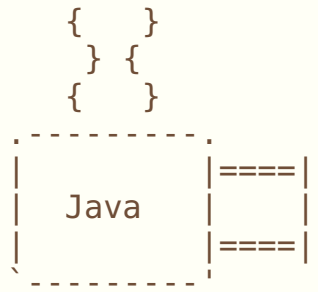


Introduction to Java

Joe Raskind

Compiled: August 04, 2026

What is a Program?

Introduction

> “
>
> *Computational processes are abstract beings that inhabit computers…People create programs to direct processes. In effect, we conjure the spirits of the computer with our spells. — Harold Abelson and Gerald Sussman*

I can claim with absolute certainty that if you are reading these notes you have used a computer program before. I can claim with less certainty, although I think there's pretty decent odds considering undergraduates are often American citizens, that you have driven, or have been driven by, a car. I'd say there's a good chance that if that's the case you likely don't know all about the mechanics involved in make a car go: the inner workings of the combustion engine, the way the transmission changes the direction of rotation, how the break fluid transfers pressure from your foot to stop the wheels, etc. This is much the same how you may not have known anything about how when you boot up your laptop that there's an entire world of transistors that begin firing off to let you eventually launch your browser of choice to browse the internet.

It's fine to not know how a car works if you're not in the market to build cars. It's less fine to not know how a computer program works if you're in the market to make computer programs. The point of this course is to get you closer to understanding how a computer program works and to get you on the right track to build up your knowledge in the future.

Why Do We Program?

Computer programming is a practical effort in real-world problem solving. It is primarily an intellectual exercise involving the careful balancing of the rules of computation with the desired outcomes envisioned by the programmer. In the most obvious way, the fact that computers dominate our daily lives implies that they must have *some sort of use value*, i.e. there must be a reason to program them. When asked, sophomore computer science students often come up with the following points:

- To **automate** tasks
- To **simplify** tasks
- To **store** the results of tasks

Once a computer program has been written and transformed into a runnable state, a human being no longer has to perform the same mental calculus that was needed in order to write it in the first place. They simply run the program and record the results (or maybe the results are recorded for them!). For example, the first computer programs were written to help speed up ballistics calculations for naval warfare—all the user needed to do was provide information about the variables involved and the machine would perform all necesarry computations.

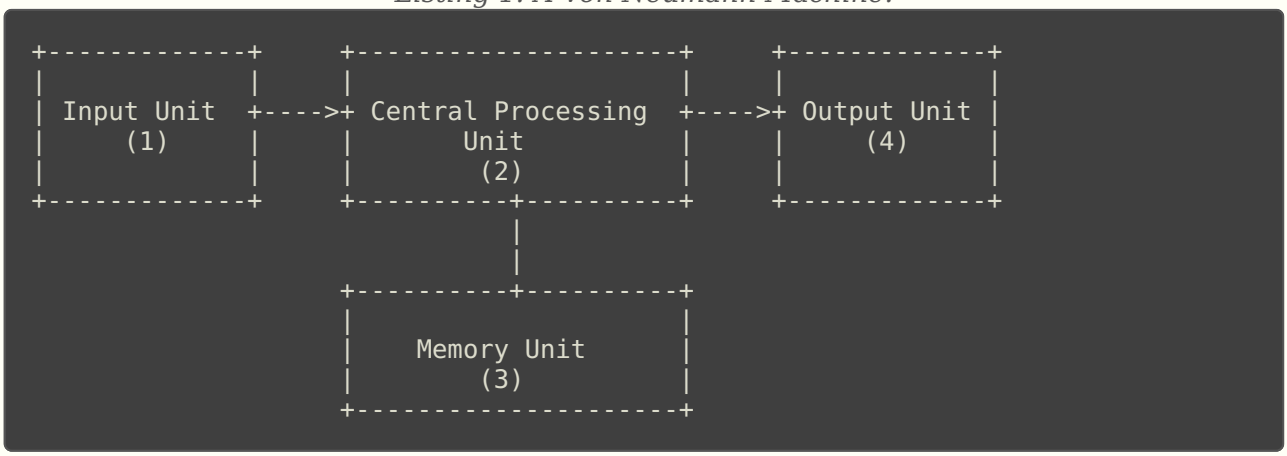
What Are the Components?

Nearly all modern day computers follow a design philosophy which was established in the late 1940s: the Von Neumann Machine. Named after the Hungarian-American mathematician Jon von Neumann, the Von Neumann Machine is a logical abstraction of a computation machine which, in its simplest formulation, is constituded of just four parts:

1. Input Unit
2. Central Processing Unit
3. Memory Unit
4. Output Unit

These four "components" are often represented as such:

Listing 1: A Von Neumann Machine.



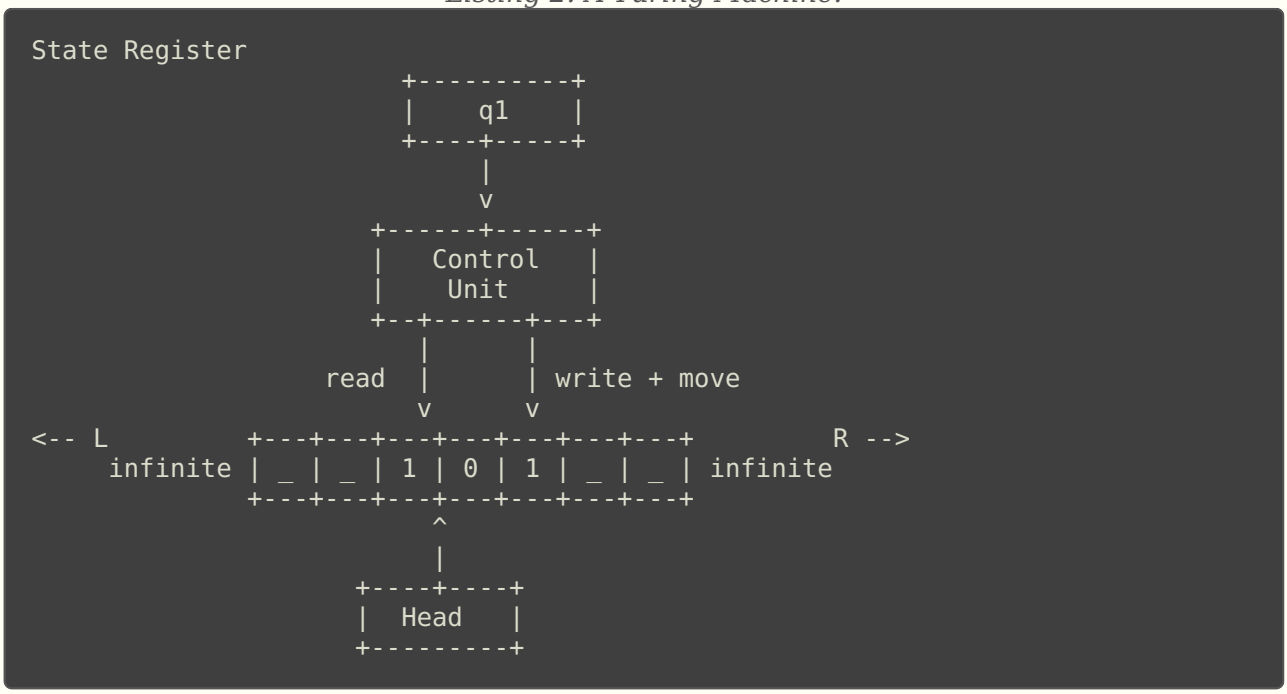
(1) Takes input from the user, without which we would have no way of interacting with the machine. (2) The "brain" of the machine that decodes instructions and carries them out. (3) Provides temoporary storage for larger computations. (4) Displays output to the user, without which we would have no way to get our results.

Real world machines are often more complicated than Listing 1, but more often than not they can still be reduced to the same abstraction from a programmer's point of view[1].

What's Special About the Von Neumann Model?

Von Neumann developed his famous abstraction with inspiration derived from another titan of computer science, Alan Turing. Turing in his watershed paper, *On Computable Numbers, with an Application to the Entscheidungsproblem*, gave the outline for a way to model computation in a pure, formal system. In it he states that "it is possible to invent a single machine which can be used to compute any computable sequence"—this powerful, abstract machine would later be named a Turing Machine. A diagram is shown below:

Listing 2: A Turing Machine.



The machine is quite simple, it takes input from tape and writes output to that tape. Its instruction set, i.e. it's *language*, dictates how the head will move around the tape. Some instructions move it to the left, others to the right; some instructions skip spaces, others write to spaces; etc. Turing's great insight was that this abstract machine could represent *every conceivable computation a human being could think up*. There is not a single actionable mathematical computation that can't be represented by a Turing Machine. Taken one step further, a Turing Machine can be constructed which takes as its input other Turing Machines. This "super" Turing Machine is called a Universal Turing Machine and it was what Von Neumann based his computer design on. There are theoretical limitations of the Turing Machine, but that is outside the scope of this class[2].

The uniqueness of the Turing Machine is that the tape does not distinguish between *instructions* and *data*, i.e. instructions can be manipulated *like anything else*.

How Do We Get the Instructions?

It should be clear now that in order to program a computer we must provide it with instructions. Here's an example you saw in the previous lecture:

Listing 3: A simple "Hello World" Java program.

```
1:  public class HelloWorld{
2:    public static void main(String args[]){
3:      System.out.println("Hello, World!");
4:    }
5:  }
```

Listing 3 is a small program that prints `"Hello, World!"` out to the screen. But *how* does it print out to the screen? How does this strange, sentence-like structure turn into something the computer can "understand"? This is something we'll learn later in the course.

So, What is a Program?

A program is nothing more than a set of instructions which tells a machine what to do. They can be as complicated as Linux or as simple as a single HALT instruction. By the end of this course you'll be able to write Java programs that fall somewhere in-between those two extremes.

Exercises

1. Think up more reasons one might program. Does it fall into one or more of the given categories?
2. Assume we have a Turing Machine with tape that has letters of the English alphabet as shown in Listing 4, further assume the head will point at *A* to start. Each time the head moves it will print the character it lands on. Write down the instructions needed to print out your first name.

Listing 4: Alphabet Tape.

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

1. Think about a computer program you use every day. While doing so, try to think about all of the steps that might be involved in getting that program to *actually run* on real hardware. (Try not to give yourself a migrane…)

Footnotes:

1 This, of course, varies depending on the sorts of programming one engages in. A software engineer dealing with high-minded business logic can think the computer "looks like" a few floating rectangles, but a computer engineer working on an embedded device will need an entirely different mental model. For this class, we'll stick to the high-minded *abstraction* provided by the Von Neumann machine. At least for now…

2 One such impasse being the "Halting Problem". It is impossible to construct a Universal Turing Machine capable of knowing whether or not an incoming Turing Machine is going to "halt" (stop processing instructions).

UNIX

What is UNIX?

UNIX is an operating system (OS), or really a collection of OSs, which was originally developed at Bell Labs by in the Late 1960s by Ken Thompson and Dennis Ritchie[1]. UNIX was developed to be a successor to MULTICS which was the first time-sharing OS, also created by Thompson and Ritchie. UNIX caught fire because it was the first[2] portable OS written—this was because it was written entirely in *C*.

Note: Dennis Ritchie's name might sound familiar. This is because he is the Ritchie from Kernighan and Ritchie's *C Programming Language* book!

What is an Operating System?

In order to understand what UNIX is, we first have to understand what an Operating System is in the first place. Simply put, an operating system is a *special program* that operates as a sort of middle man between the user and the hardware. Prior to the development of operating systems, users would interact with computers by submitting programs for execution *one-at-a-time*. If Alice needed neutron diffusion calculations and Bob needed to simulate blast wave propagation, each would have to write a separate program and wait in line to submit it. Once Alice's program finished, Bob's program could run. Since computers prior to the explosion of integrated circuits were expensive[3] users would have to contend with one another for access time. This creates an obvious problem: the owner of a computer needs to have multiple technicians on staff dedicated to handling program submission requests.

The first "operating systems" were little more than hoppers. A technician would take a stack of punchcards representing a program from a user and place it on the queue. Once a previous program finished calculating, the next would be taken off the queue. This too presents problems like: what happens when the program crashes? Or, what happens if the program stalls? Eventually, programmers decided to come up with programs that could solve these logistical issues. And thus the Operating System was born.

Modern OSs provide more than just the ability to queue up a few programs. They supply powerful abstractions that developers are constantly making use of, whether they recognize it or not. When we start programming in Java we'll find we have to do very little besides write in Java to get something runnable. This is unfathomable without the intervention of an OS.

What is the UNIX Philosophy?

The UNIX philosophy was outlined by Brian Kernighan in 1984's *The UNIX Programming Environment*, "The power of a system comes more from the relationships among programs than from the programs themselves". What he meant was that UNIX provides many useful abstractions to allow for communication between multiple processes. Instead of one super program which is in charge of many different aspects of the system, UNIX is structured in such a way to allow for many specialized programs to do simple jobs well. Later in this lecture we'll see some practical examples showcasing this behaviour.

Is Linux UNIX?

Most people no longer directly use UNIX like they did fifty years ago. Instead, we use OSs *heavily inspired by UNIX*. In this course we'll be using Linux. Linux itself isn't a singular OS, it instead refers to a family of OSs which more-or-less share a kernel (key subroutines in an OS). There are many different distros (or distrobutions) of Linux: Ubuntu, Debian, Red Hat, Arch, etc. Each distribution has its own flavour and goals, but all are *fundamentally Linux*.

Note: The name Linux is a portmanteau of the first name of its creator, Linus Torvalds, and UNIX. Linux + UNIX = Linux.

At Binghamton, we use Linux for a variety of reasons, but the most important is that it is open-source software. This means that anyone can take a look at the Linux source code. This makes it perfect for educational purposes! There's no guesswork when it comes to understanding how Linux works, we can simply take a tour through the actual code itself.

Using a UNIX-like Machine

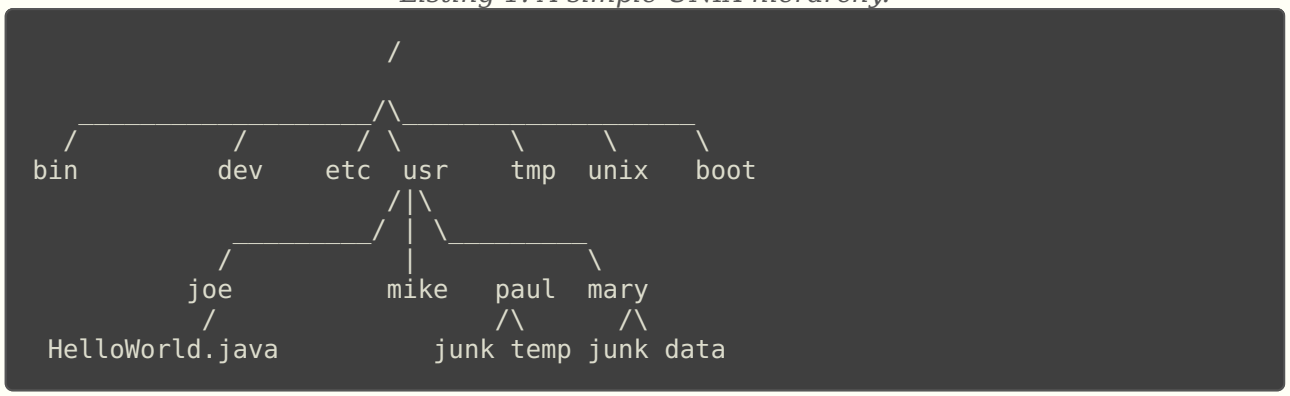
Most students who take this course have never used Linux before and have also never used the command line. The purpose of this portion of the lecture, and the first lab, is to familiarize you with some terminology and get you a bit more used to interacting with a UNIX-like system.

Note: Keep in mind, getting used to using the command line is not easy. *You will have to work at it the whole semester.* Take my word for it: it's worth the effort!

The Filesystem

Before we start looking at how to interact with data on Linux, we need to understand a little bit about the filesystem. A *filesystem* is an abstraction provided by the OS which organizes data stored in memory (often in secondary storage). This is not an OS course, so there's no need to get into *how exactly that works* here—we're just going to take for granted that it does.

Listing 1: A simple UNIX hierarchy.



In Linux, all data is stored as a *file*. A file is a sequence of bytes. Files will contain *meta-data*, or information about the file itself (size, location, permissions, etc.). Files are arranged in a hierarchy as shown by Listing 1. There are special files in the filesystem called *directories* which are files that contain other files. In Listing 1, `/`, `usr`, and `joe` are all directories and `HelloWorld.java` is a regular file.

The Shell

The *shell* is a program whose key purpose is to provide the user with direct access to the underlying system. There are many different possible shell programs that can be loaded onto a Linux/UNIX-like machine. You may or may not be familiar with `sh`, `bash`, `zsh`, `fish`, `pwsh`, etc. All of which have their own pros and cons associated with them. For this class we will be focused on using `bash` terminals[4].

A shell is a *command interpreter*. This means that when you're interacting with a shell program you are actually *programming*. Shell programs have their own languages that they use to understand the commands given to them by their users.

Note: Opening up a terminal will be different for different OSs. If you're on a Mac device, you can follow along by searching for the "Terminal" application. Most Linux distros

When you open up a shell it will look something like the following:

Listing 2: Freshly opened shell.

```
josephraskind@stargazer:~$
```

`josephraskind` is the user, `stargazer` is the name of the machine, and `~` is the shell's current working directory. The `$` indicates the beginning of the prompt, or the command that you'll be sending in. Try typing `ls` into the shell and hitting `ENTER`. Here's what it looks like for me:

Listing 3: Using ls.

```
josephraskind@stargazer:/tmp/example$ ls
homer
```

In Listing 6, I am in the subdirectory `example/` of the `tmp/` directory, which itself is a subdirectory of the root directory, `/`. What I see are the files contained within `/tmp/example/` which happens to be a directory called `homer`. You can probably guess what `ls` is supposed to do, but you can doublecheck your guess by using the command `man ls`. When I do so my terminal changes to show the following output:

Listing 4: Using man.

```
LS(1)                                                 User
Commands                                                LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information about the FILEs (the current directory by default).
Sort entries alphabetically if
       none of -cftuvSUX nor --sort is specified.

       Mandatory arguments to long options are mandatory for short options too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..
...
```

Note: You can exit `man` with the "q" key.

`man` shows the man page, or manual entry, for commands/programs installed on the machine. We can see here that `ls` is a program that lists information about files! You'll also notice that the man page lists a whole bunch of "options" that can be used to modify the behaviour of `ls`. Try out `ls -a` and see what it prints out. Here's mine:

Listing 5: Using ls -a.

```
josephraskind@stargazer:/tmp/example$ ls -a
.  ..  homer
```

You'll notice that `ls` now prints out two additional files: `.` and `..`. Believe it or not, those are also directories. Every single directory in Linux has those two directories contained within it. They represent connections to the current directory, `.`, and the parent directory, `..`. Watch what happens when I use `ls ../`:

Listing 6: Using ls.

```
josephraskind@stargazer:/tmp/example$ ls ../
example
```

We can change directories using the `cd` command:

Listing 7: Using cd.

```
josephraskind@stargazer:/tmp/example$ cd homer
josephraskind@stargazer:/tmp/example/homer$
```

My working directory has now changed from `/tmp/example/` to `/tmp/example/homer`. If I `ls` now I should see different files:

Listing 8: Using ls in homer/.

```
josephraskind@stargazer:/tmp/example/homer$ ls
iliad.txt  odyssey.txt
```

Note: `cd` is a *built-in command* whereas `ls` is a *program*. When `ls` is invoked the program must be turned into a *process* by the OS and loaded onto the CPU.

If I want to peak inside of the `iliad.txt` file I can use the `head` program:

Listing 9: Using head.

```
josephraskind@stargazer:/tmp/example/homer$ head iliad.txt
The Project Gutenberg eBook of The Iliad

This eBook is for the use of anyone anywhere in the United States and
most other parts of the world at no cost and with almost no restrictions
whatsoever. You may copy it, give it away or re-use it under the terms
of the Project Gutenberg License included with this eBook or online
at www.gutenberg.org. If you are not located in the United States,
you will have to check the laws of the country where you are located
before using this eBook.
```

Additional Functionality

Linux adheres to the UNIX philosophy by providing communication protocols that can be used to send information from one program to another. These come in two forms on the command line: *pipes* and *redirects*. Pipes *pipe* information from one program to another. Syntatically, we must place a `|` between the first in second program to indicate the direction of the dataflow. Here's an example where I count all of the lines in the *Iliad* where the word "water" is used:

Listing 10: Using a pipe.

```
josephraskind@stargazer:/tmp/example/homer$ grep -ie "water[ .,-;]" ./
iliad.txt | wc -l
12
```

First I use the `grep` program to look for instances of "water" then I take the output of that program and send it to `wc` which counts the number of lines in the text it takes as input!

Redirects can be used to send the output of a program that would normally be printed to the screen to go to a file instead. Here's an example where I use `echo` to print out a phrase to a new file:

Listing 11: Using a redirect.

```
josephraskind@stargazer:/tmp/example/homer$ ls
iliad.txt  odyssey.txt
josephraskind@stargazer:/tmp/example/homer$ echo "The Simpsons" > simpson.txt
josephraskind@stargazer:/tmp/example/homer$ ls
iliad.txt  odyssey.txt  simpson.txt
```

`echo` normally prints out to the screen, but the redirect tells the bash shell to make sure it goes to the new file simpson.txt instead.

Exercises

1. Using the `man` command, look up the `pwd` command. What does it do? Run it and describe the output.
2. Navigate to your home directory using `cd` and use `ls -a` to list all files including hidden ones. How many hidden files (files starting with .) do you see? What do you think they might be used for?
3. Create a directory structure that mirrors the filesystem diagram from Listing 1 — specifically the `usr` subtree with at least two user directories beneath it. Use `cd` and `ls` to verify your work.
4. Use `head` and `wc -l` together in a pipe to count the number of lines that `head` prints by default. What is that number?
5. The `cat` command prints the entire contents of a file to the screen. Use `echo` and a redirect (>) to create a file called `hello.txt` containing the text `"Hello, UNIX!"`. Then use `cat` to verify its contents.

6. What is the difference between `/usr` and `./usr`? Think about what `/` and `.` represent before answering.

Footnotes:

1 There were more, but Thompson and Ritchie often get the lion's share of the credit.

2 This is not strictly true, but it was the first portable OS to be shared around more than a local lab group.

3 For example, the ENIAC cost about $8.9 million in today's money (2026).

4 There are two reasons for this. (1) `bash` is what is installed on the remote servers. (2) `bash` is the terminal scripting language I am most familiar with. It also happens to be incredibly popular, so you don't lose anything learning a little bit about it.

Compilation & Execution

Introduction

In a previous chapter, we discussed a little bit about computer architecture. We learned that we can think of a computer as a machine which takes input, performs calculations based on that input, and returns output. What we're primarily interested in in this class is how we can write the "input" that the computer will use to perform calculations.

What is Machine Code?

Computers come in many different shapes and sizes, but, at base, they all follow the exact same design philosophy: there will be some internal unit which will read instructions to perform actions. In the case of a modern personal computer that philosophy is concretized by the Central Processing Unit (CPU). A familiar refrain is used by both the lowliest hacks and the holiest hackers in computer science: "*The CPU is the brain of the computer*". What they mean is that the CPU "*drives*" all of the components connected to the motherboard. All peripherals "listen" to what the CPU has to "say". But the CPU itself is "listening". What it's "listening" to are blocks of binary digits called words[1].

A typical, simplified CPU loop is as follows:

1. Fetch a binary word representing an instruction[2]
2. Decode that word to determine the desired operation
3. Execute the operation

This loop is performed for as long as the CPU is powered on.

