this. As a result, a great deal of bother you will never see will never take place.

How do I know if I can use try with resources? Just check to see if the class you are using implements `java.lang.AutoCloseable`. This interface specifies a single method, `public void close()`. Look at the API page; a lot of things implement it. If you write some kind of file-handling class, you should implement it, too.

That `while` loop looks so awful Pssst... Here is a little magic. Ditch this code

```
String line = "";
while( (line = br.readLine()) != null)
{
    System.out.println(line);
}
```

for this:

```
br.lines()
  .forEach(System.out::println);
```

Ooh, sweet. How does that work? In brief, the call `br.lines()` gives you a Netflix-like streaming view of the file. The lines come in a stream. The `forEach` method applies `System.out.println` to each line in the stream. You have had a preview of the Streams API which will be covered in more detail later. It is a stupendously powerful appratus that will make your code shorter, more readable, and more expressive.

11.8 Writing a Text File

One might just think that if there is a `BufferedReader` that there could be a `BufferedWriter`. Correct. And now that we know about exception handling, writing to a file should be a fairly simple process. Let's do it. Here an idea. Let's generate an HTML trig table from 0 to 90 degrees for sine and cosine.

What do we need? We will write a function that creates the table rows, and functions that compute sine and cosine in degrees.

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.io.BufferedWriter;
import java.io.IOException;
public class Triggie
{
    private static final double FACTOR = Math.PI/180;
    public static void main(String[] args)
    {
    }
    private static double sinDeg(int x)
    {
        return Math.sin(FACTOR*x);
    }
    private static double cosDeg(int x)
    {
        return Math.cos(FACTOR*x);
    }
    private static String makeRow(int x)
    {
        return String.format("<tr><td>%d</td><td>%.4f</td><td>%4f</td></tr>\n",
            x, sinDeg(x), cosDeg(x));
    }
}
```

Now we write the `main` method. We will use try with resources.

```
public static void main(String[] args)
{
    Path path = Path.of("trigTable.html");
    try(BufferedWriter bw = Files.newBufferedWriter(path))
    {
        bw.write("<table>\n<tr><th>x</th><th>sin(x)</th><th>cos(x)</th></tr>\n");
        for(int k = 0; k <= 90; k++)
        {
            bw.write(makeRow(k));
```

```
        }
        br.write("<table>\n");
    }
    catch(IOException ex)
    {
        ex.printStackTrace();
    }
}
```

Note the pleasing parallelism here.

11.9 Buffered IO with Binary Files

Here we introduce some new classes in `java.io`. These manage unbuffered and buffered byte streams.

- `java.io.InputStream`
- `java.io.OutputStream`
- `java.io.BufferedInputStream`
- `java.io.BufferedOutputStream`

We will create a program that copies files containing raw bytes. In our example, we will use an image file. This has been tested on a 300 megabyte video file and did the job pleasingly quickly. Let's get started with some basic stuff, inserting the needed imports and wrangling the command-line arguments.

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.nio.file.NoSuchFileException;
public class BinaryCopy
{
    public static void main(String[] args)
    {
        String donor = args[0];
        String recipient = args[1];
        if(args.length < 2)
        {
            System.err.println("Two command-line arguments are needed");
```

```
        }
        Path inFile = Path.of(donor);
        Path outFile = Path.of(recipient);
    }
}
```

Now let us proceed to creating the try with resources header.

```
try
(
    InputStream in = new
        BufferedInputStream(Files.newInputStream(inFile));
    OutputStream out =
        new BufferedOutputStream(Files.newOutputStream(outFile));
)
```

We now append the usual catch blocks.

```
try
(
    InputStream in = new
        BufferedInputStream(Files.newInputStream(inFile));
    OutputStream out =
        new BufferedOutputStream(Files.newOutputStream(outFile));
)
{
}
catch(NoSuchFileException ex)
{
    System.err.printf("File %s not found.\n", donor);
}
catch(IOException ex)
{
    ex.printStackTrace();
}
```

The coup d'grace is stunningly simple.

```
try
(
    InputStream in = new
        BufferedInputStream(Files.newInputStream(inFile));
    OutputStream out = new
        BufferedOutputStream(Files.newOutputStream(outFile));
)
```

```
{
    byte[] bytes = Files.readAllBytes(inFile);
    out.write(bytes);
}
catch(NoSuchFileException ex)
{
    System.err.printf("File %s not found.\n", donor);
}
catch(IOException ex)
{
    ex.printStackTrace();
}
```

The docs recommend against this. However it will copy a file containing several hundred megabytes with pleasing dispatch.

Bibliography

[1] I. Anaconda, *Anaconda download site.* https://www.anaconda.com/products/individual.

[2] J. Bloch, *Effective Java, Third Edition*, Addison-Wesley, 2017.

[3] U. o. H. a. M. College of Engineering, *Mastering the* vi *Editor*, http://www.eng.hawaii.edu/Tutor/vi.html, 2020.

[4] N. Ninja, *The net ninja python 3.* https://www.youtube.com/playlist?list=PL4cUxeGkcC9idu6GZ8EU_5B6WpKTdYZbK.

[5] I. Python Foundation, *Python built-in types.* https://docs.python.org/3/library/functions.html.

[6] A. Robbins, *Learning the vi and vim Editors*, O'Reilly Associates, 2008.

[7] C. Schaefer, *Python tutorials.* https://www.youtube.com/watch?v=YYXdXT2l-Gg&list=PL-osiE80TeTt2d9bfVyTiXJA-UTHn6WwU&ab_channel=CoreySchafer.

[8] J. Wang, *Vi for smarties.* http://www.jerrywang.net/vi.

[9] Wikipedia, *Duck typing.* https://en.wikipedia.org/wiki/Duck_typing.

[10] ——, *Two's complement.* http://en.wikipedia.org/wiki/IEEE_754.