All of these steps use a language tailored to the particular components of the given computer called *machine code*. This sort of code refers to the binary-to-instruction mappings provided by the instruction set manual of the target CPU. Below is a snapshot of the instruction set, in octal, that came with the CDC160A (the first microcomputer[3]):

OPCODE	F	E	G	Description
NOP	00	0X	----	No operation
HLT	77	00	----	Terminates
ADN	06	XX	----	Add XX to A
LDN	04	XX	----	Load XX into A
STM	41	00	YYYY	Store A into (i)YYYY

Where **F** is the Function Opcode, **E** is the "Execution Address", **G** is the second instruction word, **A** is the accumulator register, **X** is an octal operand, and **Y** is an octal address. So, if I wanted to perform the operation `x = 2 + 1` I could write the following:

Listing 1: CDC160A simple addition

```
04 02       # Load 2 into A
06 01       # Add 1 to A
41 00 0001  # Store A in memory (x)
```

In an unformatted binary blob it would look like:

Listing 2: CDC160A blob

```
000100000001000110000010100001000000000000000001
```

which is closer to what the machine will "see". It is very difficult for human beings to parse the latter example which is why practically no one writes machine code directly when programming[4]. Many people, however, write in what's called an *assembly language*.

What is an Assembly Language?

Assembly languages are one rung up the ladder of abstraction relative to machine code. They provide readable, symbolic abstractions for the pure binary strings directly processed by the CPU. Below is a snapshot of x86-64 assembly code:

Operation	Op1	Op2	Description
mov	src	dst	dst = src
add	src	dst	dst = dst + src

If I wanted to repeat the code from Listing 1 I could write:

Listing 3: x86-64 simple addition

```
1:  mov $2, %rdi  # Move constant 2 into register rdi
2:  add $1, %rdi  # Add 1 to register rdi
3:  mov %rdi, 0x1 # Store value of rdi into mem address 1 (x)
```

Here, instead of octal or binary values, I have symbols representing familiar words ("mov" for "move", "add" for addition, etc.). If I store the above code in a file called "simple_add.s", I can use an assembler to turn it into a binary file:

Listing 4: Using an assembler

```
1:  as simple_add.s -o simple_add.o
2:  objcopy -O binary --only-section=.text simple_add.o simple_add.bin
```

The first line invokes the GNU assembler, `as`, which turns the assembly code into an ELF file; the second line invokes objcopy which copies the contents of an object file to pure binary (the .text section is where code is stored in compiled files). If we try to directly `cat` out the **.bin** file we'll see a strange pseudo-binary string. This is because `cat` looks for UTF-8 encoded characters to print out to the screen, not unencoded binary strings! We can properly view the binary contents of the file with:

Listing 5: Hexdump command

```
xxd -b -c 1 simple_add.bin | awk '{printf("%s",$2)} END{print ""}'
```

Which should spit out the following:

Listing 6: Binary output

```
010010001100011111000111000000100000000000000000000000000010010001000001111000110000000101001000100010010011110000100101000000100000000000000000000000000
```

Wow! Over a hundred bits for only 3 short lines of assembly! Again, this should show you why it's not easy to program in machine code directly…but, even assembly looks a bit tough on the eyes. Both Listings 1 and 3 write more than one line of code to represent the logic of `x = 2 + 1`. The process that translates the `x = 2 + 1` to the machine code seen above is called *compilation*.

What is Compilation?

Put simply, compilation is the translation of one computer language to another (Language A → Language B). Compilation is a rich and complex topic in computer science, and therefore not one we can spend much time on in class; however, it is important that you are aware of the basic steps of compilation needed to turn Java source code into a runnable executable. Compilation begins with the *compiler*.

What is a Compiler?

A compiler is a program which translates computer code from one language to another. All compilers roughly follow the below path[5]:

Listing 7: Compilation Stages


position = initial + rate * 60
Lexical Analyzer
<id,1> <=> <id,2> <+> <id,3> <*> <60>
Syntax Analyzer
=
<id,1> +
<id,2> *
<id,3> 60
Semantic Analyzer
=
<id,1> +
<id,2> *
<id,3> inttofloat
60
Intermediate Code Generator
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
Code Optimizer
t1 = id3 * 60.0
id1 = id2 + t1
Code Generator

```
                |
                v
    LDF   R2, id3
    MULF  R2, R2, #60.0
    LDF   R1, id2
    ADDF  R1, R1, R2
    STF   id1, R1
```

For this class we're mainly interested in the first three stages: lexical analysis, syntactical analysis, and semantic analysis. These represent three facets of a programming languge: its symbol set, structure, and rules. In other words, what "words" we can use, how we arrange those "words", and what those arrangements "mean". In the programming language world we group the first two under "syntax" and the one remaining under "semantics"—in reasonably complicated programming languages, like Java, neither can be totally separated from one another.

Let's use an example in a language we all know very well: "The dog flew quite orangely this Mars." Each word exists within the symbol-set of the English language, i.e., they can all be found in a dictionary. Additionally, we can analyze the *structure* of the sentence:

- "The dog"
  - noun phrase acting as subject. "The" is a definite article modifying the noun "dog."
- "flew"
  - simple past tense intransitive verb, the predicate.
- "quite orangely"
  - adverb phrase acting as adverbial modifier of "flew." "Quite" is an adverb of degree modifying the manner adverb "orangely."
- "this Mars"
  - noun phrase acting as adverbial of time/place, modifying "flew." "This" is a demonstrative determiner modifying "Mars."

The sentence is well-formed, but it is completely devoid of *semantic meaning*—it's complete nonsense![6]

Much the same, I can construct lines of Java code which appear to be correct, but have no actual place in the language: `"hello world" < 1`. *Syntactically*, I have placed two operands on either side of an infix operator, but *semantically* I've tried to compaire the number one to a constant that represents a string of characters. Compilers exist to catch errors of this nature.

javac

As this is a Java course we will be using a Java compiler. There are many different Java compilers available to use, but we choose `javac` or the Java Development Kit (JDK) **java c**ompiler due to its popularity and wide availability on Linux platforms.

Listing 8: "hello world" program

```
1:  public class HelloWorld{
2:      public static void main(String args[]){
3:          System.out.println("Hello, World!");
4:      }
5:  }
```

For our purposes, using `javac` is pretty easy[7]. If we have a Java program, say `HelloWorld.java` (from Listing 8), we can compile it into a class file with the following command: `javac HelloWorld.java`. `javac` will automatically create a class file called `HelloWorld.class` which can be run by specifying the path to it in the shell:

Listing 9: Compilation and execution of HelloWorld.java

```
josephraskind@stargazer:/tmp/hello$ javac HelloWorld.java
josephraskind@stargazer:/tmp/hello$ ls
HelloWorld.class  HelloWorld.java
josephraskind@stargazer:/tmp/hello$ java HelloWorld
Hello, World!
```

In the blink of an eye `javac` went through every stage detailed in Listing 7! Each step in the process could be studied for weeks. We'll have to make do with occasionally referencing them when they're relevant.

Note: Unlike `gcc` with C files, `javac` does not create binary executables. `javac` creates .class files which contain Java Virtual Machine (JVM) bytecode which can be interpreted by a JVM program such as `java` which calls the Open JDK HotSpot JVM. We will learn more about this in the chapter on JVMs.

What are Compilation Tools?

Compilation can be rather tricky to wrap one's head around, and it can get really overwhelming when you are working on a project that spans not just two files, but two hundred or two thousand files![8] This is where getting familiar with *compilation tools* can really pay off.

Make

Make is a popular compilation tool for coding projects in Linux environments. It is yet another GNU tool which is often pre-installed in most Linux distros. Make has its own language, albeit one much more limited in scope than Java, whose programs are stored in *makefiles*. The structure of a makefile looks a bit like this:

Listing 10: Makefile syntax

```
1:   VARIABLE = lorum
2:   target: dependencies
3:         commands
```

Let's assume I still have my file `HelloWorld.java`. I can write a makefile to compile my program as follows:

Listing 11: "hello world" makefile

```
1:   HelloWorld.class: HelloWorld.java
2:           javac HelloWorld.java
```

I define the **rule** "hello world" (before the ":") which is dependent on the file `HelloWorld.java`. When the rule is triggered, make checks to see if `HelloWorld.java` needs to be compiled, i.e. "has it changed since the last compilation attempt?", then performs the set of commands defined beneath the rule. As long as there is a file named `makefile` or `Makefile` in my working directory I can invoke the file with a simple call to the `make` executable:

Listing 12: Calling make

```
josephraskind@stargazer:/tmp/hello$ make
javac HelloWorld.java
josephraskind@stargazer:/tmp/hello$ java HelloWorld
Hello, World!
josephraskind@stargazer:/tmp/hello$ make
make: 'HelloWorld.class is up to date.
```

Note: You'll notice that a second call to make did not call `javac`. This is significant for projects which have hundreds or thousands of Java files which need to be compiled prior to producing a runnable program.

We can also add extra rules which perform various non-compilation tasks:

Listing 13: "hello world" makefile with clean

```
1:   HelloWorld.class: HelloWorld.java
2:       javac HelloWorld.java
3:   clean:
4:       rm HelloWorld.class
```

This makefile has a new "dummy" rule which will remove the executable from the working directory:

Listing 14: Calling make clean

```
josephraskind@stargazer:/tmp/hello$ ls
HelloWorld.class  HelloWorld.java  makefile
josephraskind@stargazer:/tmp/hello$ make clean
rm HelloWorld.class
josephraskind@stargazer:/tmp/hello$ ls
HelloWorld.java  makefile
```

Note: `make` always calls the first rule by default!

We can use variables to further abstract our rules:

Listing 15: "hello world" makefile with variables

```
1:   FILE = HelloWorld
2:   FLAGS = -Xlint
3:   CC = javac
4:   $(FILE).class: $(FILE).java
5:       $(CC) $(FLAGS) $(FILE).java
6:   clean:
7:       rm $(FILE).class
```

Now, instead of rewriting a makefile for every new single-shot file I write, I can just replace the "HelloWorld" at the top.

Note: Be careful when copy-pasting makefile rules, `make` expects a TAB before every command!

If you do enough programming in Linux environments, you **will** encounter makefiles. It's best to get familiar with them now in a low(er) stakes environment.

Note: There are "build tools" other than make that are used in production. Programs such as `maven`, `gradle`, and `bazel` are popular in the Java ecosystem. I decided not to use them for this class in an effor to cut down on complexity. They can be a pain to use and often require serious study in their own right!

Exercises

1. Using `javac`, compile the hello world program from Listing 8 and run it. Then remove the quotations and see what happens. Reason about the output.
2. Write a makefile for the hello world program that includes three rules: one to compile the program, one `clean` rule to remove the executable, and one `run` rule that compiles and immediately runs it. What happens when you call `make run`?

3. Using the CDC160A instruction set from the lecture, write the machine code (in octal) to perform the operation `x = 5 + 3`. What would the equivalent x86-64 assembly look like?
4. Assemble the x86-64 addition code from Listing 3 using `as` and `objcopy`. Then use the `xxd` command from Listing 6 to dump the raw binary. How many bits does the output contain? Does this match what you'd expect?
5. Take the hello world makefile from Exercise 2 and convert it to use variables for the filename, compiler, and flags as shown in Listing 15. Then use it to compile a second program of your choice by only changing the variable at the top.
6. The lecture states that `cat` cannot properly display a `.bin` file. Try it yourself, run `cat simple_add.bin` after assembling. Describe what you see and explain why it looks that way.
7. The lecture mentions that `javac` compiles Java source code into bytecode rather than native machine code. Try inspecting this process step by step: first compile `HelloWorld.java` with `javac` to produce a `.class` file, then use `javap -c HelloWorld` to disassemble the bytecode. What do you notice about the instructions? How do they differ from the x86-64 assembly produced by `gcc`? What does this tell you about how Java achieves platform independence?
8. A makefile rule only recompiles when its dependency has changed. Compile `HelloWorld.class` with `make`, then run `make` again without changing anything. Now `touch HelloWorld.java` (which updates its timestamp without changing its contents) and run `make` again. What happens and why?

Footnotes:

1 Word size varies depending on architecture. For this class, we focus on x86-64 which has a word size of 64 bits. Instructions are not always word length.

2 Instructions will come from a component called the Instruction Cache, or I-Cache.

3 *A History of Modern Computing*, Paul E. Ceruzzi.

4 There are exceptions, but this is true in the broad strokes.

5 Diagram transcribed from *Compilers: Principles, Techniques, and Tools* by Aho et al.

6 Noam Chomsky famously used the sentence "Colorless green ideas sleep furiously" in *Syntactic Structures*.

7 You will likely not feel like that at first. That's okay! The more you practice, the more it will start to make sense.

8 In this course we do not go beyond more than a few files in one project.

Data

Introduction

> “
>
> *The world is the totality of facts, not of things — Ludwig Wittgenstein*

In order to perform computations we must have something to perform computations on. After all, what good is addition if there is nothing to add? In computing, we call these computational necessities "data". Conceptually, data is an information abstraction. This abstraction has analogues to phenomena in the real world—meaning data *represents something*. For example, I may have data in the form of numbers which represent the dimmensions of a shipping container, or I may have data in the form of words which represent the list of students who made the Dean's list, or I may have data in the form of a graph which represents related artists on a music platform, etc.

What interests us most in this class is not what data is philosophically, but rather how it is stored within a computer.

How is Data Stored in a Computer?

Fundamentally, data can be stored in any way a computer architect sees fit. The earliest computers stored numbers as *decimals* within their circuitry. For example, the ENIAC used what was called Binary Coded-Decimal (BCD) wherein 10 binary digits (bits) were used to represent a single value, `0000000001` for 0, `0000000010` for 1, ..., and `1000000000` for 9. It wasn't until the EDVAC that it became standard for computers to store numeric information purely in binary (`0` for 0, `1` for 1,..., and `1001` for 9). As late as 1968, of the 139 available general purpose digital computers, "46 were decimal, 72 binary, 4 octal, 15 binary/decimal, 2 biquinary"[1].

The switch to pure binary came from a desire to cut down on the amount of space taken up by numerical data. In the famous *Report on EDVAC*, John Von Neumann writes,

> "*If one contemplates using a decimal system with either the Iconoscope or delay line memory one is forced into a binary coding of the decimal system—each decimal digit being represented by at least a tetrad of binary digits. Thus an accuracy of ten decimal digits requires at least 40 binary digits. In a true binary representation of numbers, however, about 33 digits suffice to achieve a precision of 10^10.*"

We'll talk more about memory later in the semester, but right now it will suffice to say that there must be something to "hold" our bits when we run out of register space—that is what we call *memory*.

In this lecture we'll answer two questions: "How are numbers stored in a computer?" and "How is text stored in a computer?".

How are Numbers Stored in a Computer?

In order to answer this, we first have to define what we mean by "numbers". In an effort to avoid any sort of philosophical detour, we'll assume a "common sense" definition of "number": numbers are objects used to measure phenomena. A human being has two eyes, one nose, ten fingers, etc. We may also want to distinguish between simple, countable numbers and infintesimal, continuous numbers. In C, these kinds of numbers are distinguished by having different *types* (we will talk about this extensively in the future). `int` matches to integers, or countable numbers, and `float` matches to reals, or continuous numbers.

Integers

When I write the Java code `x = 17`, what is *actually* being stored inside the computer? The answer is that there will exist a set of binary digits which will be arranged to represent the idea of the number seventeen within the machine. A trip to the Java specification will tell us that,

> *For int, from -2147483648 to 2147483647, inclusive (Java Language Specification - Java SE 26 Edition, § 4.2.1)*

This likely raises another question. Why this range in particular?

Taking the log_2 of 2147483648 yields 31 (2^31 = 2,147,483,648) which means it takes 31 bits to encode that maximum number. From what you already know about computers it might seem odd to default to 31 bits for an integer. Why not 32? The reality is that an `int` *does* default to 32 bits due to the use of *two's compliment*. In two's compliment the most significant bit is a *signed bit*, meaning its presence/absence indicates a negative/positive value. For example,

- 0000 0000 0000 0001 = 1
- 1111 1111 1111 1111 = -1

It can be represented by the formula:

- (d^n-1 * -2^n-1) + (d^n-2 * 2^n-2) +...(d^1 * 2^1) + (d^0 * 2^0)

where **n** is the number of bits.

This means that 17 is stored as `0000 0000 0001 0001`.

One reason for this is that the Java language is meant to ultimately be run in a Java Virtual Machine (JVM) and the official JVM specification states that:

> *int, whose values are **32-bit signed two's-complement integers**, and whose default value is zero*

Reals

How about `x = 17.25`? Let's take another look at the spec,

> *"The floating-point types are float and double, which are conceptually associated with the 32-bit binary32 and 64-bit binary64 floating-point formats for IEEE 754 values and operations, as specified in the IEEE 754 Standard" (Java Language Specification - Java SE 26 Edition, § 4.2.3)*

This means that a `float` follows the IEEE 754 standard. It follows the following formula:

- (-1)^s + (1 + m) * 2^e
- Exponent bias = 127
- Exponent = e + bias
- m = fraction bits (start at 2^-1)

Listing 1: 17.25 in IEEE 754 Standard

```
  IEEE 754 Floating Point Specification

sign   exponent (8 bits)                          fraction (23 bits)
 |   |                    |                                          |
|                                                                     |
 v   v                    v                                          v
v                                                                     v
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0| 1| 0| 0| 0| 0| 0| 1| 1| 0| 0| 0| 1| 0| 1| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 31  30                   23 22                        (bit
index)                                 0
```

First we see if the number is positive or negative, clearly it's positive so the signed bit is set to 0. Next, we need to represent 17.25 as a true binary number. We already know the non-fractional half is 10001. The fractional half is easy to get, all we need to do is follow the algorithm:

1. Set f equal to the fractional half.
2. Set f equal to f * 2.
3. If f >= 1 record a 1; if not, record a 0.
4. Repeat 1 until pattern emerges or we run out of bits.

.25 is rather simple, we multiply by 2 to get .5 (and tally a 0), then multiply by 2 to get 1 (and tally a 1). Since the remaining fractional half is 0, there's no further need to multiply. This gives us .01 in binary. Combine that with our 17 and that gives us `10001.01`.

Next we convert `10001.01` to scientific notation, `1.000101 * 2^4`. However many spaces we moved the decimal up is what we add to the bias. This gives us 127 - (-4) = 131 which is equivalent to 100000011 in binary. We then take the remaining bits after the decimal and store that in the fractional portion. Which gives us the IEEE 754 32-bit floating point number shown in Listing 1.

How is Text Stored in a Computer?

Text is where bit-based data representation and storage becomes much more abstract. There is an obvious direct mapping between representing the number 17 as decimal number and 10001 as binary number. It becomes much less obvious when trying to map 97 to 'a'.

One traditional way of representing characters is using ASCII encoding:

Char	Decimal	Octal	Hex	Binary
A	65	101	41	01000001
B	66	102	42	01000010
C	67	103	43	01000011
...	...	...	...	...
Z	90	132	5A	01011010

ASCII encoding using a single byte (really 7 bits, 0 to 127) to encode all of the english alphabet, including uppercase and lowercase letters, common punctuation, symbols and digits, alongside specialty characters like the null character '\0' and the new line character '\n'. For example, the character string "101" would be encoded as the binary string `001100010011000000110001` ("0" is 0x30 and "1" is 0x31).

ASCII encoding works well for the English language, but begins to fall apart the moment more than 128 characters are needed. For example, languages like Japanese or Chinese need far more symbols than the basic 26 required for English works (52 if counting uppercase letters). The most popular solution nowadays is to adopt newer encoding standards like UTF-8 or UTF-16 encoding.

UTF-8

UTF-8 is a character encoding standard based on the unicode standard. It is by far the most popular text encoding and has the ability to encode 974,530 unique symbols[2] (in reality, only a small fraction of that potential is realized). The encoding rules are simple:

First Code Point	Last Code Point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0zzzzzzz			
U+0080	U+07FF	110xxxyy	10yyzzzz		
U+0800	U+FFFF	1110wwww	10xxxxyy	10yyzzzz	
U+010000	U+10FFFF	11110uvv	10vvwwww	10xxxxyy	10yyzzzz

where the characters u to z, each representing a hexadecimal digit, are replaced by their constituent four bits uuuu to zzzz, from the positions U+uvwxyz.

Say I wanted to encode the thumbs up emoji (👍), I would first need to look up its UTF-8 value. Using the link here, we can see the code is U+1F44D. That places the emoji squarely in the last row of the above table. First I need to convert the code to binary U+1F44D → `0 0001 1111 0100 0100 1101` such that it matches with `u vvvv wwww xxxx yyyy zzzz`. Next we need to match the letters with their bits in the template: `11110000 10011111 10010001 10001101`. In order to verify that this in C we should convert the binary into hexadecimal: `0xF0 0x9F 0x91 0x8D`. The following C program can be used to double check our work:

Listing 2: UTF-8 Checker

```
1:   public class ThumbsUp {
2:       public static void main(String[] args) {
3:           byte[] thumbs_up = {(byte)0xf0, (byte)0x9f, (byte)0x91,
(byte)0x8d};
4:           String s = new String(thumbs_up,
java.nio.charset.StandardCharsets.UTF_8);
5:           System.out.println(s);
6:       }
7:   }
```

Compiling and running it we get:

Listing 3: Thumbs up!

```
josephraskind@stargazer:/tmp/data$ javac ThumbsUp.java
josephraskind@stargazer:/tmp/data$ java ThumbsUp
👍
```

Note: This only works if your terminal is UTF-8 encoded. This is almost always the case for Linux devices, but is not guaranteed for Windows systems (which tend to favor UTF-16 for historical reasons).

Hopefully, this gave you a bit more insight into what's going on behind the scenes when your using your computer!

Exercises

1. Write a Java program that stores 2147483647 in a variable of type `int`, adds 1 to it, and prints it out. What do you see? Explain why. (You can print out numbers using `System.out.println(variable_name)`.)
2. Convert the number -17 to its 32-bit two's complement binary representation by hand. Verify your answer by writing a Java program that stores -17 in an `int` and prints it out using `System.out.println(x)`. Then print its binary representation using `Integer.toBinaryString(x)`. What do you see and why?
3. Convert 0.1 to IEEE 754 32-bit floating point by hand following the algorithm given in the lecture. What do you notice about the fractional part? What does this imply about storing 0.1 in a `float`? Verify by writing a Java program that prints `0.1f` using `System.out.printf("%.20f%n", 0.1f)`.
4. The string `"101"` is given in the lecture as the binary string `001100010011000000110001`. Using the ASCII table from the lecture, encode your first name as a binary string by hand. Then verify by writing a Java program that prints each character of your name alongside its decimal ASCII value using `System.out.printf("%c %d%n", c, (int)c)`.
5. Using the UTF-8 encoding table, encode the character 'é' (U+00E9) by hand. Which row of the table does it fall in? Write a Java program to verify your encoding using the same technique as Listing 2.
6. The lecture mentions that ASCII uses 7 bits (0 to 127) despite being stored in a full byte. What happens to the 8th bit? Look up "extended ASCII" and write a short explanation of how different systems took advantage of that extra bit, and why this caused compatibility problems that UTF-8 was designed to solve.

Footnotes:

1 *History of Binary And Other Nondecimal Numeration*, Anton Glaser.

2 According to Wikipedia.

Variables

Introduction

Think back to your first time encountering an algebraic formula; say something as simple as $y = x + 1$. You likely learned that **x** is an unknown value, or at least a value which isn't known the moment you glanced at the formula. **x** is a stand-in, a signifier, a symbol, a name, i.e., something that *represents* the presence of some magnitude. **x** is a *variable*. The value attached to **x** is subject to change. The potential value attached to it is external to the formula itself.

Variables in Programming Languages

Variables within programming languages satisfy a similar, although not identical, position to variables within algebraic formulas[1]. Variables are *names* for data stored in the computer. When we write `x = 1` we are saying that `x` is a name which represents the computational object `1` which exists somewhere within the hardware of the machine running the program. As the name suggests, variables can change values. For example, we can write `x = x + 1` which increments the values of `x`.

Scope

All identifiers, including variable names, will have a *scope* associated with them. Scope has two facets in C: (1) the *lexical scope*; (2) the *linkage scope*. Here, we focus on the *lexical scope* which refers to "the region of the program text within which the identifier's characteristics are understood"[2]. An example should be instructive:

Listing 1: Examples of scope

```
1:  public class Scope {
2:      // Block 0, "Global" scope
3:      static int g = 4;
4:
5:      public static void main(String[] args) {
6:          // Block 1, Outer main method
7:          int x = 1;
8:          {
9:              // Block 2, Inner main method
10:             int y = 2;
11:             {
12:                 // Block 3, Innermost main method
13:                 int z = 3;
14:             }
15:         }
16:     }
17: }
```

Listing 4 shows a Java program which defines 4 `int` type variables: **g**, **x**, **y**, **z**, each one within a different code *block*. In Java, braces, "{}", indicate the starting and closing of a code block, i.e., a collection of statements (you can think of statements as the sentences of programming languages). These blocks are the arbiter's of a variable's *lexical scope* (hereon referred to simply as "scope"). When a variable is defined in a given block, it can be "seen" by all statements/expressions within the same block as well as interior blocks. For example, **g** is visible at Block levels 0, 1, 2, *and* 3, despite it only being declared in Block 0. We call Block 0 the *global scope*—this refers to anything declared outside of a function block (such as the function `main`). Furthermore, **x** is visible at Block levels 1, 2, and 3, but *not* at Block 0. Every time a block is exited, all of the names defined therein are forgotten. To further emphasize this I will update Listing 4 so that **z** is referenced in Block 1 after it has been defined in Block 3:

Listing 2: Examples of scope

```
1:  public class Scope {
2:    // Block 0, Global scope
3:    static int g = 4;
4:
5:    public static void main(String[] args) {
6:      // Block 1, Outer main method
7:      int x = 1;
8:      {
9:        // Block 2, Inner main method
10:       int y = 2;
11:       {
12:         // Block 3, Innermost main method
13:         int z = 3;
14:       }
15:     }
16:     x = z + 1;
17:   }
18: }
```

When I try to compile this code, I get the following error:

Listing 3: Scope error

```
josephraskind@stargazer:/tmp/scope$ javac Scope.java
Scope.java:16: error: cannot find symbol
        x = z + 1;
            ^
  symbol:   variable z
  location: class Scope
1 error
```

Why does the compiler say that **z** cannot be found? I clearly define it in Block 3, no?

Again, this has to do with the way scope works in Java. The moment we exit a block, we lose the ability to reach all of the identifiers defined within that block. This means **z** effectively vanishes from the perspective of the compiler the moment we exit Block 3 (Line 12).

Scope rules also allows us to reuse old identifiers so long as we do not use the same identifier in the same scope. For example, take a look at the following code which defines **x** twice in `main`:

Listing 4: Examples of scope

```
1:   public class Dec{
2:       public static void main(String[] args){
3:           int x = 1; //Declaring x
4:           int x = 2; //Declaring x, again
5:       }
6:   }
```

If I try to compile that code:

Listing 5: Redeclaration error

```
josephraskind@stargazer:/tmp/scope$ javac Dec.java
Dec.java:4: error: variable x is already defined in method main(String[])
        int x = 2; //Declaring x, again
            ^
1 error
```

I get an error that tells me I "already defined" **x**. This happens because all identifiers within a given scope must be *unique*. I cannot have two different memory locations tied to the same name, there must be a one-to-one mapping within a scope between name and memory location at all times. If I wanted to fix the above error, I would need to define each **x** in their own, mutually exclusive scopes (I will leave that as an exercise for the reader).

Note: Scope will become relevant again when we discuss functions!

What a Variable *Really* Is in Java

In Java, variables are not purely symbolic names. They are intimately tied up with the *memory semantics* of the JVM runtime system. They are truly just standins for *locations in memory*[3].

To illustrate this I will write a simple Java program which declares two `int` variables and adds them together and stores the result in a third variable:

Listing 6: Adder

```
1:   public class Adder{
2:       public static void main(String[] args){
3:           int a = 100;
4:           int b = 200;
5:           int c = a + b;
6:
7:       }
8:   }
```

Then I will compile the program down to its JVM bytecode representation with `javac Adder.c; javap -c Adder.class`. We can see the resultant JVM bytecode file here:

Listing 7: Adder (JVM bytecode)

```
 1:   Compiled from "Adder.java"
 2:   public class Adder {
 3:     public Adder();
 4:       Code:
 5:          0: aload_0
 6:          1: invokespecial #1                  // Method java/lang/
Object."<init>":()V
 7:          4: return
 8:
 9:     public static void main(java.lang.String[]);
10:       Code:
11:          0: bipush        100
12:          2: istore_1
13:          3: sipush        200
14:          6: istore_2
15:          7: iload_1
16:          8: iload_2
17:          9: iadd
18:         10: istore_3
19:         11: return
20:   }
```

The `main` method's `Code` label shows where our program's code begins. We can see our constants clearly on Lines 11 and 13 which must mean that `istore_1` and `istore_2` are directly related to **a** and **b**, respectively. How can that be? The constants are stored inside of the JVM's local variable array. It is nothing more than a location in memory which is managed by the JVM. By the time we reach JVM runtime land, we no longer have symbolic names

like **a** and **b**, but rather purely locations in memory managed by the JVM runtime which is outlined by the compiler.

Exercises

1. Look at Listing 4 again, and edit the code such that **x** is defined twice in `main`, but the compiler does not throw any errors.
2. Compile Listing 6 with `javac Adder.java` and inspect the bytecode with `javap -c Adder.class`. Identify which lines in the bytecode correspond to each of the three variable declarations. What local variable slots are used for **a**, **b**, and **c**? Why do you think slot 0 is skipped?
3. Modify Listing 6 to add a fourth variable `int d = c * 2` and recompile. Run `javap -c` again. What new bytecode instructions appear? Which slot is **d** assigned to?
4. Write a Java program that declares a static field `static int g = 10` outside `main` and a local variable `int g = 20` inside `main`. Does it compile? Which value gets printed and why? Now try to print the static `g` from inside the block that defines the local `g` using `Classname.g`—where `Classname` is replaced with the name of your class—is it possible?
5. The lecture states that once a block is exited its variables "vanish from the perspective of the compiler." Write a Java program that demonstrates this by attempting to access a variable from an inner block after that block has closed. Record the exact compiler error you receive and explain in your own words what it means in terms of scope.
6. Earlier in the lecture, I wrote the code snippet `x = x + 1` and referred to it as "incrementing" **x**. Outside of computer science, `x = x + 1` likely looks like a nonsense statement. From what you now know about variables, how is that possible in a programming language? Write a Java program that defines an integer **x**, assigns it the value 100, and increments it by 200. Then inspect the bytecode with `javap -c` to see how the JVM represents this operation.

Footnotes:

1 All subsequent references to "variables" will be references to "variables" in the computer science sense.

2 *C Programming Language - Second Edition*, K&R.

3 This will return when we talk about references.

Types

Introduction

In a previous chapter, I wrote that *data* is an essential "information abstraction" that falls under the category of "computational necessity". This idea was further developed when we looked at how data is stored in a machine, both as constant values and as *variables*. Now, we'll explore how different kinds of data are interpreted by the Java programming language and the *rules* that govern them.

Type System

We now know that all data in a computer is stored as a collection of binary digits (bits). We also know that in Java, if I want to store numeric information I can store it in something called an `int` variable. I could declare one such variable using the statement `int x = 65`, which says that **x** is a symbolic name for a location in memory holding the bits `1000001`. But we also know that `1000001` can be interpreted as the letter 'A' in ASCII. What determines how those bits are interpreted?

In programming languages, one way of managing the interpretation of data is through the construction of a *type system*. In Benjamin Pierce's seminal book, *Types and Programming Languages*, he provides a possible definition, **"a type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."** This is a highly technical and accurate, albeit a bit bureaucratic, definition of a type system. I can give a bit more of a looser, intuitive definition,

- **A type system provides the *rules* for different kinds of data.**

This means that type systems lay out all of the ways that data can interact during a program's execution.

Nearly every high-level programming language in use today[1] has a type system. Type systems provide a variety of benefits to programmers:

- Error checking
  - Converting a string into a number before taking the square root
- Maintenance
  - Update the definition of a data type and its updated everywhere
- Enforces contracts
  - Guarantees behaviors of two variables
- Documentation
  - Easier to read code when types are involved

- Language safety
    - A safe language is one that protects its own abstractions

The Java programming language has a type system and it is important you understand how it works.

Java is *Statically* Typed

Firstly, Java is *statically* typed. This means that every single variable and function within a program must have a type assigned to it at compile-time. Here is an example of a small Java program:

Listing 1: Variable declarations

```
1:  public class Decl{
2:      public static void main(String[] args){
3:          int i = 10;
4:          float f = 10.5f;
5:          String s = "Hello, World!";
6:      }
7:  }
```

In order for the above program to compile, variables **i**, **f**, and **s** must all have *types* associated with them—this is different from dynamically typed languages like Python where the programmer does not need to explicitly specify the type at the time of declaration. **i** is declared with the type `int`, **f** with the type `float`, and **s** with the type `String`. Here's what happens if I remove the `int` from line 2 in Listing 1:

Listing 2: Removing int type

```
josephraskind@stargazer:/tmp/types$ javac Decl.java
Decl.java:3: error: cannot find symbol
        i = 10;
        ^
  symbol:   variable i
  location: class Decl
1 error
```

We can see in the error message from Listing 2 that something went wrong in the compilation process. The error message, however, looks a bit strange. It says that **i** must cannot be *found*. Again, this is because all variables in Java must be declared with a type—any reference to a variable without a type identifier behind it immediately tells the compiler that the variable is *not being declared*, but is being used.

The compiler does not just check to see that every variable/function has a type, but it also checks to make sure that variables/functions follow the rules of their declared types. For example if I alter Listing 2 to:

Listing 3: Invalid type assignment

```
1:  public class Decl{
2:      public static void main(String[] args){
3:          int i = 10;
4:          float f = "Hello, World";
5:          String s = "Hello, World!";
6:      }
7:  }
```

Note: You will have noticed that `String` looks a bit different than `int` and `float`, this is because it is a *reference* type. We will learn more about reference types later on.

When I try to compile the code I get the following message:

Listing 4: Compilation error from Listing 3

```
josephraskind@stargazer:/tmp/types$ javac Decl.java
Decl.java:4: error: incompatible types: String cannot be converted to float
        float f = "Hello, World";
                  ^
1 error
```

This happens because the type assigned to the expression on the right side of the `=` is not the same as the type assigned to the variable on the left side (and cannot be implicitly converted).

Java is *Strongly* Typed

Another example of a statically typed language is C. Although C is *statically* typed, it's often not considered *strongly* typed. This is because it is very easy to "break" the type system. For example if I slightly change the sort of alteration performed in Listing 3, to the following:

Listing 5: Valid type assignment?

```
1:  int main(){
2:    int i = "Hello, World!";
3:    float f = 10.5;
4:    const char * s = "Hello, World!";
5:  }
```

Then I compile it:

Listing 6: No compilation error from Listing 5

```
josephraskind@stargazer:/tmp/types$ gcc types.c
types.c: In function ‘main’:
types.c:2:11: warning: initialization of ‘int’ from ‘char *’ makes integer
from pointer without a cast [-Wint-conversion]
    2 |   int i = "Hello, World!";
      |           ^~~~~~~~~~~~~~~
```

I dont get an error?

Note: Technically, the above does not "break" the type system, but it does break reasonable expectations from the programmers perspective. It seems nonsensical to be able to assign an integer to the string constant "Hello, World!". In reality, that's not exactly what's happening...

Java, in contrast, is much better at preventing these sorts of wanton conversions. There are limitations, especially at the frontiers of the language, but generally the advantage that Java provides over other languages is the relative safety associated with its type system. For example, let's take a similar example to Listing 5:

Listing 7: Similar assignment in Java

```
1:  public class TypeSafety {
2:      public static void main(String[] args) {
3:          int i = "Hello, World!";  // Does this work?
4:      }
5:  }
```

And when we compile it:

Listing 8: Similar assignment in Java

```
josephraskind@stargazer:/tmp/types$ javac TypeSafety.java
TypeSafety.java:3: error: incompatible types: String cannot be converted to int
        int i = "Hello, World!";  // Does this work?
                ^
1 error
```

Unlike the C example, this Java code does not compile. Java refuses to allow it to reach the JVM execution stage. This is the practical difference between weak and strong typing: C trusts the programmer and produces a runnable, if "broken", binary, while Java enforces the contract at compile time and stops the process entirely.

Java Types

We will now go through, almost, all of the different types in Java. Starting with the basic types.

Primitive Types

Java makes a distinction between **primitive types** and **reference types**. For the time being we will be concerned only with primitive types. Later on we will discuss reference types in greater detail. The basic, primitive types are as categorized as follows (Java Language Specification § 4.2):

1. Integral Types

2. Floating-Point Types
3. Boolean Type

Integral Types

The integral types are defined as follows:

- For `byte`, from -128 to 127, inclusive
- For `short`, from -32768 to 32767, inclusive
- For `int`, from -2147483648 to 2147483647, inclusive
- For `long`, from -9223372036854775808 to 9223372036854775807, inclusive
- For `char`, from '\u0000' to '\uffff' inclusive, that is, from 0 to 65535

Floating-Point Types

The floating-point types are defined as follows:

- For `float`, 32-bit binary32 floating-point format for IEEE 754
- For `double`, 64-bit binary64 floating-point format for IEEE 754

Boolean Type

The `boolean` type simply stores the literals `true` and `false`.

Type Conversion

Sometimes mixing and matching types is *legal* in the language despite intuition pointing the opposite direction. This sort of type fluidity is called type conversion. One data type is converted to another based on a set of rules by the language specification. There are two kinds of type conversion: explicit and implicit.

Explicit Conversion

Explicit type conversion occurs when the programmer uses direct syntax to indicate the change from one type to another. Here is an example:

Listing 9: Explicit type conversion

```
1:  public class ExplicitConversion{
2:      public static void main(String[] args){
3:          double d = (double) 10;
4:      }
5:  }
```

What we see on Line 3 is an explicit type conversion from the constant 10's `int` type to a `double`. Placing the name of the desired type in parentheses prior to the expression explicitly tells the compiler to convert the `int` to a `double`—this is called a "cast". In Line 3 we cast the int 10 to the double 10.0.

Implicit Conversion

Implicit type conversion occurs in situations where a type issue arises, but can be resolved with the compiler's intervention between compatible types. An example:

Listing 10: Implicit type conversion

```
1:  public class ImplicitConversion{
2:    public static void main(String[] args){
3:      double d = 10;
4:    }
5:  }
```

Listing 10 works exactly the same as Listing 9 even though there is no explicit cast. The compiler recognizes that there is a type compatibility between doubles and ints as they are both numeric types (*arithmetic* types in the spec).

Conversion Rules

As we saw above, the Java programming language allows for conversions, both implicit and explicit, between data types. In order for this to be deterministic the language specification details exactly how certain conversions are meant to be achieved/implemented by the compiler. You are not responsible for memorizing all of the edge cases provided by the language specification for handling type conversion, but you should be aware that it can happen. Some rules you should be aware of:

- Java does not allow "narrowing" conversions to occur implicitly
  - Going from an int to a double is allowed, but a double to an int must be explicitly cast
- When converting from real values to integer values the fractional portion is always discarded
- When a larger data type is converted to a smaller data type (i.e., int to char), the least significant bits are retained.
- "Narrower" types will be automatically converted to "wider" types
  - `float` + `int` → `float` + `float`

Type Modifiers

Java also provides the ability for programmers to assign "extra" rules to a variable by providing type *modifiers*. As of Java23 there are three type modifiers: `final`, `volatile`, and `transient`. In this course, we will only concern ourselves with `final`. The final modifier turns the variable into a "constant", meaning once it has been assigned a value, that value cannot be changed. For example:

Listing 11: final qualifier

```
1:  public class Final{
2:    public static void main(String[] args){
3:      final int i;
4:      i = 10; // No problem
5:      i = 20; // Problem!
6:    }
7:  }
```

The above code will not compile because the integer **i** cannot be assigned a second value. You may think that it's odd that Java provides the ability to write constant values. After all, why not just write 10 for every instance of **i**? The reason is that sometimes you want to change constants when testing your code. For example, if I'm testing to see how many students pass CS210 it's easier for me to change a constant variable `final double PASS_SCORE` than it is for me to individually change it throughout my entire source code file. This is doubly true if the constant is propagated throughout dozens of files.

Exercises

1. The chapter states that narrower types are automatically converted to wider types in mixed arithmetic expressions. Write a Java program that divides two `int` variables where the result has a fractional component (e.g. `7 / 3`), stores it in a `double`, and prints it. Does the fractional part appear? Now try storing the result of `(double) 7 / 3` in a `double` and print again. Explain the difference in terms of implicit and explicit conversion.
2. Write a Java program that declares a `final int` representing the passing grade for this course (60). Attempt to change it after declaration and record the compiler error. Then remove the `final` qualifier and confirm it compiles. Why might using `final` be preferable to just writing the number 60 throughout your code?
3. Write a Java program that explicitly casts each of the following and prints the result: `(int) 3.9`, `(int) -3.9`, `(char) 65`, `(double) 7 / 2`, `(double) (7 / 2)`. Before running, predict each result. Explain any surprises.
4. The chapter states that when converting from a larger type to a smaller type, the least significant bits are retained. Write a Java program that assigns `int i = 256` to a `byte b` via explicit cast and prints both. Explain the result in terms of bit representation.
5. Java provides a way to inspect the size of primitive types through wrapper classes. Print the size in bits of `Byte`, `Integer`, `Float`, and `Double` using `Byte.SIZE`, `Integer.SIZE`, `Float.SIZE`, and `Double.SIZE`. Do they match what the chapter states?
6. The chapter contrasts Java's strong typing with C's weak typing using Listing 8. Write a Java program that attempts three different invalid type assignments of your own invention and record the compiler errors for each. For each error, explain in your own words why Java refuses to allow it.

7. Write a Java program that demonstrates implicit conversion in an arithmetic expression: declare an `int` and a `double`, add them together, and store the result in both an `int` and a `double` variable. Does the `int` storage require an explicit cast? Print all variables and explain what you observe.
8. The chapter mentions that Java is considered *strongly* typed while C is considered *weakly* typed despite both being statically typed. In your own words, explain the difference between static typing and strong typing using examples from the chapter.
9. Using Java's wrapper classes, print `Integer.MAX_VALUE`, `Integer.MIN_VALUE`, `Byte.MAX_VALUE`, and `Byte.MIN_VALUE`. Verify that the ranges match what you would expect given the bit sizes from the previous exercise.
10. Write a Java program that stores the value `0xFFFFFFFF` cast to an `int` and performs the following operations: addition of 1, division by 2, and multiplication by 2. Print all results. Then repeat using a `long` instead of an `int`. For each operation, explain why the `int` and `long` results differ.

Footnotes:

1 I'm writing this on 7/7/26. Exceptions I'm aware of are Brainfuck, which is intentionally esoteric, and BCPL which was the progenitor to C—neither of which are seriously used.

Operators

Introduction

Up to this point, we've discussed data, how that data is stored in variables, and how those variables must all have types. This gives us everything we need in order to understand how we can write Java code to make these typed variables *interact* with one another. These interactions can be effaced through the use of *operators* in Java, i.e., special symbols which have semantic meaning for computations involving data.

Infix Notation

When language designers begin the long, exciting process of building out a language, they have to make a choice on how exactly that language is going to look. One of the first considerations is what sort of notation will be used for parsing operations: prefix, infix, or postfix. A quick demonstration will show what I mean:

- Prefix → `+ 2 1`
- Infix → `2 + 1`
- Postfix → `2 1 +`

All of the above are ways of representing the semantic action of "add together the constants two and one". For the programmer, the difference is purely visual. For the language designer, or, more accurately, the compiler developer, the difference will manifest itself in how the language is *parsed*[1]. (We'll come back to this when we discuss stacks and queues.) For now, all we need to know is that Java uses infix notation for all of its binary operators.

Operators in Java

Here, I will enumerate all of the operators in Java and provide examples for how each one functions.

Assignment

Assignment operators are used to capture values and place them into variables.

=

= is the basic assignment operator. The left operand must be a *lvalue* while the right operand must have a *rvalue*. For the following examples, assume `int x` and `int y` have already been declared"

- `x = 1` → `1` is now stored in the memory location tied to `x`
- `x = y` → The value tied to the memory location of `y` at the time of evaluation is now stored in the memory location tied to `x`

Note: Without getting into too much theory, *lvalues* are locations in memory (variables, or pointer dereferences) whereas *rvalues* are data (which can include memory locations). The variable `x` returns an *l* or *r* value based on its surrounding context.

Unary Operators

Most operators in Java are *binary* operators which means they take two *operands*. In contrast, a unary operator only takes one operand.

++ / --

++/-- are the unary prefix increment/decrement operators. They perform an increment or decrement of 1 to the variable given. Assume `int x = 0;` has already been processed:

- `++x` → `1` and a side effect of setting `x` equal to `1`
- `--x` → `-1` and a side effect of setting `x` equal to `-1`

Note: The prefix operators above can be supplied to the end of a variable to create postfix operators. The difference is that `x++` would return `0`, the value of `x` prior to the increment/decrement. The side effect remains unchanged (`x` would be equal to `1`).

+

+ can be used as a unary operator. It has limited use and only exists to maintain symmetry between + and -.

- `+1` → `1`

-

- can be used as a unary operator for negation.

- `-1` → `-1`
- `-(-1)` → `1`

~

~ is the bitwise complement operator. I will flip the bits in the underlying integral type.

- `~((byte) 1)` → `-2` (`0b00000001` → `0b11111110`)
- `~0` → `-1` (`0x00000000` → `0xFFFFFFFE`)

!

! is the logical complement operator. It takes a boolean true/false and flips it to its opposite.

- `!true` → `false`
- `!false` → `true`

Multiplicative Operators

*

* is the multiplication operator. It is used to multiply two numeric values.

- `4 * 2` → `8`

/

/ is the division operator. It is used to divide two numeric values.

- `4 / 2` → `2`

Note: When two integers are involved integer division is performed. Functionally, that means the fractional result of the operation is removed. `2 / 4` equals `0`.

%

% is the remainder operator. It is used to get the remainder after the left operand is divided by the right operand.

- `4 % 2` → `0` (`2` goes into `4` two times with `0` remainder)
- `2 % 4` → `2` (`4` goes into `2` 0 times with `2` remainder)

Additive Operators

+ (Numerics)

+ is the addition operator when both operands are numerics. It is used to add two numeric values.

- `4 + 2` → `6`

-

- is the subtraction operator. It is used to subtract two numeric values.

- `4 - 2` → `2`

+ (Strings)

+ is the string concatenation operator when one of the operands is a String type. It is used to create a new String object.

- `"cat" + "dog"` → `"catdog"`
- `4 + "2"` → `"42"`
- `"2" + 4` → `"24"`

Implicit Conversion, Again

Something that often trips students up is the fact that implicit conversion kicks in during the use of binary arithmetic operators. For example,

- `1 + 1.5` → `double`
- `1 + 1.5f` → `float`
- `1 / 2.0` → `double`

And so on. In all of the above cases `1` is converted to the *wider* type.

Shift Operators

<<

<< is the left bitshift operator. From the specification: "The value of n << s is n left-shifted s bit positions; this is equivalent (even if overflow occurs) to multiplication by two to the power s."

- `0xf << 4` → `0xf0`
- `0x80000000 << 1` → `0x00`

>>

>> is the signed right bitshift operator. From the specification: "The value of n >> s is n right-shifted s bit positions with sign-extension. The resulting value is floor(n / 2^s). For non-negative values of n, this is

equivalent to truncating integer division, as computed by the integer division operator /, by two to the power s."

- `0xf0 >> 4` → `0x0f`
- `0xf >> 4` → `0x00`
- `0x80000000 >> 1` → `0xc0000000`

>>>

>>> is the signed right bitshift operator. From the specification:

> *The value of n >>> s is n right-shifted s bit positions with zero-extension, where:*
>
> - *If n is positive, then the result is the same as that of n >> s.*
> - *If n is negative and the type of the left-hand operand is int, then the result is equal to that of the expression (n >> s) + (2 << ~s).*
> - *If n is negative and the type of the left-hand operand is long, then the result is equal to that of the expression (n >> s) + (2L << ~s).*

- `0xf0 >>> 4` → `0x0f`
- `0xf >>> 4` → `0x00`
- `0x80000000 >>> 1` → `0x40000000`

Relational Operators

Relational operators are only capable of being used with numeric types[2]. They are used to perform comparisons between two values. All relational operations return a boolean type, either *true* or *false*.

>

> is the "greater than" operator. It is used to check if the left operand is greather than the right operand.

- `2 > 4` → `false`

<

< is the "less than" operator. It is used to check if the left operand is less than the right operand.

- `2 < 4` → `true`

>=

>= is the "greater than or equal to" operator. It is used to check if the left operand is greater than or equal to the right operand.

- `2 >= 4` → `false`
- `4 >= 4` → `true`

<=

<= is the "less than or equal to" operator. It is used to check if the left operand is less than or equal to the right operand.

- `2 <= 4` → `true`
- `4 <= 4` → `true`

Equality Operators

Equality operators are used to check equality between two values. All equality operations return either a boolean type.

==

== is the equality operator. It is used to check if the left operand is equal to the right operand.

- `2 == 4` → `false`
- `2 == 2` → `true`

Note: Strict equality is dangerous when comparing floating point values! Relational operators should be used instead when dealing with floating points.

Note: Similarly, equality is dangerous when comparing reference types (non-primitives)! We will learn more about this when we discuss reference types in more detail.

!=

!= is the inequality operator. It is used to check if the left operand is not equal to the right operand.

- `2 != 4` → `1`
- `2 != 2` → `0`

Bitwise Operators

Bitwise operators require an integral type. They are used to perform "bit-twiddling", i.e. setting/unsetting bits in integer values.

&

& is the bitwise **AND** operator. Each bit in the resulting value is set if and only if each of the corresponding bits in the converted operands is set.

- `0xff & 0xf0` → `0xf0`
- `0xfc & 0xf3` → `0xf0`
- `0xff & 0x0f` → `0x0f`

|

| is the bitwise **OR** operator. Each bit in the resulting value is set if one of the corresponding bits in the converted operands is set.

- `0xff | 0xf0` → `0xff`
- `0xfc | 0xf3` → `0xff`
- `0xff | 0x0f` → `0xff`

^

^ is the bitwise **XOR** operator. Each bit in the resulting value is set if and only if exactly one of the corresponding bits in the converted operands is set.

- `0xff ^ 0xf0` → `0x0f`
- `0xfc ^ 0xf3` → `0x0f`
- `0xff ^ 0x0f` → `0xf0`

~

~ is the bitwise **NOT** operator. Each bit in the resulting value is set if and only if the corresponding bits in the operand is not set.

- `~((char)0xff)` → `0x00`
- `~((char)0xfc)` → `0x03`
- `~((char)0x0f)` → `0xf0`

Note: If I don't explicitly cast the hexadecimal constant to a `char`, the javac compiler will keep it as an `int` and flip more bits than I wanted to show.

Conditional Operators

Conditional operators can only take the boolean type.

&&

&& is the conditional **AND** operator. It is used for boolean conjunction.

- `true && false` → `false`
- `true && true` → `true`

Note: `&&` is guaranteed to be evaluated from left to right. This means that if the left operand is *false*, the right operand will not be evaluated. This is often called "short circuiting".

||

|| is the conditional **OR** operator. It is used for boolean disjunction. Much like **&&**, **||** will short circuit if the left operand resolves to a *true*.

- `false || false` → `false`
- `true || false` → `true`

Operators in Other Programming Languages

It just so happens that Java-style syntax borrows heavily from the C programming language. Many of the symbols chosen by C are used in other programming languages. For example, almost all of the operators and their symbols in C are identical in Python—and by extension, Java. Even if they weren't, many programming languages share the exact same *logic* and abstractions that C does (at least in the general sense). In Java, C *and* Python, `2 * 3` represents the multiplication of the integers 2 and 3. Knowing the ins and outs of one programming language prepares you for learning another!

Exercises

1. Predict the result of each of the following expressions before writing a Java program to verify them. Pay close attention to integer division:
   - `7 / 2`
   - `7 / 2.0`
   - `7.0 / 2`
   - `(double) 7 / 2`
   - `(double) (7 / 2)`
2. Write a Java program that declares two `int` variables `a = 15` and `b = 4` and prints the result of all five arithmetic operators applied to them. What type is the result of each operation?
3. The chapter that strict equality is dangerous with floating point values. Write a Java program that computes `0.1 + 0.2` and checks if it equals `0.3` using `==`. Print the result. Then print the raw value of `0.1 + 0.2` using `System.out.printf("%.20f%n", 0.1 + 0.2)`. Explain what you observe.

4. Write a Java program that prints the results of all six relational operators (`>`, `<`, `>=`, `<=`, `==`, `!=`) comparing `a = 5` and `b = 5`. Predict each result before running.
5. The chapter states that `&&` and `||` short circuit. Write a Java program that demonstrates short circuiting by placing a division by zero on the right side of a `&&` where the left side is `false`, and on the right side of a `||` where the left side is `true`. Does the program crash? Explain why or why not.

6. Using only bitwise operators, write a Java program that:

    - Sets the 4th bit of `0x00` to 1 using `|`
    - Clears the 4th bit of `0xff` using `&`
    - Flips the 4th bit of `0xff` using `^`

    Print each result in hexadecimal using `System.out.printf("%x%n", result)`.

7. Write a Java program that demonstrates left and right bitshifting on the value `0x01`. Left shift it 4 times, printing the result each time. Then right shift the final result 4 times back, printing each time. What do you observe? Express each result in both hex and decimal. Then repeat using `>>>` instead of `>>` on a negative value and explain the difference.
8. The chapter states that `1 + 1.5` produces a `double`. Write a Java program that performs the following mixed-type operations and prints each result. Use `System.out.println` and observe what type is inferred from the output:
    - `int + float`
    - `int + double`
    - `float + double`
9. Evaluate the following expressions by hand using the bitwise operator rules, then verify in Java using `System.out.printf("%x%n", result)`:
    - `0xAB & 0x0F`
    - `0xAB | 0x0F`
    - `0xAB ^ 0x0F`
    - `~((byte)0xAB)`
10. The chapter introduces three different notations for writing operations: prefix, infix, and postfix. Rewrite the following Java expressions (which use infix notation) in both prefix and postfix notation:
    - `2 + 3`
    - `10 - 4`
    - `3 * 5`
    - `8 / 2`

Footnotes:

1. Most popular languages nowadays default to infix notation; however, there are still some languages which do not. For example, Lisp, a *relatively* popular functional programming language, and all of its various dialects use prefix notation. To my knowledge, Forth is the only programming language that anyone uses which sticks to postfix notation.

2. This is excluding the *instanceOf* relational operator which we will ignore for now.

Expressions

Introduction

Up to this point, we've learned what variables are and how to use operations to perform computations using those variables. What we haven't explored are the fundamental units within the code that allow for the compiler to return the values of said computations. These units are called *expressions*.

Evaluation

A commonality across all programming languages is that *expressions return values*. Take, for example, the simple expression `3 * 2`. We know now that it will *evaluate* to `6` because the `*` operator represents multiplication and the constants `3` and `2` are equivalent to their decimal symbols. What might be less obvious is that there are actually *three* expressions in that one, simple expression:

1. The constant value 3 in the left operand.
2. The constant value 2 in the right operand.
3. The operation of multiplying the left and right operand.

This can be made more clear if we were to replace `3` and `2` with the integer variables `x` and `y`, giving us `x * y`:

1. The value of the variable `x` (what is stored at the memory location mapped to the symbol `x`) in the left operand
2. The value of the variable `y` (what is stored at the memory location mapped to the symbol `y`) in the right operand
3. The operation of multiplying the left and right operand.

In order for the multiplication operation to be successful, the runtime must *evaluate* what is stored in the variables.

Precedence

All expressions must be able to be evaluated *deterministically* and *unambiguously*. What do I mean by that? Well, take for example a more complicated set of expressions, `5 * 2 + 3 * 4`. How exactly should that be evaluated by the compiler? Grade school math teaches us that it should evaluate to 22, and it turns out that's also the case in Java, but why is that? If I read the symbols from left to right it tells me I should multiply 5 by 2, then add 4, and then multiply by 4, giving me 52. This is because, without *precedence rules*, infix notation is inherently *ambiguous*—there's no way to

know how to order the expressions without external information about the symbols.

All programming languages that use infix notation must define precedence rules that govern expression evaluation order. It follows then that the Java standard has exactly that:

Operators	Associativity
() [] .	left to right
! ~ ++ -- + - (type)	right to left
* / %	left to right
+ - +	left to right
<< >> >>>	left to right
< <= > >= instanceof	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= |= <<= >>= >>>=	right to left

Note: In the second row, `+`, `-`, and `*` are unary operators.

When using the above rules, there can be absolutely no abiguity about the expression `5 * 2 + 3 * 4`. We can see that binary `*` takes higher precedence than binary `+`. There are quite a lot of rules here, but most will become second nature after you get some experience under your belt.

Using Parentheses

Oftentimes, it can become a bit confusing to parse out how exactly a certain expression will be evaluated. Case in point, let's take a look at this monstrosity `1 + ~2 + 3 * 4 / 5 * 6 & 7 | 8`. What we see here *is* valid Java, and any working Java compiler will be able to evaluate the given expression. In fact, I had Claude Sonnet generate a Java program, which you can download here, to show exactly how that expression gets parsed:

Listing 1: Abstract Syntax Tree for `int x = 1 + ~2 + 3 * 4 / 5 * 6 & 7 | 8`

```
COMPILATION_UNIT
      BLOCK
        VARIABLE: int x = 1 + ~2 + 3 * 4 / 5 * 6 & 7 | 8
          MODIFIERS:
          PRIMITIVE_TYPE: int
          OR: 1 + ~2 + 3 * 4 / 5 * 6 & 7 | 8
            AND: 1 + ~2 + 3 * 4 / 5 * 6 & 7
              PLUS: 1 + ~2 + 3 * 4 / 5 * 6
                PLUS: 1 + ~2
                  INT_LITERAL: 1
                  BITWISE_COMPLEMENT: ~2
                    INT_LITERAL: 2
                MULTIPLY: 3 * 4 / 5 * 6
                  DIVIDE: 3 * 4 / 5
                    MULTIPLY: 3 * 4
                      INT_LITERAL: 3
                      INT_LITERAL: 4
                    INT_LITERAL: 5
                  INT_LITERAL: 6
              INT_LITERAL: 7
            INT_LITERAL: 8
```

Visually, there's so much noise in the original expression that it's difficult for even a veteran Java programmer to understand what's happening at first glance. This is where using parentheses can be very helpful to the new Java programmer. If I wrap the complicated expression in parentheses instead, it becomes more obvious what's going on:

- `(((1 + (~2)) + (((3 * 4) / 5) * 6)) & 7) | 8`

Parentheses can make things a bit more visually noisy, but many text editors, like Emacs, come preequipped with parentheses highlighting which makes the task a bit easier.

Expressions Contra Statements

Expressions can be constrasted with statements insofar as expressions always return values whereas statements do not. For example, an `if` statement, which we'll discuss next lecture, handles control flow, it does not return a value. This is in direct contradistinction with the `+` operator which will always return the value equal to the sum of the two operands on either side of it.

Exercises

1. Evaluate the following expressions by hand using the precedence table, then verify in Java using `System.out.println(expr)`:
   - `5 * 2 + 3 * 4`
   - `5 * (2 + 3) * 4`
   - `10 - 4 / 2 + 1`
   - `10 - 4 / (2 + 1)`

2. The chapter identifies three sub-expressions in `3 * 2`. How many sub-expressions can you identify in `5 * 2 + 3 * 4`? List each one and the order in which they are evaluated according to the precedence table.
3. Download the AST builder program linked in the chapter and run it on a Java file containing the expression `1 + ~2 + 3 * 4 / 5 * 6 & 7 | 8`. Verify the output matches Listing 1. Then add parentheses to the expression to change the evaluation order and observe how the AST changes.
4. Write the fully parenthesized version of the following expressions as the compiler would evaluate them, then verify by computing the result:
    - `8 >> 1 + 1`
    - `1 | 2 & 3`
    - `!true || !false && true`
    - `3 * 4 + 5 * 6 - 7 / 2`
5. The second row of the precedence table notes that `+` and `-` appear as unary operators. Write a Java program that demonstrates the difference between unary and binary `+` and `-` by using each in context. Note that unlike C, Java has no unary `*` operator — what does C's unary `*` do that Java handles differently?
6. Write a Java program that declares `int x = 5` and `int y = 3` and evaluates `x = y = 2`. Print `x` and `y` after the assignment. Using the associativity column of the precedence table, explain why the result is what it is.
7. The chapter states that expressions always return values but statements do not. Classify each of the following as an expression or a statement and explain why:
    - `x + 1`
    - `int x = 5`
    - `x = x + 1`
    - `3 * 4`
8. Write a Java program that evaluates `(((1 + (~2)) + (((3 * 4) / 5) * 6)) & 7) | 8` and the equivalent unparenthesized version `1 + ~2 + 3 * 4 / 5 * 6 & 7 | 8`. Do they produce the same result? What does this confirm about the precedence rules?
9. Using the precedence table, determine which of the following pairs of expressions are equivalent without running them. Then verify in Java:
    - `a + b * c` and `a + (b * c)`
    - `a & b | c` and `(a & b) | c`
    - `!a && b` and `(!a) && b`
    - `a = b = 0` and `a = (b = 0)`
10. The chapter gives `1 + ~2 + 3 * 4 / 5 * 6 & 7 | 8` as an example of a confusing but valid Java expression. Write your own complex expression using at least five different operators, write its fully parenthesized form, compute the result by hand, verify with a Java program, and then run it through `ASTDumper.java` to confirm the parse tree matches your parenthesized version.

Conditional Statements

Introduction

Think about the last time you went to a grocery store. If it's anything like my local supermarket, when you approach the sliding doors to enter they appear to open as if they were operated by invisible elves. If we're able to reasonably convince ourselves that it isn't magical elves, there must be something present allowing for that movement to occur. Further investigation would have us finding a sensor which is able to detect whether some body is in front of the door and *if* that's the case the mechanism to open the door should fire. Furthermore, there is something preventing the door from opening *if* there is nothing detectable. The internal logic of the automatic door is *conditional* on the *state* of its components.

Programs too will have special logic that will only be used in the event a certain condition is met. In Java, these logical scaffolds are called *conditional statements*.

Control Flow

Conditional statements are used to direct the runtime behavior of the executing program. Depending on how they are evaluated, conditional statements will decide which set of instructions are processed next. In computer science we call this the *control flow* of a program—i.e. how each instruction *flows* into the next. Java provides a few different statements to aid in the construction of a program's control flow.

if Statement

The `if` statement is the fundamental conditional statement. It's syntax is as follows:

Listing 1: if statement syntax

```
1:   if(conditional)
2:     statement; //Do this
```

As long as `conditional` evaluates to a *true* value then the `statement` will be executed. A filled in example should illustrate the point:

Listing 2: Conditional example

```
1:  public class If{
2:      static final int X = _; // Will set later...
3:      public static void main(String[] args){
4:          System.out.println("Welcome to the even checker!");
5:          if(X % 2 == 0)
6:              System.out.printf("%d is even!%n", X);
7:      }
8:  }
```

If I compile and run this with `X` set to 4, I get the following results:

Listing 3: Listing 2 with X = 4

```
josephraskind@stargazer:/tmp/cond$ javac If.java; java If
Welcome to the even checker!
4 is even!
```

What about if I set `X` equal to 5?

Listing 4: Listing 2 with X = 5

```
josephraskind@stargazer:/tmp/cond$ javac If.java; java If
Welcome to the even checker!
```

What happened to my printout? Well, since the conditional failed the corresponding call to `System.out.printf` was not executed. The `printf` statement is attached to the `if` statement evaluating a *true* conditional—it will **never** execute if a *false* is evaluated.

Taking a look at the bytecode generated by `javap -c If.class` should help us understand what's happening:

Listing 5: Listing 2 as JVM bytecode

```
 1:   public class If {
 2:     static int X;
 3:
 4:     public If();
 5:       Code:
 6:          0: aload_0
 7:          1: invokespecial #1                  // Method java/lang/
Object."<init>":()V
 8:          4: return
 9:
10:     public static void main(java.lang.String[]);
11:       Code:
12:          0: getstatic     #7                  // Field java/lang/
System.out:Ljava/io/PrintStream;
13:          3: ldc           #13                 // String Welcome to the
even checker!
14:          5: invokevirtual #15                 // Method java/io/
PrintStream.println:(Ljava/lang/String;)V
15:          8: getstatic     #21                 // Field X:I
16:         11: iconst_2
17:         12: irem
18:         13: ifne          38
19:         16: getstatic     #7                  // Field java/lang/
System.out:Ljava/io/PrintStream;
20:         19: ldc           #27                 // String %d is even!%n
21:         21: iconst_1
22:         22: anewarray     #2                  // class java/lang/Object
23:         25: dup
24:         26: iconst_0
25:         27: getstatic     #21                 // Field X:I
26:         30: invokestatic  #29                 // Method java/lang/
Integer.valueOf:(I)Ljava/lang/Integer;
27:         33: aastore
28:         34: invokevirtual #35                 // Method java/io/
PrintStream.printf:(Ljava/lang/String;[Ljava/lang/Object;)Ljava/io/PrintStream;
29:         37: pop
30:         38: return
31:
32:     static {};
33:       Code:
34:          0: iconst_5
35:          1: putstatic     #21                 // Field X:I
36:          4: return
37:   }
```

There's a lot going on here. First, `#13` and `#27` represent the locations in the constant pool for the two string constants used in the print statements. Line 13 loads the `"Welcome..."` string into the first argument, and Line 14 calls `System.out.println`. Lines 15-17 represent the conditional expression `x % 2 == 0`: 4 is loaded into the operator stack alongside the constant 2 and the integer remainder operator is called. Line 18 represents the conditional check which is seeing if the remainder is not equal to 0. If the remainder does not equal to 0 then control will be redirected over to Line 30 or instruction `38:`, otherwise every further instruction will be processed. Therefore when `X` is equal to 5 control will be transferred ober to `38:` and the second call to `printf` will be passed over.

`if` statements can also encompass more than one statement. For example:

Listing 6: if statement syntax with blocks

```
1:   if(conditional){ // Start block with {
2:     statement 1;
3:     statement 2;
4:     ....
5:     statement N;
6:    } // End block with }
```

Applied to a "real code" expmaple we would see:

Listing 7: Conditional example with block

```
 1:   public class If{
 2:       static final int X = _; // Will set later...
 3:       public static void main(String[] args){
 4:           System.out.println("Welcome to the even checker!");
 5:           if(X % 2 == 0){
 6:               System.out.printf("%d is even!%n", X);
 7:               System.out.printf("Isn't that neat!\n");
 8:           }
 9:       }
10:   }
```

Running the above with `X` equal to 10 we get:

Listing 8: Listing 7 with X = 10

```
josephraskind@stargazer:/tmp/cond$ javac If.java; java If
Welcome to the even checker!
10 is even!
Isn't that neat!
```

And with `X` equal to 13:

Listing 9: Listing 7 with X = 13

```
josephraskind@stargazer:/tmp/cond$ javac If.java; java If
Welcome to the even checker!
```

Note: Be careful when omitting "{}" from your conditional statements! Without them, Java will only match the first statement to the conditional's control flow behaviour. It's good practice to ere on the side of caution and always include brackets when possible.

else if

`if` statements can be further modified to allow for mutually exclusive behaviors. For example, say we didn't want users to enter negative numbers we could write:

Listing 10: Checking for negatives with three /if/s

```
 1:  public class Ifs{
 2:      static final int X = -4; // Will set later...
 3:      public static void main(String[] args){
 4:          System.out.printf("Welcome to the even checker!\n");
 5:          if(X < 0){
 6:              System.out.printf("Ew! %d is negative!\n", X);
 7:          }
 8:          if(X % 2 == 0){
 9:              System.out.printf("%d is even!\n", X);
10:          }
11:          if(X % 2 != 0){
12:              System.out.printf("%d is odd!\n", X);
13:          }
14:      }
15:  }
```

But if I run the above code with `X` equal to -4 I get the following results:

Listing 11: Conditional example with two conditionals

```
josephraskind@stargazer:/tmp/cond$ javac Ifs.java; java Ifs
Welcome to the even checker!
Ew! -4 is negative!
-4 is even!
```

But this wasn't what I wanted! I didn't want to print out the evenness of the negative number!

We can solve this with the use of an `else if` clause:

Listing 12: Using else if

```
 1:  public class Ifs{
 2:      static final int X = _; // Will set later...
 3:      public static void main(String[] args){
 4:          System.out.printf("Welcome to the even checker!\n");
 5:          if(X < 0){
 6:              System.out.printf("Ew! %d is negative!\n", X);
 7:          }else if(X % 2 == 0){
 8:              System.out.printf("%d is even!\n", X);
 9:          }else if(X % 2 != 0){
10:              System.out.printf("%d is odd!\n", X);
11:          }
12:      }
13:  }
```

Now if I run the above code with `X` equal to -4 we'll see something different:

Listing 13: Conditional example if else

```
josephraskind@stargazer:/tmp/cond$ javac Ifs.java; java Ifs
Welcome to the even checker!
Ew! -4 is negative!
```

The `if else` clause makes each subsequent `if` block *mututally exclusive*. No two blocks can be entered for the same set of conditional checks.

else

Sometimes we want mutually exclusive behaviors without having to write an additional conditional check. For example,

Listing 14: Conditional example with two conditionals

```
1:  public class Else{
2:      final static int X = 4; // Will set later...
3:      public static void main(String[] args){
4:          System.out.printf("Welcome to the even checker!\n");
5:          if(X % 2 == 0){
6:              System.out.printf("%d is even!\n", X);
7:          }
8:          if(X % 2 != 0){
9:              System.out.printf("%d is odd!\n", X);
10:         }
11:     }
12: }
```

gets the job done, but it looks a bit clunky. There's a lot more work our eyes have to do when scanning the code. Instead we can write,

Listing 15: if-else statement

```
1:  public class Else{
2:      final static int X = 4; // Will set later...
3:      public static void main(String[] args){
4:          System.out.printf("Welcome to the even checker!\n");
5:          if(X % 2 == 0){
6:              System.out.printf("%d is even!\n", X);
7:          }else{
8:                  System.out.printf("%d is odd!\n", X);
9:          }
10:     }
11: }
```

now `X % 2 != 0` is implied as it is the only other possibility if the original conditional check fails.

We can combine this with our negative check above (Listing 12):

Listing 16: Using else if with else

```
1:   public class Ifs{
2:       static final int X = -4; // Will set later...
3:       public static void main(String[] args){
4:           System.out.printf("Welcome to the even checker!\n");
5:           if(X < 0){
6:               System.out.printf("Ew! %d is negative!\n", X);
7:           }else if(X % 2 == 0){
8:               System.out.printf("%d is even!\n", X);
9:           }else(X % 2 != 0){
10:                  System.out.printf("%d is odd!\n", X);
11:          }
12:      }
13:  }
```

Note: You can only ever have one `else` in your if statement conditional chain.

switch Statement

The `switch` statement is a variation of the `if` statement. It provides slightly different functionality to using `if-else` statement chains. The syntax is as follows:

Listing 17: switch statement syntax

```
1:   switch(expr){
2:    case SOMETHING:
3:       statement 1; // Do statement
4:    case SOMETHING_ELSE:
5:      statement 2; // Do statement
6:      break; // Exit control flow
7:    default:
8:      statement 3; // Do statement
9:   }
```

The `switch` statement takes an expression that resolves to an integral type, a String type, or an Enum type. And will transfer control to the block of code underneath the corresponding label. The `default` label is used when no corresponding label for the expression's value can be found. The `break` statement transfers control out of the switch block—this is particularly useful because without it the control will "fall through" to the next case. Meaning in the above example if `expr` resolves to `SOMETHING` then both `statement 1` and `statement 2` will be executed (`statement 3` will not because of the `break` statement).

?: Operator

Although not strictly a conditional statement, the `?:` "ternary" operator is often bunched into the same group due to its conditional evaluation logic. The syntax is as follows

Listing 18: Ternary operator syntax

```
conditional ? eval_true : eval_false
```

Using the evenness example from earlier:

Listing 19: Using else if

```
1:   public class Ternary{
2:       final static int X = _;
3:       public static void main(String[] args){
4:         System.out.printf("Welcome to the even checker!\n");
5:         System.out.printf(X % 2 == 0 ? "%d is even!\n" : "%d is odd!\n" ,
X);
6:       }
7:   }
```

Compiling and running the above gives us:

Listing 20: Using the ternary expression

```
josephraskind@stargazer:/tmp/cond$ javac Ternary.java; java Ternary
Welcome to the even checker!
4 is even!
```

Exercises

1. Write a Java program using an `if` statement that checks if a `final int X` is greater than 100 and prints "X is large!" if so. Test it with at least three different values. What happens when the condition is false?
2. Modify Listing 2 to also print "X is not even!" when `X` is odd using an `else` clause. Test with both even and odd values. Why is this preferable to writing a second `if` statement with the condition `X % 2 != 0`?
3. Write a Java program that uses `if`, `else if`, and `else` to classify a `final int X` into one of the following categories and prints the result:
    - Negative
    - Zero
    - Between 1 and 100 inclusive
    - Greater than 100
4. The lecture warns that omitting `{}` from an `if` statement only attaches the first statement to the conditional. Write a Java program that demonstrates this bug by writing an `if` without braces followed by two `System.out.println` calls. Which call is controlled by the `if`? Verify by testing with a condition that evaluates to false.

5. Write a Java program using a `switch` statement that takes a `final int day` between 1 and 7 and prints the corresponding day of the week. Use a `default` case for any value outside that range.
6. The lecture states that without a `break` statement, control "falls through" to the next case in a `switch`. Write a Java program that demonstrates this by writing a `switch` with three cases and no `break` statements. What gets printed when the first case matches? When might fall-through behavior actually be useful?

7. Rewrite the following `if-else` chain as a `switch` statement:

```
1:  final int X = 2;
2:  if(X == 1) System.out.println("one");
3:  else if(X == 2) System.out.println("two");
4:  else if(X == 3) System.out.println("three");
5:  else System.out.println("other");
```

   Are there any cases where a `switch` cannot replace an `if-else` chain? Note that unlike C, Java's `switch` also accepts `String` and `enum` types.

8. Write a Java program that uses nested `if` statements to check whether a `final int X` is both positive and even. Print an appropriate message for each of the four possible combinations: positive even, positive odd, negative even, negative odd.
9. The bytecode output in Listing 5 shows that the JVM uses an `ifne` (jump if not equal to zero) instruction to implement the `if` statement. Looking at the bytecode, identify which line performs the conditional jump and where control is transferred when the condition is false. How does this compare to what you would expect from the high-level Java source?
10. Further modify the ternary example in Listing 19 to include a check for a negative number all in the same line.

Loops & Arrays

Introduction

Oftentimes we run into situations where we would like to repeat the same computation multiple times. As an example, consider a program which computes the power of one number raised to another. We know that 3^8 is equivalent to $3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3$. This would be easy enough to hard code[1]. We could write the following Java program:

Listing 1: Naive 3^8 program

```
1:  public class Pow{
2:      public static void main(String[] args){
3:          int val = 3 * 3 * 3 * 3 * 3 * 3 * 3 * 3;
4:          System.out.printf("3^8 = %d\n", val);
5:      }
6:  }
```

And it works well enough, printing the expected `3^8 = 6561`. But what if **3** and **8** weren't constant values, but variables? If I replace 3 with `x`:

Listing 2: Naive power with variables

```
1:  public class Pow{
2:      public static void main(String[] args){
3:          int x = 3;
4:          int y = 10;
5:          int val = 3 * 3 * 3 * 3 * 3 * 3 * 3 * 3;
6:          System.out.printf("%d^%d = %d\n", x, y, val);
7:      }
8:  }
```

If we run the above, we don't get the expected `3^10 = 59049`, but `3^10 = 6561`! It's clear that because Line 5 does not adapt to the fact that the target power has changed there can be no alteration in the computational task. One way we can fix this is with *looping*.

while Loops

There are three looping constructs provided by the Java programming language, two of which are slight variatios on the foundational `while` loop. The syntax of a while loop is as follows:

Listing 3: while loop syntax

```
1:  while(condition)
2:    block; // Do this
```

Pretty simple! While a certain condition is met, i.e. the expression resolves to *true*, the instructions in the block will be executed. Once the instructions in the block have been exhausted the condition is checked again. If it is met the instructions start over (hence the "loop"); if the condition is not met the next set of instructions outside of the loop will be executed.

Here's how we can modify the power example to work with a while loop:

Listing 4: power as a while loop

```
 1:   public class Pow{
 2:       public static void main(String[] args){
 3:           int x = 3;       // Base
 4:           int y = 10;      // Target power
 5:           int pow = 1;     // Starting accumulator value
 6:           int i = 0;       // Iterator
 7:           while(i < y){    // Conditional check
 8:               pow = pow * x; // Computation
 9:               i = i + 1;     // Incrementing i to eventually satisfy
looping condition
10:           }
11:           System.out.printf("%d^%d = %d\n", x, y, pow);
12:       }
13:   }
```

There's a lot going on here. First, you'll notice there are two more variables: `pow` and `i`. `pow` is an accumulator which stores the partial result of each computational step needed to get the final power value. `i` acts as a tally that keeps track of how many loops have been performed. Additionally, you can see that there's some new logic: a conditional operator `i < y` and a code block. The conditional is used to limit the number of loops performed (more on this later), and the code block is for defining a set of computations to be performed each loop. Here, the code block is multiplying the `pow` accumulator with the base `x` and incrementing the `i` loop tally. We can trace the value at the end of every loop for each variable in a table to keep track of what's happening:

Loop #	i	pow
1	1	3
2	2	9
3	3	27
4	4	81
5	5	243
6	6	729
7	7	2187
8	8	6561
9	9	19683
10	10	59049

After the 10th loop completes the conditional is checked on final time. Since `i` is now equal to 10 the expression `i < 10` resolves to *false* and

control is transferred to the first block outside of the loop, which is the print function call.

Infinite Loops

It is possible to accidentally create loops that never terminate. These are often referred to as *infinite loops*. Here's an example of how one could occur through a little bug in our program:

Listing 5: power implementation that loops infinitely

```
1:  public class Pow{
2:        public static void main(String[] args){
3:          int x = 3;          // Base
4:          int y = 10;         // Target power
5:          int pow = 1;        // Starting accumulator value
6:          int i = 0;          // Iterator
7:          while(i < y){       // Conditional check
8:              pow = pow * x; // Computation
9:                             // Something's missing
10:         }
11:         System.out.printf("%d^%d = %d\n", x, y, pow);
12:       }
13:   }
```

We can see here that the body of the while loop has had the tallying statement removed. This will cause i to never be incremented, thus keeping it at 0, and therefore never allowing for the conditional check to change values.

Note: Try compiling the code from Listing 4 yourself. You'll find that the compiler does not warn you of the infinite loop!

Infinite loops are not always a bad thing. Sometimes it makes sense for a program to include an infinite loop. For example, a chat room program might want to constantly poll a collection of network socket connections to see whether or not a new message has come through. Or, maybe a video game program might want to have a thread dedicated to always painting the screen for graphics content. What's important is to recognize when an infinite loop is happening and how to stop it if it's *not what you intended*.

How Are Loops Implemented?

In order to get a better intuition for how loops work we can take a look at how the assembly code is generated for them. Here's what it looks like when I run `javac Pow.java; javap -c Pow.class` on Listing 4:

Listing 6: Listing 4 in x86-64 assembly

```
1:  public static void main(java.lang.String[]);
2:    Code:
3:       0: iconst_3
4:       1: istore_1
5:       2: bipush        10
6:       4: istore_2
7:       5: iconst_1
8:       6: istore_3
9:       7: iconst_0
10:      8: istore        4
11:     10: iload         4
12:     12: iload_2
13:     13: if_icmpge     26
14:     16: iload_3
15:     17: iload_1
16:     18: imul
17:     19: istore_3
18:     20: iinc          4, 1
19:     23: goto          10
20:     26: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
21:     29: ldc           #13                 // String %d^%d = %d\n
22:     31: iconst_3
23:     32: anewarray     #2                  // class java/lang/Object
24:     35: dup
25:     36: iconst_0
26:     37: iload_1
27:     38: invokestatic  #15                 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
28:     41: aastore
29:     42: dup
30:     43: iconst_1
31:     44: iload_2
32:     45: invokestatic  #15                 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
33:     48: aastore
34:     49: dup
35:     50: iconst_2
36:     51: iload_3
37:     52: invokestatic  #15                 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
38:     55: aastore
39:     56: invokevirtual #21                 // Method java/io/PrintStream.printf:(Ljava/lang/String;[Ljava/lang/
Object;)Ljava/io/PrintStream;
40:     59: pop
41:     60: return
```

Again, there's a lot going on here so we'll focus on the major points. There are three labels to be aware of here `0:`, the label for the start of `main`, `10:`, the label for the conditional check, and `26:`, the label for the code block outside of the loop. Here is a table representing the variables to stack locations:

Variable	Location
x	1
y	2
pow	3
i	4

Line 11 starts us on our loop with `iload 4` which puts the value of the `i` variable onto the operations stack. The same is done for `y` on Line 12. On Line 13 the instruction `if_icmpge` checks to see if `i >= y`. If so, we jump to `26:` on Line 20; if not, we fall through and continue the loop. The body of the loop spans Lines 14-18. It concludes with an unconditional jump on Line 19 back to the conditional check. The rest of the bytecode is setting up and executing the `System.out.printf` invocation.

do while Loops

Java provides a slight variation on the `while` loop with the `do while` loop. The syntax is as follows:

Listing 7: while loop syntax

```
do{
  block; // Do this
}while(condition); // Needs a ; to terminate statement
```

The biggest difference here is that the block is processed first *before* the first conditional check. The `do while` construct does not add anything new, it just makes certain kinds of looping logic a bit easier to write. For example, let's say I wanted to write a program that prompted the user to enter a positive number. I could write:

Listing 8: while-based positive number prompt

```
import java.util.Scanner;
public class PositiveCheck{
    public static void main(String[] args){
        int x;
        System.out.printf("Enter a positive number: ");
        Scanner sc = new Scanner(System.in);
        x = sc.nextInt();
        while(x <= 0){
            System.out.printf("Enter a positive number: ");
            x = sc.nextInt();
        }
        System.out.printf("You entered: %d\n", x);

    }
}
```

I could cut down on the repeat code with a `do while` statement instead:

Listing 9: do while-based positive number prompt

```
import java.util.Scanner;
public class PositiveCheck{
    public static void main(String[] args){
        int x;
        Scanner sc = new Scanner(System.in);
        do{
            System.out.printf("Enter a positive number: ");
            x = sc.nextInt();
        }while(x <= 0);
      System.out.printf("You entered: %d\n", x);

    }
}
```

for Loops

The last looping structure provided by Java is likely also it's most famous: the `for` loop. The syntax for the `for` loop is as follows:

Listing 10: while loop syntax

```
1:   for(expr1 ; expr2 ; expr3)
2:     block; // Do something
```

`expr1` is the *declaration expression*. It is where you can define or set a variable at the entrance of the `for` loop. `expr2` is the *conditional expression*. It is where you can place the conditional check that will be used to determine if another loop will be completed. `expr3` is the *increment expression*. It is used for performing an additional operation at the end of the loop that is not included in the block body. `expr1`, `expr2`, and `expr3` are all optional.

Note: When `expr2` is not present it will be treated as a *true* value. This means that `for(;;)` is equivalent to a `while(true)` statement.

Here is an example of the power program with a `for` loop:

Listing 11: power as a for loop

```
 1:   public class Pow{
 2:       public static void main(String[] args){
 3:           int x = 3;                          // Base
 4:           int y = 10;                         // Target power
 5:           int pow = 1;                        // Starting accumulator value
 6:           for(int i = 0; i < y; i = i + 1){ // Declaration, conditional
check, and increment expression
 7:               pow = pow * x;                // Computation
 8:           }
 9:           System.out.printf("%d^%d = %d\n", x, y, pow);
10:       }
11:   }
```

Both Listing 4 and Listing 11 are functionally equivalent[2]. The obvious change is that the loop tallying variable, **i**, has been declared in `expr1` and is incremented in `expr3`. Everything else is identical.

Many people prefer to write `for` loops over `while` loops due to convenience. From the compiler's perspective, it often does not matter.

Note: Additions like the `for` loop and the `do while` loop can be classified as *syntactic sugar*. They do not provide additional functionality for the language, but they provide stylisitic benefits.

Nested Loops

Loops can be nested within one another. For example, if I wanted to print out a 4x3 grid of asterixes I could write:

Listing 12: Nested for loops

```
1:  public class NestedLoop{
2:      public static void main(String[] args){
3:          for(int i = 0; i < 4; i++){
4:              for(int j = 0; j < 3; j++){
5:                  System.out.printf("*");
6:              }
7:              System.out.printf("\n");
8:          }
9:      }
10: }
```

There are many situations where nested for loops are desirable, particularly with complex data structures.

break Keyword

We can use the `break` statement that we learned about with the switch statement. `break` will immediately transfer control out of the loop to the next block available as if the condition failed. If we did not want our power function to iterate anymore if pow was greater than 100:

Listing 13: power with break

```
1:  public class Pow{
2:      public static void main(String[] args){
3:          int x = 3;          // Base
4:          int y = 10;         // Target power
5:          int pow = 1;        // Starting accumulator value
6:          int i = 0;          // Iterator
7:          while(i < y){       // Conditional check
8:              pow = pow * x;  // Computation
9:              if(pow > 100)
10:                 break;      // Exists the loop
11:             i = i + 1;      // Incrementing i to eventually satisfy
looping condition
12:         }
13:         System.out.printf("%d^%d = %d\n", x, y, pow);
14:     }
15: }
```

I leave it as an exercise to you to see what happens when the above is ran.

continue Keyword

Similarly to the `break` keyword, the `continue` keyword alters the control flow of a loop. When a `continue` is reached control flow will be transferred to the "end" of the loop. For a `while` loop that means the conditional will immediately be rechecked. For a `for` loop that means the *increment expression* will be performed then the conditional will be checked. Here is a `for` loop that sums up all even numbers from 1 to 100:

Listing 14: Even sum with continue

```
 1:   public class ContinueLoop{
 2:       public static void main(String[] args){
 3:           int sum = 0;
 4:           for(int i = 0; i < 100; i++){
 5:               if(i % 2 == 1)
 6:                   continue;
 7:               sum = sum + i;
 8:           }
 9:           System.out.printf("sum = %d\n", pow);
10:       }
11:   }
```

Arrays

Arrays refer to a special sort of type which represent contiguosly allocated nonempty values. Arrays have a base, or *element*, type associated with them. For example, we can define an integer array, `arr`, with ten elements using the following syntax: `int arr[] = new int[10]`. Unlike C, arrays are guaranteed to have set default values. We do also have the option to set array values as well. Let's have a look:

Listing 15: Uninitialized array

```
 1:   public class UninitArray{
 2:       public static void main(String[] args){
 3:           int arr[] = new int[5];                 // Declaring an int
array of 5 elements
 4:           for(int i = 0; i < 5; i++)
 5:                 System.out.printf("%d\n", arr[i]); // The [] are used for
selecting an element
 6:             int arr_z[] = new int[]{1,2,3,4,5};    // Declaring a new
array with initialized values
 7:             for(int i = 0; i < 5; i++)
 8:                 System.out.printf("%d\n", arr_z[i]);
 9:       }
10:   }
```

And after we compile and run the above:

Listing 16: Uninitialized array results

```
 1:   josephraskind@stargazer:/tmp/arrays$ javac UninitArray.java; java
UninitArray
 2:   0
 3:   0
 4:   0
 5:   0
 6:   0
 7:   1
 8:   2
 9:   3
10:   4
11:   5
```

Lines 2-6 print out the elements of the unitialized, default-value array. Lines 7-11 print out the elements of the initialized array.

Arrays are *contiguous* in memory. This means that there is an entire block of memory reserved for the elements of that array side-by-side. When I write `int x[] = new int[5]`, that means that on my machine 20 bytes are reserved for the array `x` (5 4 byte ints). We can see this in code:

Listing 17: Contiguous array

```
 1:  import sun.misc.Unsafe;
 2:  import java.lang.reflect.Field;
 3:
 4:  public class ContiguousArray {
 5:      public static void main(String[] args) throws Exception {
 6:          Unsafe unsafe = getUnsafe();
 7:
 8:          int[] arr = {1, 2, 3, 4, 5};
 9:
10:          long scale = unsafe.arrayIndexScale(int[].class);
11:          long base  = unsafe.arrayBaseOffset(int[].class); // Objects
have header information
12:
13:          // Get the actual base address of the arr instance
14:          Object[] holder = new Object[]{arr};
15:          long arrAddress = unsafe.getLong(arr,
unsafe.arrayBaseOffset(Object[].class));
16:
17:          System.out.println("Element size (bytes): " + scale);
18:          System.out.println("Array size (bytes):   " + (scale *
arr.length));
19:          System.out.println();
20:
21:          for (int i = 0; i < arr.length; i++) {
22:              long address = arrAddress + base + (scale * i);
23:              System.out.printf("address 0x%x: %d%n", address, arr[i]);
24:          }
25:      }
26:
27:      private static Unsafe getUnsafe() throws Exception {
28:          Field f = Unsafe.class.getDeclaredField("theUnsafe");
29:          f.setAccessible(true);
30:          return (Unsafe) f.get(null);
31:      }
32:  }
```

After compiling and running the above:

Listing 18: Contiguous array results

```
1:  Element size (bytes): 4
2:  Array size (bytes):   20
3:
4:  address 0x200000011: 1
5:  address 0x200000015: 2
6:  address 0x200000019: 3
7:  address 0x20000001d: 4
8:  address 0x200000021: 5
```

You'll notice that every address is 4 spots higher than the last. Again, this is because each element is an `int`, and in Java an `int` is always 4 bytes.

[] operator

[] is an operator used for extracting values in an array. Given an integer array, `n`, of the first 10 natural numbers:

- `n[0]` → `1`
- `n[1]` → `2`
- etc.

Note: Arrays are indexed by 0. The first element matches to the index 0, the second element matches to the index 1, and so on.

[] also returns *lvalues*, so they can be used for assignment.

- if `n[0] = 10` then `n[0]` → `1`

Note: Java performs runtime checking to prevent "out of bounds" accessing. An *exception* is raised if an invalid index is provided.

Multidimensional Arrays

Arrays can take on more than 1 dimmension. For example, I could construct a 3x4 matrix using the following syntax: `int matrix[][] = new int[3][4]`. This creates an array of 12 integer elements which can be indexed using two sets of brackets. Conceptually we can think of the matrix looking like:

Listing 19: Two dimensional array

```
        [0]        [1]        [2]        [3]
    +----------+----------+----------+----------+
[0] |    1     |    2     |    3     |    4     |
    +----------+----------+----------+----------+
[1] |    5     |    6     |    7     |    8     |
    +----------+----------+----------+----------+
[2] |    9     |    10    |    11    |    12    |
    +----------+----------+----------+----------+
```

```
     matrix            [0]        [1]        [2]        [3]
  +----------+     +----------+----------+----------+----------+
[0]|   ref *--+---->|    1     |    2     |    3     |    4     |
  +----------+     +----------+----------+----------+----------+
[1]|   ref *--+---->|    5     |    6     |    7     |    8     |
  +----------+     +----------+----------+----------+----------+
[2]|   ref *--+---->|    9     |    10    |    11    |    12    |
  +----------+     +----------+----------+----------+----------+
```

Enhanced For Loop

Java provides another type of For Loop which the documentation calls an "enhanced for statement", but is colloquially referred to as a "For-Each Loop", that iterates over every element in a provided Array or *Iterable* object. For the time being, we'll ignore the latter and focus on the former. Given an array of five elements, I can use an enhanced for loop like so:

Listing 20: Enhanced for loop

```
1:  public class EnhancedFor{
2:      public static void main(String[] args){
3:            int arr[] = new int[]{1,2,3,4,5};
4:            for(int i: arr)
5:                System.out.printf("%d\n", i);
6:      }
7:  }
```

Running and compiling we get:

Listing 21: Enhanced for loop result

```
josephraskind@stargazer:/tmp/arrays$ java EnhancedFor.java
1
2
3
4
5
```

How does this work? Looking at the syntax we find:

Listing 22: Enhanced for loop syntax

```
for(localVariableDeclaration : Expression)
    statement
```

Where `localVariableDeclaration` is a variable whose type matches to the `Expression` on the right hand side of the `:`. Each loop represents a subsequent element in the Array until all elements have been exhausted. It is similar to the basic Python `for` loop.

Note: It is best practice to *always* use an enhanced for loop whenever possible as it presents indexing errors by definition.

Exercises

1. Rewrite the power program from Listing 4 using a `for` loop instead of a `while` loop. Then rewrite it again using a `do while` loop. Which version do you find most readable and why?

2. The chapter provides a trace table for Listing 4 showing the value of `i` and `pow` at the end of each loop. Construct a similar trace table by hand for the following program, then verify by running it:

```
int x = 2;
int y = 8;
int pow = 1;
int i = 0;
while(i < y){
    pow = pow * x;
    i = i + 1;
}
```

3. Write a `for` loop that computes the sum of all integers from 1 to 100. Then write the equivalent using a `while` loop. Verify both produce the same result.
4. The chapter states that `for(;;)` is equivalent to `while(true)`. Write a Java program using each form that prints "looping..." exactly 5 times using a `break` statement to exit. Verify both behave identically.

5. Modify the even sum program from Listing 14 to instead sum all odd numbers from 1 to 100 using `continue`. What change do you need to make to the conditional? Verify your result.
6. The chapter notes that the compiler does not warn about infinite loops. Write a Java program containing an infinite loop as shown in Listing 5, compile it with `javac`, and confirm no warning is produced. Then use `Ctrl+C` in the terminal to kill it. What does `Ctrl+C` do at the OS level?
7. Write a `for` loop using the `break` keyword that finds the first integer greater than 1 whose square exceeds 500. Print both the integer and its square.
8. The chapter shows the bytecode output for the `while` loop in Listing 6. Compile the `for` loop version of the power program from Listing 11 with `javac` and inspect the bytecode with `javap -c`. Compare the output to Listing 6. Are they identical? What does this tell you about the relationship between `for` and `while` loops at the bytecode level?

9. Using nested loops as shown in Listing 12, write a Java program that prints the following multiplication table:

```
1  2  3  4  5
2  4  6  8  10
3  6  9  12 15
4  8  12 16 20
5  10 15 20 25
```

10. The chapter states that all three expressions in a `for` loop are optional, as shown in Listing 10. Write three separate Java programs demonstrating this: one with `expr1` omitted, one with `expr3` omitted, and one with both omitted. In each case, what change must you make elsewhere to keep the program correct?
11. Declare a 3x4 `int` matrix as in Listing 19 and initialize it with values 1 through 12 using nested `for` loops. Then print every element using nested enhanced for loops. Compare the two loop styles—what does the enhanced for loop not give you access to that the regular `for` loop does?
12. Write a Java program that declares an `int` array of 10 elements and sets each element to its index multiplied by 2 using a regular `for` loop. Then use an enhanced for loop to print all elements. Why can you not use an enhanced for loop for the initialization step?
13. The chapter states that Java performs runtime bounds checking on arrays. Write a Java program that deliberately accesses an index beyond the end of an array and record the exception that is thrown. Then write the equivalent in terms of what would happen in C—why is Java's behavior safer? (If you do not have experience with C, then reflect on what might happen.)

Footnotes:

1 "Hard coding" refers to writing logic directly into source code with little to no abstractions.

2 Although, they are not *semantically* equivalent. In the while loop version, `i` is declared outside of the loop block and therefore is visible after the loop finishes. In the for loop version, `i` is declared in the declaration expression and therefore is only visible within the for loop. From the perspective of the program output, this is splitting hairs. From the perspective of understanding the language runtime, this is an important distinction.

Objects & Classes

Introduction

> *“ For, since all things that exist are only particulars, how come we by general terms; or where find we those general natures they are supposed to stand for? Words become general by being made the signs of general ideas: and ideas become general, by separating from them the circumstances of time and place, and any other ideas that may determine them to this or that particular existence. By this way of* ***abstraction*** *they are made capable of representing more individuals than one; each of which having in it a conformity to that abstract idea, is (as we call it) of that sort." — John Locke, An Essay Concerning Human Understanding (§ 3.3.6)*

Abstraction is the most consequential mental power that we have as human beings. It allows us to generate universal ideas from particular phenomena. For example, as I write this paragraph I am wearing clothes. In fact, I'm thinking of a certain article of clothing. As clothing, its function is to cover my body. This article of clothing is often referred to as a t-shirt as its shape resembles the letter *T*. The t-shirt I'm wearing is grey, it is made of 100% cotton, and has a brand logo affixed to it. There were likely others manufactured with an identical design, but this is the only one covering me at this intersection of time and space.

What we've done above is go from a **universal** idea to a **particular** phenomena. There are many sorts of things which could be considered clothing, but there's only one thing that I am currently wearing. Categorizing things is often very convenient. When I write "t-shirt", you know exactly what I mean because you have in your head the general concept of what a "t-shirt" is. The mental image may not immediately have a color, weight, or texture associated with it but it does have a *shape* as the shape is what the concept represents. You also know that the "t-shirt" is an article of *clothing* which denotes a concept of "covering oneself". We constantly use *words* to represent *concepts of things* and not the *things themselves* which includes both their attributes and their functionality.

In Java, the concept of a t-shirt would be called a *class*, and the specific grey cotton one I am wearing right now would be called an *object*.

Objects in Java

Object-Oriented Programming (OOP) languages leverage the innate human power of abstraction to allow for programmers to define their own types called **objects**. The idea surrounding objects is simple: all general concepts of things have shared characteristics and behaviours. For example, a car will be a certain color (characteristic) and will be able to accelerate (behavior). However, the color and acceleration of a car depends on the specific *instance* of the actual car itself. We can define a simple Car object below:

Listing 1: Car class definition

```
1:  class Car{
2:      int color;
3:      void accelerate(){
4:          // Speed up...
5:      }
6:      Car(int c){
7:          color = c;
8:      }
9:  }
```

Here we have a Car `class` definition with one **field**, `color`, and one **method**, `accelerate()` (I will explain the `class` syntax in greater detail below). I can **instantiate** two Car *objects* like so:

Listing 2: Car object instantiation

```
1:  Car red_car  = new Car(0xFF0000); // Red Car
2:  Car blue_car = new Car(0x0000FF); // Blue Car
```

"Instantiation" refers to creating an **instance** of a class in the form of an **object**. The class definition of `Car` holds the blueprints for the creation of a `Car` object whose reference is stored in the variable `red_car`.

Why Use Classes?

Classes are incredibly useful when programming in Java. They allow for programmers to define type-specific logic to methods that would not be aware of it otherwise. For example, we could utilize an array as if it were a derived type like `Frac` as follows:

Listing 3: Using an array as a class stand-in

```
 1:  public class FracArray {
 2:      static void printFrac(int[] arr){
 3:          System.out.printf("%d/%d%n", arr[0], arr[1]);
 4:      }
 5:
 6:      public static void main(String[] args){
 7:          int[] frac1 = {1, 2};
 8:          int[] frac2 = {3, 4};
 9:          String[] frac3 = {1, 2, 3, 4, 5};
10:
11:          printFrac(frac1);
12:          printFrac(frac2);
13:          printFrac(frac3); // Does this compile?
14:      }
15:  }
```

The above will because `printFrac` expects an `int[]` and is provided exactly that three times. However, it has no way of enforcing that the array truly represents a fraction—again, nothing stops a programmer from passing `{1, 2, 3, 4, 5}` and having only the first two elements used silently. The method has no type information telling it what the array is *supposed to represent*.

If we instead define a proper `Frac` class:

Listing 4: Using a class

```
 1:  public class Frac {
 2:      int n;
 3:      int d;
 4:
 5:      Frac(int num, int den){
 6:          n = num;
 7:          d = den;
 8:      }
 9:
10:      void print(){
11:          System.out.printf("%d/%d%n", n, d);
12:      }
13:
14:      public static void main(String[] args){
15:          Frac frac1 = new Frac(1, 2);
16:          Frac frac2 = new Frac(3, 4);
17:
18:          frac1.print();
19:          frac2.print();
20:      }
21:  }
```

Compiling and running:

Listing 5: Using a class results

```
josephraskind@stargazer:/tmp/class$ javac Frac.java; java Frac
1/2
3/4
```

Now `Frac` carries its own type information. The compiler knows exactly what a `Frac` is, what fields it contains, and what operations are valid on it. Passing anything other than a `Frac` to a method expecting one is a hard compile-time error rather than a silent misbehavior. The advantage of classes is that they bundle data and behavior together under a named type that the compiler can reason about and enforce.

Defining a Class

The syntax for defining a class is rather simple:

```
access_modifier class ClassName{
    fields;
    methods()
}
```

Classes can be tagged with "access modifiers" which denote the "visibility" of the class[1]. After the `class` keyword is used the name of the class can then by declared. In Java, the typical convention is to use "Upper CamelCase" or "PascalCase" where every word is capitalized (i.e. "buffered input stream" becomes "BufferedInputStream"). The scope symbols, `{}`, are use to define all of the fields and methods that are within the scope of the class definition.

Instantiating an Object

In Listing 2 you saw how to instantiate an object, but let's break down exactly what's going on. The second half of the assignment statement is the instantiation expression. The instantiation is denoted by the `new` keyword. `new` is used to create a new object instance of a defined class. What follows after `new` must be the name of a valid class and arguments that are supplied to the class' **constructor** (I will explain what a constructor is in the next chapter). If everything checks out from the compiler's end, when the program is run the class definition will be used to create an instance of the object in memory which represents that blueprint.

Objects contra Primitives

Later on we will discuss how Java differentiates object reference types and primitives. For now, we can stick to a conceptual understanding of their differences. Objects represent complicated derived types which are composed of attributes in the form of primitives and other objects, and behaviours in the form of methods. Simply put, objects are more complex, fully featured types relative to primitives. As an example, lets take the standard library defined `String` class vs the `char` primitive:

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

        // char: a primitive, just a single character value
        char c = 'H';

        // char has no methods --- these would not compile:
        // c.toUpperCase();
        // c.isLetter();

        // String: an object built from chars, with rich behavior
        String s = "Hello, World!";

        System.out.println(s.length());                 // 13
        System.out.println(s.toUpperCase());            // HELLO, WORLD!
        System.out.println(s.contains("World"));        // true
        System.out.println(s.replace("World", "Java")); // Hello, Java!
        System.out.println(s.substring(7, 12));         // World
        System.out.println(s.charAt(0));                // H --- gives back a char
        System.out.println(s.isEmpty());                // false
        System.out.println(s.startsWith("Hello"));      // true

        // char is just a number under the hood
        System.out.println((int) c);                    // 72
    }
}
```

We can see that the `String` class provides quite a lot of functionality over the simple `char` primitive.

Exercises

1. The chapter draws an analogy between classes and objects using a t-shirt. Think of your own real-world concept and identify: (a) what the *class* would be, (b) at least three *fields* that instances of that class would have, and (c) at least two *behaviors* (methods) that instances could perform. Write your answer in plain English before attempting any code.
2. Looking at Listing 1, the `Car` class has one field (`color`) and one method (`accelerate`). Extend the analogy: what other fields and methods would a more complete `Car` class need? List at least three of each and describe in plain English what each field stores and what each method does.

3. Compile and run the `Frac` class from Listing 4. Then attempt to add the following line to `main` and recompile:

   ```java
   Frac frac3 = new Frac("hello", "world");
   ```

   Record the compiler error. In your own words explain why Java refuses this and how this demonstrates the advantage of classes over plain arrays as shown in Listing 3.

4. Compile and run the `CharVsString` example from the chapter. Then attempt to add the following lines and recompile:

```
char c = 'H';
c.toUpperCase();
```

   Record the error. Why can you call `toUpperCase()` on a `String` but not on a `char`? What does this tell you about the difference between primitives and objects?

5. The chapter states that a class is a *blueprint* and an object is an *instance* of that blueprint. In Listing 2, two `Car` objects are created from the same `Car` class. What do `red_car` and `blue_car` have in common and how do they differ? Could you create a third `Car` object from the same class without modifying the class definition?
6. The chapter uses the word "instantiation" to describe creating an object. Break down the line `Car red_car = new Car(0xFF0000)` piece by piece and explain what each part does: `Car` (left of `=`), `red_car`, `new`, `Car(0xFF0000)`.
7. Python uses the same dot notation as Java for accessing object behavior — for example `"hello".upper()` in Python is equivalent to `"hello".toUpperCase()` in Java. List three `String` methods from the chapter example and find their Python `str` equivalents. Are there any methods in Java's `String` that Python's `str` does not have, or vice versa?
8. The chapter states that `char` is "just a number under the hood" and demonstrates this by casting `'H'` to `int` to get `72`. Using what you know from the data chapter about ASCII encoding, verify this by hand. Then write a short Java program that prints the integer value of at least five different characters and verify each against the ASCII table.

Footnotes:

1 I explain access modifiers in greater detail in the next chapter

Instance Fields & Methods

Introduction

Recall that objects in Java have both *characteristics* and *behaviors*. Within the Java language we can represent characterstics through the use of *fields*, or variables, and behaviors through the use of *methods*.

Note: For the following chapter, I take for granted your familiarity with *functions* in the computer science sense of the term. If you are unfamiliar with functions then I suggest you check out this chapter written for a C class which goes over the basics.

What is an Instance?

An instance refers to the in-memory representation of an object. Take the following class definition:

Listing 1: The BankAccount class

```
1:    public class BankAccount {
2:      String owner;
3:      double balance;
4:
5:      BankAccount(String owner, double initialBalance){
6:          this.owner   = owner;
7:          this.balance = initialBalance;
8:      }
9:
10:     void deposit(double amount){
11:         balance = balance + amount;
12:     }
13:
14:     void withdraw(double amount){
15:         balance = balance - amount;
16:     }
17:
18:     void printBalance(){
19:         System.out.printf("%s's balance: $%.2f%n", owner, balance);
20:     }
21:
22: }
```

Note: The this keyword makes explicit the reference to the instance which is being used for the method invocation.

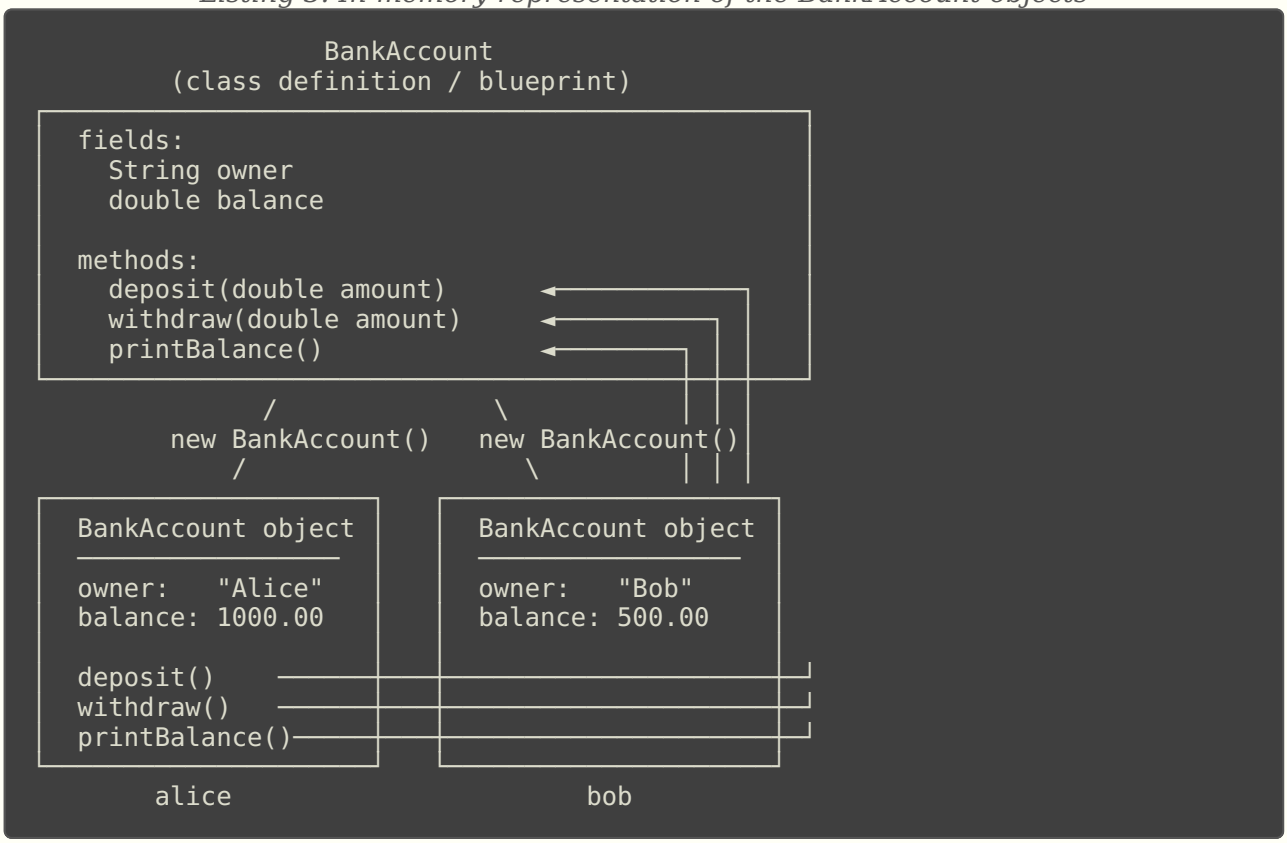
The `BankAccount` class has two *instance* fields (`owner` and `balance`) and three *instance* methods (`deposit`, `withdraw`, and `printBalance`). We can create two instances of the class with the following:

Listing 2: Instantiating BankAccount objects

```
1:   BankAccount alice = new BankAccount("Alice", 1000.00);
2:   BankAccount bob   = new BankAccount("Bob",   500.00);
```

The variables `alice` and `bob` store two distinct *instances* of the `BankAccount` class in the form of in-memory **objects**. We can represent these objects pictorially like so:

Listing 3: In-memory representation of the BankAccount objects



We can see that there are two `BankAccount` objects with their own, distinct internal fields. We can also see, however, that the methods defined in the class are all shared by the instances. This is because `alice` and `bob` are both *instances* of the `BankAccount` class and as such they have the same characteristics (i.e. fields) and the same behaviors (i.e. methods). Characteristics may change from object to object, but behaviors do not (a bank account may have such and such amount, but it should always be able to take in new deposits).

Instance Fields

Fields are the variables tied to a particular class. *Instance* fields are specifically the fields which are tied to *instances* of an object, not shared between the entire class type. Unless otherwise specified (with the `static`

keyword), all fields are instance fields. In Listing 1 there are two instance fields `owner` and `balance`. Just like how we defined local variables in the past, instance fields have types attached to them. `owner` is a `String`, itself an object type, and `balance` is a primitive `double`. These fields can be freely referenced within the class. Let's take a look at some code which makes use of the `BankAccount` class:

Listing 4: A program that uses BankAccount

```
1:   public class BankUser{
2:      public static void main(String[] args){
3:           BankAccount acc = new BankAccount("Alice", 1000.00);
4:           System.out.printf("acc's owner: %s\n", acc.owner); // Using . to
reference instance field
5:       }
6:   }
```

I can compile and run this code to get the following:

Listing 5: A program that uses BankAccount results

```
1:   josephraskind@stargazer:/tmp/instance$ ls
2:   BankAccount.java  BankUser.java
3:   josephraskind@stargazer:/tmp/instance$ javac BankUser.java; java BankUser
4:     acc's owner: Alice
```

Note: This only compiles because I have both .java files in the same directory.

How did this work? `BankAccount` defined an instance field, `owner`, which was referenced by `BankUser` in its main method on Line 4. We can access instance fields and methods using the `.` notation. `acc.owner` tells the JVM to find the object tied to the `acc` variable and return the value of the field called `owner`. Let's see what happens when I have two variables:

Listing 6: A program that uses BankAccount with two accounts

```
1:   public class BankUser{
2:      public static void main(String[] args){
3:           BankAccount acc_1 = new BankAccount("Alice", 1000.00);
4:           BankAccount acc_2 = new BankAccount("Bob", 500.00);
5:           System.out.printf("acc_1's owner: %s\n", acc_1.owner);
6:           System.out.printf("acc_2's owner: %s\n", acc_2.owner);
7:       }
8:   }
```

After compiling and running:

Listing 7: A program that uses BankAccount with two accounts results

```
1:   josephraskind@stargazer:/tmp/instance$ javac BankUser.java; java BankUser
2:   acc_1's owner: Alice
3:   acc_2's owner: Bob
```

How are there two different values being printed for the calls to `.owner`? Again, this has to do with the fact that there are *two different instances* of `BankAccount`. `acc_1` is tied to Alice and `acc_2` is tied to Bob. `acc_1` and `acc_2` are both `BankAccount` *objects*, but they are *not the same* `BankAccount` object.

Instance Methods

Classes define behaviors, alongside characteristics. In Java those behaviors are referred to as `methods`[1]. Let's update our `BankUser` to include a call to, or **invocation** of, the `printBalance` method:

Listing 8: BankUser invoking printBalance

```
1:   public class BankUser{
2:      public static void main(String[] args){
3:           BankAccount acc_1 = new BankAccount("Alice", 1000.00);
4:           BankAccount acc_2 = new BankAccount("Bob", 500.00);
5:           acc_1.printBalance();
6:           acc_2.printBalance();
7:       }
8:   }
```

After compiling and running:

Listing 9: BankUser invoking printBalance results

```
josephraskind@stargazer:/tmp/instance$ javac BankUser.java; java BankUser
Alice's balance: $1000.00
Bob's balance: $500.00
```

We've already established that each instance has its own internal values associtated with the fields tied to the class definition, so it makes sense that we see `Alice` and `1000`. But why do those calls to `printBalance` work? They work because `acc_1` is an instance of a `BankAccount` object and by the class definition it has the `printBalance` method bound to it. For example, if I wrote `acc_1.printBALance` instead, we would encounter a compilation error as no method with that name is guaranteed to exist for the `BankAccount` type.

But how does `printBalance` "know" the values of `owner` and `balance`? Both `owner` and `balance` are tied to the instance of a `BankAccount` class, so by definition there will be an `owner` and a `balance` field for each instance. When the `printBalance` method is invoked it is invoked with *context*—the context of which instance it was invoked from. Line 5 invokes `printBalance` with `acc_1` which we know contains a `BankAccount` object whose fields contain the values `Alice` and `1000`; likewise, Line 6 invokes `printBalance` with `acc_2` which we know contains a `BankAccount` object whose fields contain the values `Bob` and `500`.

Let's see what happens when we call `deposit` on `acc_1`, but not on `acc_2`:

Listing 10: BankUser invoking deposit

```
 1:   public class BankUser{
 2:      public static void main(String[] args){
 3:           BankAccount acc_1 = new BankAccount("Alice", 1000.00);
 4:           BankAccount acc_2 = new BankAccount("Bob", 500.00);
 5:           acc_1.printBalance();
 6:           acc_2.printBalance();
 7:           acc_1.deposit(500);
 8:           System.out.println("After Deposit:");
 9:           acc_1.printBalance();
10:           acc_2.printBalance();
11:       }
12:   }
```

After compiling and running:

Listing 11: BankUser invoking deposit results

```
josephraskind@stargazer:/tmp/instance$ javac BankUser.java; java BankUser
Alice's balance: $1000.00
Bob's balance: $500.00
After Deposit:
Alice's balance: $1500.00
Bob's balance: $500.00
```

We can see that `500` was added to `acc_1`'s `1000`, but not `acc_2`'s `500`. Why not? Again, this is because `acc_1` is a totally distinct instance of `BankAccount` from `acc_2`. Even though they both "share code" in the form of method definitions, they do not share the same runtime information in regards to their instance field values.

Note: We will learn later that some fields *can* be shared. Those fields are called *static* fields.

Constructors

A **constructor** is a special function which details how an object will be instantiated. In Listing 1 the constructor is defined between Lines 5 and 8. Constructors are always named after the class and never have a return type. When the `new` keyword is used to instantiate an object, the constructor associated with that object is invoked. For example, when we write `new BankAccount("Charlene", 211000)` we are calling the `BankAccount` constructor. Constructors are primarily used to set up useful contexts for a new object or set instance fields (the `BankAccount` class does the latter).

When no constructor is defined, a default constructor will be created by the Java compiler. The default constructor will leave all instance fields to their default values.

Note: Constructors are often used to enforce type invariants/contracts. We will learn how we can do so using exceptions.

Encapsulation

OOP came from a desire to attribute greater functionality to programmer-defined types. One such aspect is the ability to **encapsulate** data by restricting its visibility to prevent, wittingly or unwittingly, malicious actors from ruining a class's assumptions. This is achieved in Java through the use of **access modifiers**.

Access Modifiers

So far, you've likely seen the access modifier `public` the most. There are four access modifiers in Java with varying levels of privilege associated with them:

Modifier	Class	Package	Subclass	World	Description
`public`	√	√	√	√	Accessible from anywhere
`protected`	√	√	√	✗	Accessible within the same package and by subclasses
(default)	√	√	✗	✗	Accessible only within the same package (no keyword required)
`private`	√	✗	✗	✗	Accessible only within the class it is declared in

Note: All classes, fields and methods that do not specify a modifier are assigned "default".

The reason for why this might be desireable can be quickly illustrated with the simple `BankAccount` class which has been slightly edited:

Listing 12: The BankAccount class with negative protection

```
 1:     public class BankAccount {
 2:       String owner;
 3:       double balance;
 4:
 5:       BankAccount(String owner, double initialBalance) throws
RuntimException{
 6:           this.owner   = owner;
 7:           this.balance = initialBalance;
 8:           if(balance < 0)
 9:               throw new RuntimeException("Cannot have negative balance!");
10:       }
11:
12:       void deposit(double amount){
13:           balance = balance + amount;
14:       }
15:
16:       void withdraw(double amount){
17:           balance = balance - amount;
18:       }
19:
20:       void printBalance(){
21:           System.out.printf("%s's balance: $%.2f%n", owner, balance);
22:       }
23:
24:   }
```

Now, it is impossible to instantiate the `BankAccount` class with a negative number for the balance:

Listing 13: BankUser creating account with negative balance

```
1:   public class BankUser{
2:       public static void main(String[] args){
3:           BankAccount acc = new BankAccount("Alice", -1000.00);
4:           acc.printBalance();
5:       }
6:   }
```

After compiling and running:

Listing 14: BankUser creating account with negative balance result

```
josephraskind@stargazer:/tmp/instance$ javac BankUser.java; java BankUser
Exception in thread "main" java.lang.RuntimeException: Cannot have negative
balance!
        at BankAccount.<init>(BankAccount.java:9)
        at BankUser.main(BankUser.java:3)
```

But since balance is left as a `default` field, it can easily be manipulated to violate the invariant:

Listing 15: BankUser updating account with negative balance

```
1:   public class BankUser{
2:       public static void main(String[] args){
3:           BankAccount acc = new BankAccount("Alice", 1000.00);
4:           acc.balance = -1000.00;
5:           acc.printBalance();
6:       }
7:   }
```

After compiling and running:

Listing 16: BankUser creating account with negative balance result

```
josephraskind@stargazer:/tmp/instance$ javac BankUser.java; java BankUser
Alice's balance: $-1000.00
```

This can easily be solved by flipping `balance` to `private`:

Listing 17: The BankAccount class with negative protection

```
1:     public class BankAccount {
2:       String owner;
3:       private double balance;
4:       //...
5:   }
```

After compiling:

Listing 18: BankUser creating account with negative balance result after private field set

```
josephraskind@stargazer:/tmp/instance$ javac BankUser.java; java BankUser
BankUser.java:4: error: balance has private access in BankAccount
        acc.balance = -1000.00;
           ^
1 error
```

The compiler tells us that `balance` is no longer accessible. This is because it is now *restricted* to being referenced inside of the `BankAccount` class definition.

Exercises

1. In your own words, explain the difference between a **class** and an **instance**. Using the `BankAccount` example from the chapter, identify which parts of the code represent the class definition and which parts represent instances.
2. Compile and run Listing 10. Then add a call to `acc_2.deposit(250)` after the existing deposit and print both balances again. Does the deposit to `acc_2` affect `acc_1`? Explain why or why not in terms of instances.
3. Add a `withdraw` call to Listing 10 that withdraws $2000 from `acc_1`. What is the resulting balance? Does the `BankAccount` class prevent

this? What would need to change in the class definition to prevent a negative balance from occurring through a withdrawal?

4. Listing 17 prevents instantiation with a negative balance but Listing 15 shows the invariant can still be broken. Explain in your own words why making `balance` `private` solves this problem. What does `private` prevent that the constructor check alone cannot?
5. The chapter states that when no constructor is defined, the Java compiler creates a default one. Remove the constructor from `BankAccount` and attempt to instantiate it with `new BankAccount("Alice", 1000.00)`. What error do you get? Then instantiate it with `new BankAccount()` and print `acc.owner`. What value is printed and why?
6. The chapter states that methods are invoked *with context*. In your own words explain what this means. When `acc 1.printBalance()` is called, how does `printBalance` know to print `Alice` and `1000.00` rather than `Bob` and `500.00`?
7. The access modifier table shows four levels of privilege. Rank them from most to least restrictive and give a real-world analogy for each — for example, `private` is like a personal diary that only you can read.
8. Add a `transfer` method to `BankAccount` that takes another `BankAccount` object and a `double` amount as arguments and transfers that amount from the current account to the other. Test it by transferring $200 from `acc 1` to `acc 2` and printing both balances before and after.
9. The chapter introduces the `.` notation for accessing instance fields and methods. Write a program that creates a `BankAccount`, accesses its `owner` field directly, calls `printBalance`, and then attempts to access a field that does not exist (e.g. `acc.phone`). Record the compiler error and explain what it tells you.
10. Make both `owner` and `balance` `private` in `BankAccount`. Now `BankUser` can no longer access `acc.owner` directly. Write a method `getOwner` in `BankAccount` that returns the value of `owner` and update `BankUser` to use it instead. This pattern is called a **getter** — why might this be preferable to leaving `owner` as a `default` field?
11. The chapter states "a bank account may have such and such amount, but it should always be able to take in new deposits." Give two more real-world examples of objects where characteristics vary between instances but behaviors remain the same. For each, identify at least two fields and two methods.
12. Create a second class called `SavingsAccount` that has an additional field `interestRate` of type `double`. Add a method `applyInterest` that multiplies `balance` by `(1 + interestRate)`. Instantiate one `SavingsAccount` and one `BankAccount` and demonstrate that `applyInterest` cannot be called on a `BankAccount` object.
13. The chapter notes that constructors are "always named after the class and never have a return type." Try adding a return type of `void` to the `BankAccount` constructor and recompile. What happens? Then try naming the constructor `bankAccount` (lowercase b) instead. What happens?
14. Consider the following code snippet:

```
BankAccount a = new BankAccount("Alice", 500.00);
BankAccount b = a;
b.deposit(200.00);
a.printBalance();
```

Before running it, predict what `a.printBalance()` will print. Then run it and verify. Does the result surprise you? What does this tell you about how object variables work in Java? We will explore this further in the chapter on references.

15. The `BankAccount` constructor uses the `this` keyword on Lines 6 and 7 of Listing 1:

```
this.owner   = owner;
this.balance = initialBalance;
```

Remove the `this` keyword from Line 6 so it reads `owner = owner` and recompile. Does it compile? Run it and call `printBalance`. What value is printed for the owner and why? Now restore `this.owner = owner` and explain in your own words what `this` refers to and why it is necessary when a parameter name matches a field name.

Footnotes:

1 Other languages, like Python and C, call them *functions*.

References

Introduction

When discussing types, we mentioned the fact that Java makes a distinction between two categories of types in the language: **primitive** types and **reference** types. At the time, we handwaived the difference and focused primarily on the primitive types provided by Java (byte, short, int, long, float, double, char, and boolean). In this chapter, we will take a look at what makes *reference* types so different as to justify its own category.

What Makes a Type Primitive?

Before we investigate reference types we should first recall what makes a primitive type primitive. The idea is simple, primitive types are "predefined by the Java programming language and named by its reserved keyword" (§ 4.2). Structurally, from the point of view of Java, primitive types are distinct from reference types insofar as the primitive types represent the *atomic stuff* of the language. Furthermore, the *values* of primitive types are predefined:

> *The numeric types are the integral types and the floating-point types.*
>
> *The integral types are byte, short, int, and long, whose values are 8-bit, 16-bit, 32-bit and 64-bit signed two's-complement integers, respectively, and char, whose values are 16-bit unsigned integers representing UTF-16 code units (§3.1).*
>
> *The floating-point types are float, whose values exactly correspond to the 32-bit IEEE 754 binary32 floating-point numbers, and double, whose values exactly correspond to the 64-bit IEEE 754 binary64 floating-point numbers.*
>
> *The boolean type has exactly two values: true and false.*

A primitive type is one of eight possible types given by the language, and each can only contain values within their respective ranges. We also find one other clause within the specification about primitive types:

> *Primitive values do not share state with other primitive values*

This is a subtle point, but one that has massive implications. A code example will illustrate the idea:

Listing 1: Primitive values do not share state

```
1:  public class PrimitiveState{
2:      public static void main(String[] args){
3:          int x = 10;
4:          int y = x;
5:          x += 1;
6:          System.out.printf("x: %d%n", x);
7:          System.out.printf("y: %d%n", y);
8:      }
9:  }
```

What would you expect Listing 1 to print out? If we compile and run it we get the following:

Listing 2: Primitive values do not share state results

```
josephraskind@stargazer:/tmp/references$ javac PrimitiveState.java; java
PrimitiveState
x: 11
y: 10
```

We see that `x` contains the incremented value and `y` does not. This is because the state of a primitive value is not passed on to another variable during assignment. The `int` that `x` is tied to is distinct from the `int` that `y` is tied to because only its *value* was passed during the assignment statement. We have two distinct *variables* with two distinct *values*. We'll see that reference types do not share this behaviour!

What is a Reference Type?

By definition, a reference type is any type that is **not** a primitive type (i.e. anything that is not a byte, short, int, long, float, double, char, or boolean). We know now that types which we define through the use of classes are called **objects** when instantiated. Therefore, all objects in Java are reference types —this is also true of all arrays. What makes reference types special is their values "are pointers to these objects" (§ 4.3.1). What does that mean? A code example might be more instructive:

Listing 3: Reference values share state

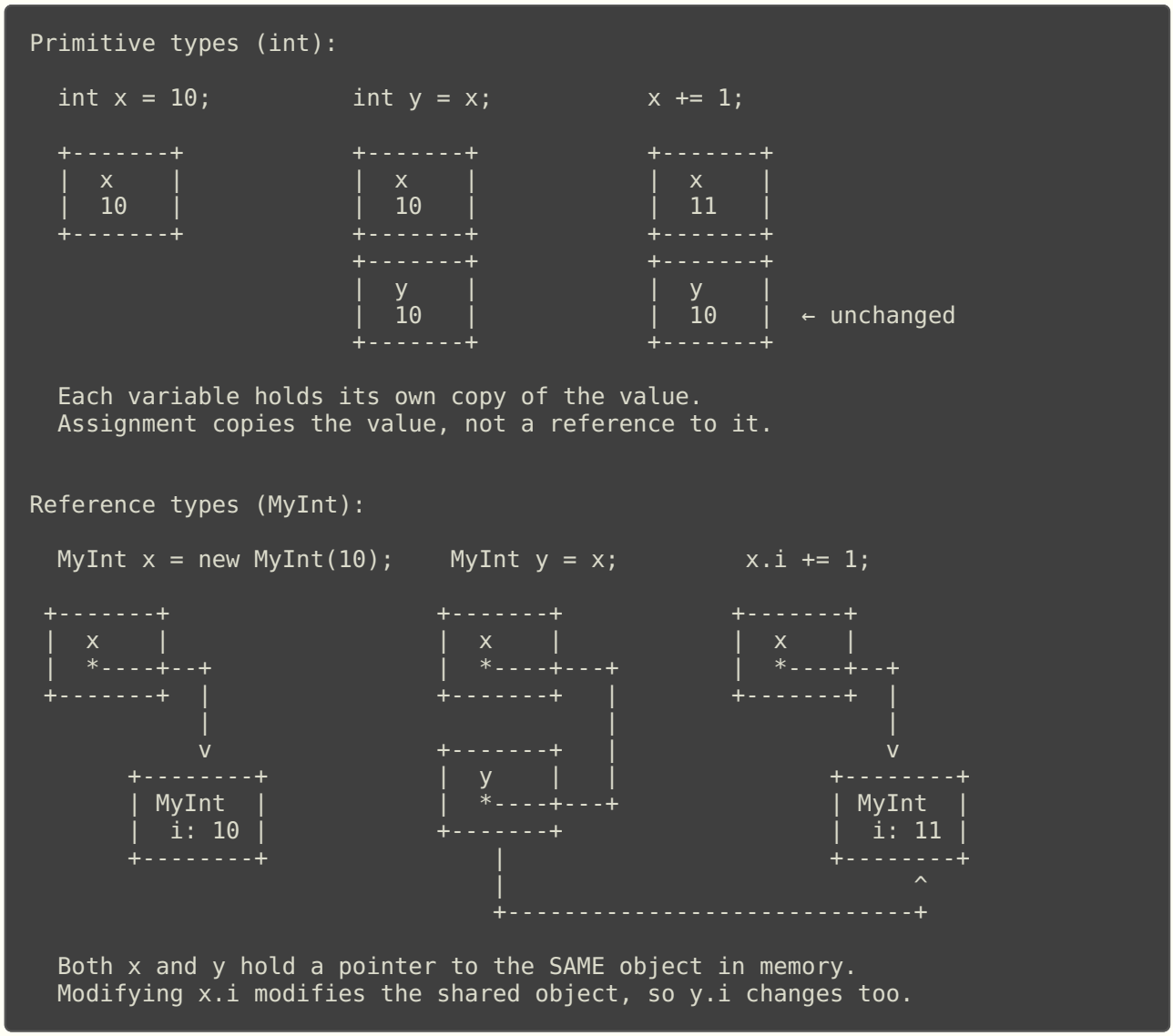
```
 1:  class MyInt{
 2:      int i;
 3:      MyInt(int i){
 4:          this.i = i;
 5:      }
 6:  }
 7:
 8:  public class ReferenceState{
 9:      public static void main(String[] args){
10:          MyInt x = new MyInt(10);
11:          MyInt y = x;
12:          x.i += 1;
13:          System.out.printf("x: %d%n", x.i);
14:          System.out.printf("y: %d%n", y.i);
15:      }
16:  }
```

You can see all I've done is switch out the primitive `int` for an object called `MyInt` which acts as a wrapper around a basic `int`. It wouldn't be too absurd to assume we would get the same printout of "11\n10", but when we compile and run the code we get:

Listing 4: Reference values share state results

```
josephraskind@stargazer:/tmp/references$ javac ReferenceState.java; java
ReferenceState
x: 11
y: 11
```

How could that be?! This behavior hinges on the fundamental difference between primitive and reference types. Primitive types cannot share state whereas reference types **can**. This is because reference types store *pointers* to the underlying objects in memory. The following diagram shows this phenomenon visually:



Passing References

In Java, when we send in objects as parameters to methods we are always sending in *references*. For example, if we take the BankAccount class from the previous chapter:

Listing 5: Passing references

```
1:  public class PassingReferences{
2:
3:      static void applyBonus(BankAccount acc, double amount){
4:          acc.deposit(amount);
5:      }
6:
7:      public static void main(String[] args){
8:          BankAccount alice = new BankAccount("Alice", 1000.00);
9:
10:         System.out.println("Before bonus:");
11:         alice.printBalance();
12:
13:         applyBonus(alice, 500.00);
14:
15:         System.out.println("After bonus:");
16:         alice.printBalance();
17:     }
18: }
```

Here we can see that this new static method `applyBonus` takes a `BankAccount` object called `acc`. We then access the public method `deposit` and send in the `amount`. In the `main` method we print the bonus before and after a call to `applyBonus`. When we compile and run the above code we get the following output:

Listing 6: Passing references results

```
josephraskind@stargazer:/tmp/references$ javac PassingReferences.java; java
PassingReferences
Before bonus:
Alice's balance: $1000.00
After bonus:
Alice's balance: $1500.00
```

Even though `acc` is a different variable than `alice` **they both point to the same object**.

Immutable Objects

Over-reliance on the use of references can create situations where it becomes difficult to track updates to an object as side effects may occur in a variety of contexts. This has led to the design pattern of crafting **immutable objects**, or objects whose internal state cannot ever be updated. Instead of updating an object's state, a new object will be created to reflect the desired update. For example, we can update the `BankAccount` class to be made immutable:

Listing 7: Immutable BankAccount

```
class BankAccount {
    final String owner;
    final double balance;

    BankAccount(String owner, double initialBalance){
        this.owner   = owner;
        this.balance = initialBalance;
    }

    BankAccount deposit(double amount){
        return new BankAccount(owner, balance + amount);
    }

    BankAccount withdraw(double amount){
        return new BankAccount(owner, balance - amount);
    }

    void printBalance(){
        System.out.printf("%s's balance: $%.2f%n", owner, balance);
    }

}
```

Now if we rerun `PassingReferences`:

Listing 8: Using immutable BankAccount with PassingReferences

```
josephraskind@stargazer:/tmp/references$ javac PassingReferences.java; java
PassingReferences
Before bonus:
Alice's balance: $1000.00
After bonus:
Alice's balance: $1000.00
```

Now the update is not reflected in the `main` method's object!

#+ATTR_HTML :class notes Java's String class is immutable.

The null Value

The Java Language Specification lays out two possible reference values: "pointers to… objects, and a special null reference, which refers to no object". By default, all uninitialized reference types will refer to the `null` value. Again, the `null` value indicates the lack of a particular object associated with a reference type variable. For example, I could create a variable of type `BankAccount` without initializing it:

Listing 9: Default null value for uninitialized reference types

```
1:   public class NullObject{
2:       static BankAccount emptyAcc;
3:       public static void main(String[] args){
4:             System.out.printf("emptyAcc: %s%n", emptyAcc);
5:       }
6:   }
```

When we compile and run the above code we find the following printed:

Listing 10: Default null value results

```
josephraskind@stargazer:/tmp/references$ javac NullObject.java; java NullObject
emptyAcc: null
```

Because the `null` value refers to the lack of an instance tied to a given variable that means we cannot reference instance fields/method:

Listing 11: Referencing instance fields/methods with a null holding variable

```
1:   public class NullObject{
2:       static BankAccount emptyAcc;
3:       public static void main(String[] args){
4:             null.printBalance()
5:       }
6:   }
```

Compiling and running the above we get:

Listing 12: Referencing instance fields/methods with a null holding variable results

```
josephraskind@stargazer:/tmp/references$ javac NullObject.java; java NullObject
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"BankAccount.printBalance()" because "NullObject.emptyAcc" is null
        at NullObject.main(NullObject.java:4)
```

We get a runtime exception! Again, `null` refers to a non-existent object, therefore we cannot dereference it.

Comparing Reference Types

A simple mistake many beginner Java programmers make is to do basic equivalency checks between reference types using the `==` operator. `==` only checks equivalency between stored *values*. This is not a problem when comparing two primitive types as they store *values themselves*. This **is** a problem when comparing two reference types as they store *pointers to objects*. Take a simple equivalancy comparison between two Strings:

Listing 13: Comparing two Strings

```
1:  public class ComparingStrings{
2:      public static void main(String[] args){
3:          String s1 = new String("Dog");
4:          String s2 = new String("Dog");
5:          boolean result = s1 == s2;
6:          System.out.printf("s1 == s2 returns %b%n", s1, s2, result);
7:      }
8:  }
```

When we compile and run the above we get:

Listing 14: Comparing two Strings results

```
josephraskind@stargazer:/tmp/references$ javac ComparingStrings.java; java
ComparingStrings
s1 == s2 returns false
```

What?! Again, this is because what's really going on is that Java is comparing the *pointers* stored in `s1` and `s2` and not the internal values themselves. Many built-in standard library Java objects have an `equals` method for direct equivalency checking:

Listing 15: Comparing two Strings with equals

```
1:  public class ComparingStrings{
2:      public static void main(String[] args){
3:          String s1 = new String("Dog");
4:          String s2 = new String("Dog");
5:          boolean result = s1.equals(s2);
6:          System.out.printf("s1.equals(s2) returns %b%n", s1, s2, result);
7:      }
8:  }
```

When we compile and run the above:

Listing 16: Comparing two Strings with equals results

```
josephraskind@stargazer:/tmp/references$ javac ComparingStrings.java; java
ComparingStrings
s1 == s2 returns false
```

That's better!

Note: Whenever you write your own classes in Java, make sure that you write your own methods for direct comparisons!

Exercises

1. Compile and run Listing 1. Then change `int` to `double` and repeat the experiment with decimal values. Does the behavior change? What does this confirm about all primitive types?
2. Compile and run Listing 3. Then add a third variable `MyInt z = y` and increment `z.i` by 5. Print `x.i`, `y.i`, and `z.i`. Before running, predict each value. Explain the result in terms of references.
3. The chapter draws a distinction between copying a value and copying a reference. In your own words, explain what happens in memory when you write `int y = x` versus `MyInt y = x`. Use the ascii diagrams from the chapter as a guide.
4. Compile and run Listing 5. Then modify `applyBonus` to also print the balance of `acc` inside the method before returning. Does the value printed inside `applyBonus` match the value printed in `main` after the call? What does this confirm about references being passed to methods?
5. The chapter states that even though `acc` and `alice` are different variables, they point to the same object. Write a program that demonstrates this by passing `alice` to a method that calls `withdraw` rather than `deposit`. Verify that the withdrawal is reflected in `main` after the method returns.
6. Replace the mutable `BankAccount` with the immutable version from Listing 7 and rerun Listing 5. The balance no longer updates in `main`. Modify `applyBonus` so that it returns the new `BankAccount` object produced by `deposit` and update `main` to capture and print the returned object. What does this pattern require the programmer to do differently compared to the mutable version?
7. The chapter states that Java's `String` class is immutable. Write a program that demonstrates this by attempting to change a character in a `String` using a method call. What happens? Now create a new `String` using concatenation and assign it back to the original variable. Does the original `String` object change, or is a new one created?
8. Compile and run Listing 9. Then attempt to call `emptyAcc.deposit(100)` instead of printing it. Record the exception. In your own words explain what a `NullPointerException` is and why it occurs. How could you guard against it before calling a method on a reference type variable?
9. Compile and run Listing 13. Then change `new String("Dog")` to just `"Dog"` for both `s1` and `s2` and rerun. Does `==` now return `true`? Research why this happens (hint: look up "string interning" in Java) and write a short explanation.
10. Compile and run Listing 15. Notice the `printf` format string uses `%b` but passes three arguments (`s1`, `s2`, `result`). What does the output look like? Fix the format string so it correctly prints the result and recompile.

11. Add an `equals` method to the `BankAccount` class that returns `true` if two `BankAccount` objects have the same `owner` and `balance`:

```
boolean equals(BankAccount other){
    // your implementation here
}
```

Test it with two accounts that have identical fields and two that differ. Then test what happens when you use `==` instead — explain why the results differ.

12. The chapter explains that the immutable `BankAccount` returns a new object from `deposit` and `withdraw` rather than modifying the existing one. Write a short program that calls `deposit` on an immutable `BankAccount` but does not assign the return value. Print the balance after. What do you observe? What does this tell you about the risk of ignoring return values from immutable objects?
13. Write a method `transfer` that takes two `BankAccount` objects and a `double` amount and transfers that amount from the first account to the second. Use the mutable `BankAccount` defined in the previous chapter. Verify that both balances update correctly after the call.
14. The chapter states that uninitialized reference type fields default to `null`. Write a class with three fields: an `int`, a `String`, and a `BankAccount`. Instantiate the class without setting any fields and print all three. What are their default values? Why does `int` have a different default than the reference types?

15. The following code contains a subtle bug related to references:

```
BankAccount a = new BankAccount("Alice", 500.00);
BankAccount b = a;
b = new BankAccount("Alice", 500.00);
System.out.println(a == b);
System.out.println(a.equals(b));
```

Before running, predict the output of both print statements. Then run it and verify. Explain what happens to the reference stored in `b` after the second line and why `a` is unaffected by the reassignment of `b`.

Overloading

Introduction

Oftentimes when writing classes in Java, we run into situations where a method provides useful functionality, but its implementation should change depending on the arguments sent to it. Java allows us to redefine a method with the same name but a different signature through **overloading**.

How To Overload a Method

Overloading in Java is simple. For a method to be overloaded, the name of the method must be reused and a different set of parameters must be defined. The new set can be a different number of parameters, different types of parameters, or a mixture of both. Take the `Log` class below as an example:

```
 1:   public class Logger {
 2:
 3:       static void log(String message){
 4:           System.out.printf("[LOG] %s%n", message);
 5:       }
 6:
 7:       static void log(String message, String level){
 8:           System.out.printf("[%s] %s%n", level, message);
 9:       }
10:
11:       static void log(String message, String level, int lineNumber){
12:           System.out.printf("[%s] (line %d) %s%n", level, lineNumber,
message);
13:       }
14:
15:       static void log(Exception e){
16:           System.out.printf("[ERROR] %s: %s%n",
e.getClass().getSimpleName(), e.getMessage());
17:       }
18:
19:       public static void main(String[] args){
20:           log("Application started");
21:           log("Disk usage above 90%", "WARNING");
22:           log("Variable x is null", "ERROR", 42);
23:           log(new IllegalArgumentException("Invalid input supplied"));
24:       }
25:   }
```

Lines 3, 7, 11, and 15 all declare a method named `log`. What varies between them are the parameters provided in the method declarations. The first declaration takes one `String`; the second takes two `String`s; the third takes two `String`s and an `int`; the fourth and final takes an `Exception`. The compiler "knows" which `log` to call based on the arguments provided in the method call. Line 20's call corresponds with Line 3's declaration, etc.

Note: This is called *ad hoc polymorphism*.

When Overloading is Useful

Overloading occurs frequently within the Java standard library. This is often because Java programmers favour streamlined, aesthetically cohesive API's over distinctly named methods. For example, some believe it is easier to think about a single `log` method that changes based on argument context than a `logSingleMessage`, `logMessageWithLevel`, `logMessageWithLineNumber`, and `logWithException`.

Overloading Constructors

Overloading often pops up in constructor definitions. For example, the String class has 16 unique constructors! We can see some of them utilized below:

```java
import java.nio.charset.StandardCharsets;

public class AllStringConstructors {
    public static void main(String[] args) throws Exception {

        // 1. String() – empty string
        String s1 = new String();
        System.out.println("1: '" + s1 + "'");

        // 2. String(String original) – copy of another string
        String s2 = new String("Hello");
        System.out.println("2: " + s2);

        // 3. String(char[]) – from char array
        char[] chars = {'J', 'a', 'v', 'a'};
        String s3 = new String(chars);
        System.out.println("3: " + s3);

        // 4. String(char[], int offset, int count) – subarray of chars
        String s4 = new String(chars, 1, 3); // "ava"
        System.out.println("4: " + s4);

        // 5. String(int[] codePoints, int offset, int count) – from
Unicode code points
        int[] codePoints = {72, 101, 108, 108, 111}; // "Hello"
        String s5 = new String(codePoints, 0, codePoints.length);
        System.out.println("5: " + s5);

        // 6. String(byte[]) – from bytes using default charset
        byte[] bytes = "Java".getBytes();
        String s6 = new String(bytes);
        System.out.println("6: " + s6);

        // 7. String(byte[], int offset, int length) – subarray of
bytes, default charset
        String s7 = new String(bytes, 1, 3); // "ava"
        System.out.println("7: " + s7);

        // 8. String(byte[], Charset) – from bytes using specified
Charset object
        String s8 = new String(bytes, StandardCharsets.UTF_8);
        System.out.println("8: " + s8);

        // 9. String(byte[], String charsetName) – from bytes using
charset name
        String s9 = new String(bytes, "UTF-8");
        System.out.println("9: " + s9);

        // 10. String(byte[], int offset, int length, Charset) –
subarray with Charset object
        String s10 = new String(bytes, 0, 4, StandardCharsets.UTF_8);
        System.out.println("10: " + s10);

        // 11. String(byte[], int offset, int length, String
charsetName) – subarray with charset name
        String s11 = new String(bytes, 0, 4, "UTF-8");
        System.out.println("11: " + s11);

        // 12. String(StringBuffer) – from StringBuffer
        StringBuffer sb = new StringBuffer("Hello from StringBuffer");
        String s12 = new String(sb);
        System.out.println("12: " + s12);

        // 13. String(StringBuilder) – from StringBuilder
```

```
59:           StringBuilder sbl = new StringBuilder("Hello from
StringBuilder");
60:           String s13 = new String(sbl);
61:           System.out.println("13: " + s13);
62:       }
63:   }
```

Compiling and running we get:

```
josephraskind@stargazer:/tmp/overloading$ javac AllStringConstructors.java;
java AllStringConstructors
1: ''
2: Hello
3: Java
4: ava
5: Hello
6: Java
7: ava
8: Java
9: Java
10: Java
11: Java
12: Hello from StringBuffer
13: Hello from StringBuilder
```

Exercises

1. The `BankAccount` class from earlier chapters currently requires both an owner name and an initial balance to be provided at construction time. Add a second constructor to `BankAccount` that takes only an owner name and defaults the balance to `0.0`. Write a program that creates one account using each constructor and calls `printBalance` on both. What does this tell you about when overloading constructors is useful?

2. The chapter states that return type alone cannot distinguish two overloads. Attempt to compile the following class and record the error:

   ```
   public class InvalidOverload {
       static int compute(int x){
           return x * 2;
       }
       static double compute(int x){
           return x * 2.0;
       }
   }
   ```

   In your own words, explain why Java cannot use the return type to resolve which method to call. Hint: think about what happens when the return value is not assigned to a variable, as in `System.out.println(compute(5))`.

3. When the compiler must choose between two overloaded methods and an exact match is not available, it will use **widening** to find the

closest compatible type. Write the following class, predict which overload will be called for each invocation, then compile and verify:

```
public class Widening {
    static void print(int x){
        System.out.println("int: " + x);
    }
    static void print(double x){
        System.out.println("double: " + x);
    }
    public static void main(String[] args){
        print(42);       // which overload?
        print(3.14);     // which overload?
        print(42L);      // which overload? (long)
        print((byte) 5); // which overload? (byte)
    }
}
```

What general rule can you infer about how Java resolves overloads when no exact match exists?

4. Add a `deposit` overload to `BankAccount` that accepts an `int` in addition to the existing `double` version. Write a program that calls both overloads and verify the results are identical. Then consider: is this a good use of overloading, or would a single `double` method suffice? What are the tradeoffs?
5. The chapter mentions that the `String` class has 13 constructors. Compile and run the `AllStringConstructors` program from the chapter. Then identify which constructors produce identical output despite having different signatures (e.g. constructors 8, 9, 10, and 11 all print `Java`). Explain why different constructors can produce the same result, and why Java still provides all of them as separate overloads rather than collapsing them into one.

Static Fields & Methods

Introduction

In a previous chapter we investigated instance fields and methods and how they related to class definitions and objects. We learned that every *instance* of an object has its own corresponding fields tied to *that instance* which is distinct from another instance. Java provides the direct opposite of instance fields with **static fields**.

What Does static Mean?

`static` is a reserved keyword which can be prepended to either a variable or a method in a class definition. Per the Java Language Specification for static fields:

> *If a field is declared static, there exists exactly one incarnation of the field, no matter how many instances (possibly zero) of the class may eventually be created. A static field, sometimes called a class variable, is incarnated when the class is initialized.*

This means that static fields are tied to an *entire class*. They are not associated with a specific instance of a class (i.e. an object). The same goes for static methods:

> *A method that is declared static is called a class method.*
>
> *A class method is always invoked without reference to a particular object.*

Both static methods and static fields belong to the *class itself* and not a given instance of that class.

We can modify the `BankAccount` class from the instance chapter to include a static field which tracks the unique ID of each bank account created:

Listing 1: Modified BankAccount class

```
 1:   public class BankAccount {
 2:       private static int UID = 0;
 3:       private String owner;
 4:       private double balance;
 5:       private int uid;
 6:
 7:       BankAccount(String owner, double initialBalance){
 8:           this.owner   = owner;
 9:           this.balance = initialBalance;
10:           this.uid = BankAccount.UID;
11:           BankAccount.UID += 1;
12:       }
13:
14:       void deposit(double amount){
15:           balance = balance + amount;
16:       }
17:
18:       void withdraw(double amount){
19:           balance = balance - amount;
20:       }
21:
22:       void printBalance(){
23:           System.out.printf("(%d) %s's balance: $%.2f%n", uid, owner,
balance);
24:       }
25:
26:       static int getNumAccountsCreated(){
27:           return BankAccount.UID;
28:       }
29:   }
```

We can then use the above class definition like so:

Listing 2: Using the modified BankAccount class

```
 1:   public class UsingStaticAccount{
 2:       public static void main(String[] args){
 3:           String[] names = {"Alan", "Barbara", "Clive", "Denise", "Eddie",
"Fran"};
 4:           BankAccount[] accs = new BankAccount[names.length];
 5:           for(String n : names){
 6:               int i = BankAccount.getNumAccountsCreated();
 7:               accs[i] = new BankAccount(n, 1000);
 8:           }
 9:           for(BankAccount acc : accs){
10:               acc.printBalance();
11:           }
12:           System.out.printf("We created %d accounts!%n",
BankAccount.getNumAccountsCreated());
13:       }
14:   }
```

After compiling and running the code we get:

Listing 3: Using the modified BankAccount class results

```
josephraskind@stargazer:/tmp/static$ javac UsingStaticAccount.java; java
UsingStaticAccount
(0) Alan's balance: $1000.00
(1) Barbara's balance: $1000.00
(2) Clive's balance: $1000.00
(3) Denise's balance: $1000.00
(4) Eddie's balance: $1000.00
(5) Fran's balance: $1000.00
We created 6 accounts!
```

That's a lot to process!

Let's go through this step by step. First, we modified `BankAccount` to include a new static field `UID`, a new instance field `uid`, and a new static method `getNumAccountsCreated`. Next, the constructor was modified to set the `uid` of the new instance wiht the value of `UID`. Because `UID` is private it can only be referenced in the context of the BankAccount definition. Furthermore, because `UID` is static, the value is "shared" between all instances (at the class level). This is why each constructor invocation is able to use the value of `UID` to set the value of their internal `uid`s. We can also see that the `printBalance` method was modified to include a reference to the `uid`. Each printout of `uid` in `UsingStaticAccount` is different because `uid` is an *instance field*. Finally, `getNumAccountsCreated` returns the value of the `UID` static field—it is allowed to do so because it is defined as static in its method declaration.

`UID` was referenced using the class name paired with the `.` notation. `BankAccount.UID` retrieved the static `UID` value. Again, static members are not tied to an instance, but rather the class, therefore I need the class name when referencing a static member.

Since static methods are not tied to a particular instance they **cannot** reference instance members. For example, if I created a new, static function called `printStaticBalance`:

```
1:   static void printStaticBalance{
2:       System.out.printf("(%d) %s's balance: $%.2f%n", uid, owner, balance);
3:   }
```

it would not compile! `uid`, `owner`, and `balance` only make semantic sense when referring to a particular instance of `BankAccount`. However, it is possible for instance methods to reference static members.

Note: Many beginner Java programmers do not recognize this when they start programming. Oftentimes they will make the mistake of calling instance methods from a static context and get frustrated with the compiler when it points out that is illegal in the language. Always double check that you are in the proper method context when invocation errors crop up.

Exercises

1. Compile and run Listing 2. Then create a second program that instantiates three `BankAccount` objects and prints the result of `BankAccount.getNumAccountsCreated()` once after all three have been created. What value does it return? Now call `getNumAccountsCreated` before creating any accounts and print that too. What does this confirm about when the static field is initialized and how it accumulates across all instantiations?

2. The chapter states that static members are accessed via the class name rather than an instance variable. Attempt to access `getNumAccountsCreated` through an instance variable instead:

   ```
   BankAccount acc = new BankAccount("Alice", 1000.00);
   System.out.println(acc.getNumAccountsCreated());
   ```

   Does it compile? Does it run? What warning or note does the compiler give you and why does it discourage this style even if it technically works?

3. It turns out that `public static void main` must be static because the JVM calls it before any objects exist. Try removing the `static` keyword from a `main` method and attempt to run it. What error do you get? In your own words explain why the JVM cannot call a non-static `main` method at program startup.
4. Add a `static final double INTEREST_RATE = 0.05` field to `BankAccount` representing a fixed annual interest rate shared by all accounts. Then add an instance method `applyInterest` that multiplies `balance` by `(1 + INTEREST_RATE)`. Test it on two accounts and verify that changing `INTEREST_RATE` to `0.10` updates the behavior for all accounts without modifying any instance. Why is `final` an appropriate qualifier for this field?
5. The chapter states that instance methods can reference static members but static methods cannot reference instance members. Write a small class that demonstrates both directions: one instance method that reads a static field, and one static method that attempts to read an instance field. Compile both and record what happens. Explain in your own words why the restriction only applies in one direction.
6. The chapter shows `UID` is declared `private static`. Change `private` to `public` and recompile. Now attempt to reset the counter from `UsingStaticAccount` with `BankAccount.UID = 0` after creating a few accounts. What happens to subsequent `uid` values? What does this demonstrate about why `UID` should remain `private`?
7. Write a class `Counter` with a single `private static int count = 0` field, a constructor that increments `count`, and a static method `getCount` that returns it. Instantiate `Counter` objects in a loop and verify `getCount` reflects the total number of instances created. Then write a second class in the same program that also instantiates `Counter`

objects. Does `getCount` reflect instantiations from both classes? Explain why.

8. The chapter shows that `BankAccount.UID` persists across all instances created during a program run. Consider what happens if you wanted to reset the UID counter — for example when writing tests. Add a `static void resetUID` method to `BankAccount` that resets `UID` to `0`. Write a program that creates three accounts, resets the counter, then creates three more. Print all six accounts' `uid` values. What do you observe and what does this suggest about the risks of mutable static state?

Exceptions

Introduction

General purpose programming languages provide an immense amount of freedom for the programmer. Assuming hardware can keep up, absolutely anything that can be imagined can be created. From programs that simulate the building blocks of the universe, to programs that render Sonic the Hedgehog into three dimensional space, a general purpose programming language is only limited by the creativity and engineering capabilities of the programmer. This freedom is a double-edged sword, however, as it also provides the programmer with more than enough rope to hang herself with. Some languages, like Java, provide built-in mechanisms for dealing with situations where these sorts of issues arise. In Java, these mechanisms are called **exceptions** and they allow for the programmer to plan against potential errors that might crop up during program execution.

What is an Exception?

A definition taken directly from the language specification (§ 11) should do nicely:

> *When a program violates the semantic constraints of the Java programming language, the Java Virtual Machine signals this error to the program as an exception.*

At this point, we know that a "semantic constraint" refers to anything in the language that must be satisfied to keep up with the *meaning* and *use* of the language. For example, dividing a number by zero would "violate the semantic constrains of the Java programming language" because Java explicitly does not allow for such a thing to happen. Let's check it out:

Listing 1: Dividing by zero

```
1:  public class DivideByZero{
2:      public static void main(String[] args){
3:          System.out.printf("5/0 = %d%n", 5/0);
4:      }
5:  }
```

After compiling and running we get:

Listing 2: Dividing by zero results

```
josephraskind@stargazer:/tmp/exceptions$ javac DivideByZero.java; java
DivideByZero
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivideByZero.main(DivideByZero.java:3)
```

The program compiles just fine, but when we try to run it we get an *exception* at the exact moment the illegal operation is performed. Again, this happens to prevent such illegal semantic occurrences from being processed by the runtime engine. Java does not define what happens when an integer is divided by zero and as such *nothing should happen*.

Note: Not all languages provide built-in exceptions. For example, C categorizes nearly all semantic violations as "undefined behavior" where the language designers functionally ignore edge cases.

Types of Exceptions

Java defines two sorts of exceptions: **checked exceptions** and **unchecked exceptions**.

Unchecked Exceptions

The `ArithmeticException` exception we witnessed earlier falls under the unchecked exception category. Unchecked exceptions are all exceptions that are either subclasses of `RuntimeException` or `Error`. `RuntimeException` is a category of exception whereby recovery is thought to still be possible whereas for `Error` it is unlikely the program can recover. What separates unchecked exceptions from checked exceptions si that they do not need to be wrapped in a try-catch clause—neither do they need the `throws` modifier appended to a method declaration.

Checked Exceptions

Checked exceptions are all exceptions which are **not** unchecked exceptions. By contrast, checked exceptions must follow a strict set of syntax in order to function properly:

- Checked exceptions must be **caught** using a try-catch clause; or
- Checked exceptions must be included in a method declaration annotation

For example, the standard library class `FileReader`'s constructor will throw a `FileNotFoundException` if an incorrect filepath was sent into the method (you can refer to the method API here):

Listing 3: Using FileReader's constructor

```
1:  import java.io.FileReader;
2:  public class UsingFileReader{
3:      public static void main(String[] args){
4:          FileReader fr = new FileReader("./fake_file.txt");
5:      }
6:  }
```

When I compile the above code I get the following:

Listing 4: Using FileReader's constructor results

```
josephraskind@stargazer:/tmp/exceptions$ java UsingFileReader.java
UsingFileReader.java:5: error: unreported exception FileNotFoundException;
must be caught or declared to be thrown
        FileReader fr = new FileReader("./fake_file.txt");
                        ^
1 error
error: compilation failed
```

Because FileReader **throws** a `FileNotFoundException` I must either tag the main with a `throws` annotation, or I must catch the exception in a try-catch block:

Listing 5: Using FileReader's constructor with throws

```
1:  import java.io.FileReader;
2:  import java.io.FileNotFoundException;
3:  public class UsingFileReader{
4:      public static void main(String[] args) throws FileNotFoundException{
5:          FileReader fr = new FileReader("./fake_file.txt");
6:      }
7:  }
```

Now I can compile and run it:

Listing 6: Using FileReader's constructor with throws results

```
josephraskind@stargazer:/tmp/exceptions$ java UsingFileReader.java
Exception in thread "main" java.io.FileNotFoundException: ./fake_file.txt (No
such file or directory)
        at java.base/java.io.FileInputStream.open0(Native Method)
        at java.base/java.io.FileInputStream.open(FileInputStream.java:213)
        at java.base/java.io.FileInputStream.<init>(FileInputStream.java:152)
        at java.base/java.io.FileInputStream.<init>(FileInputStream.java:106)
        at java.base/java.io.FileReader.<init>(FileReader.java:60)
        at UsingFileReader.main(UsingFileReader.java:5)
```

Perfect!

Now let's see how it looks with the try-catch statement:

Listing 7: Using FileReader's constructor with try-catch

```
 1:  import java.io.FileReader;
 2:  import java.io.FileNotFoundException;
 3:  public class UsingFileReader{
 4:      public static void main(String[] args){
 5:          try{
 6:              FileReader fr = new FileReader("./fake_file.txt");
 7:          }catch(FileNotFoundException e){
 8:              e.printStackTrace();
 9:          }
10:      }
11:  }
```

After compiling and running we get:

Listing 8: Using FileReader's constructor with try-catch results

```
josephraskind@stargazer:/tmp/exceptions$ java UsingFileReader.java
java.io.FileNotFoundException: ./fake_file.txt (No such file or directory)
        at java.base/java.io.FileInputStream.open0(Native Method)
        at java.base/java.io.FileInputStream.open(FileInputStream.java:213)
        at java.base/java.io.FileInputStream.<init>(FileInputStream.java:152)
        at java.base/java.io.FileInputStream.<init>(FileInputStream.java:106)
        at java.base/java.io.FileReader.<init>(FileReader.java:60)
        at UsingFileReader.main(UsingFileReader.java:6)
        at java.base/
jdk.internal.reflect.DirectMethodHandleAccessor.invoke(DirectMethodHandleAccess
or.java:103)
        at java.base/java.lang.reflect.Method.invoke(Method.java:580)
        at jdk.compiler/
com.sun.tools.javac.launcher.SourceLauncher.execute(SourceLauncher.java:264)
        at jdk.compiler/
com.sun.tools.javac.launcher.SourceLauncher.run(SourceLauncher.java:153)
        at jdk.compiler/
com.sun.tools.javac.launcher.SourceLauncher.main(SourceLauncher.java:78)
```

Handling Exceptions

Now we have some idea of what exceptions are, how do they work? Simply put, when an exception is thrown control will be transferred to the "nearest" catch block in the method call stack. If none is found then excecution will be halted. The following diagram visualizes the process:

The try-catch statement will `catch` on exceptions of a given class. For example, in the above diagram the `catch` is set to `ArithmeticException` so it can only match with an `ArithmeticException` or a subclass of `ArithmeticException`. We can define try-catch statements with more than one possible `catch`:

Listing 9: Try-catch with multiple catch statements

```
 1:   public class MultipleCatches{
 2:       public static void main(String[] args){
 3:           int user = Integer.parseInt(args[0]);
 4:           try{
 5:               int res = 1 / user;       // Will throw exception if user is
0
 6:               int[] arr = new int[1];
 7:               int val = arr[user];      // Will throw exception if user > 0
 8:           }catch(ArithmeticException e){
 9:               System.out.println("Caught by ArithmeticException catch!");
10:           }catch(RuntimeException e){
11:               System.out.println("Caught by RuntimeException catch!");
12:           }
13:           System.out.println("Exiting...");
14:       }
15:   }
```

If we compile and run the code with two different inputs:

Listing 10: Try-catch with multiple catch statements results

```
josephraskind@stargazer:/tmp/exceptions$ java MultipleCatches.java 0
Caught by ArithmeticException catch!
Exiting...
josephraskind@stargazer:/tmp/exceptions$ java MultipleCatches.java 1
Caught by RuntimeException catch!
Exiting...
```

First, let's take a look at what happened in the initial run. We sent in `"0"` as the input which was converted to the integer `0` and used to divide 1. This cannot happen which causes Java to throw an `ArithmeticException`. Because the statement was wrapped in a try-catch block control will tried to be transferred to the catch block. The first matching block is on Line 8 as it matches the exception exactly. This causes the `"Caught by ArithmeticException catch!"` to be printed out to the screen.

Second, on the latter run we sent in `"1"` as the input which was converted to the integer `1` and used to divide 1. This does not cause any errors which allows for the array, `arr`, to be instantiated. We then try to retrieve the value at index 1 which does not exist and therefore throws an `IndexOutOfBoundsException`. There is no explicit catch for `IndexOutOfBoundsException` present, but there is for `RuntimeException` of which `IndexOutOfBoundsException` is a subclass. Therefore, the exception is caught by the `catch` block on Line 10 which causes the `"Caught by RuntimeException catch!"` to be printed out to the screen.

Additionally, since both exceptions were caught by the try-catch block the program exited normally as evidenced by the two `"Exiting..."` printouts. This is the major advantage of exception handling. Other languages may have the entire runtime system crash when such an event occurs, but Java allows for these mistakes to be dealt with to prevent such a serious occurrance.

Defining an Exception

Java provides us with the ability to define our own exceptions. Sometimes the exceptions provided by the Java standard library do not reflect the sorts of issues that might crop up in our own program logic. Let's take the `BankAccount` class as an example. We know that it is unreasonable to consider a call to withdraw which has a negative amount—you cannot take negative money out of the bank. This provides us with a perfect opportunity to define an exception that we can raise any time a withdrawl or transation occurs! We can do so as follows:

Listing 11: Creating a BankAccount exception

```
1:   public class WithdrawlException extends RuntimeException{
2:       public WithdrawlException(String msg){
3:         super(msg);
4:     }
5:   }
```

I defined a new class caled `WithdrawlException` which is a *subclass* of `RuntimeException`. I then provide a constructor which takes a message to be printed out by whatever catches the exception. We can alter the `BankAccount` class to throw a `WithdrawlException` in the event of a negative withdrawl:

Listing 12: Altering BankAccount to throw a WithdrawlException

```
 1:   public class BankAccount {
 2:       private String owner;
 3:       private double balance;
 4:
 5:       BankAccount(String owner, double initialBalance){
 6:           this.owner   = owner;
 7:           this.balance = initialBalance;
 8:       }
 9:
10:       void deposit(double amount){
11:           balance = balance + amount;
12:       }
13:
14:       void withdraw(double amount){
15:           if(amount < 0){
16:               throw new WithdrawlException("Cannot have a negative
amount!");
17:           }
18:           balance = balance - amount;
19:       }
20:
21:       void printBalance(){
22:           System.out.printf("%s's balance: $%.2f%n", owner, balance);
23:       }
24:   }
```

When the illegal operation is found (a negative amount is sent to `withdraw`) the `WithdrawlException` is *thrown*. Throw is a reserved keyword that always takes an exception object—because one is not in memory

previously I must instantiate it with a call to `new`. Finally, we can test it out with a main class:

Listing 13: Withdrawing negative funds

```
1:  public class NegativeWithdrawl{
2:      public static void main(String[] args){
3:          BankAccount acc = new BankAccount("Alice", 1000);
4:          acc.withdraw(-1000);
5:          System.out.println("Exiting...");
6:      }
7:  }
```

After compiling and running we get:

Listing 14: Withdrawing negative funds results

```
josephraskind@stargazer:/tmp/exceptions$ java NegativeWithdrawl.java
Exception in thread "main" WithdrawlException: Cannot have a negative amount!
        at BankAccount.withdraw(BankAccount.java:16)
        at NegativeWithdrawl.main(NegativeWithdrawl.java:4)
```

Voilá! After sending in a negative amount to `withdraw` in the `main` function we set off a chain of events which lead to the termination of the application.

Note: The "Exiting..." string was not printed out to the screen. This is because the `WithdrawlException` was not caught and therefore it was propagated "outside" of the main leading to the program halting before the `println`.

Exercises

1. Compile and run Listing 1. Then wrap the division in a try-catch block that catches `ArithmeticException` and prints a friendly error message instead of crashing. Verify that execution continues normally after the catch block by adding a `System.out.println("Program continues")` after the try-catch.
2. The chapter shows two ways to handle a checked exception: using `throws` as in Listing 5 and using try-catch as in Listing 7. Implement both approaches yourself using `FileReader`. In your own words explain the difference in behavior between the two — what happens to the exception in each case?
3. Compile and run Listing 9 with the inputs `0` and `1` to verify the results match the chapter. Then run it with the input `-1`. What happens? Trace through the code manually and explain which exception is thrown and which catch block handles it.
4. The chapter states that catch blocks are matched in order from top to bottom. Write a program with a try block that throws an `ArithmeticException` and two catch blocks: one for `RuntimeException`

and one for `ArithmeticException`, in that order. Does it compile? What does the compiler tell you and why?

5. Add a `WithdrawlException` catch block to Listing 13 so that the program handles the exception gracefully and prints `"Exiting..."` as originally intended. Verify that the program no longer crashes and that the message is printed.

6. The chapter defines `WithdrawlException` as a subclass of `RuntimeException` making it an unchecked exception. Redefine it as a subclass of `Exception` instead to make it a checked exception:

```
public class WithdrawlException extends Exception {
    public WithdrawlException(String msg){
        super(msg);
    }
}
```

Recompile `BankAccount` and `NegativeWithdrawl`. What new compiler errors appear? What changes must you make to get the code to compile again? What does this tell you about the practical difference between checked and unchecked exceptions?

7. Add a second custom exception `OverdraftException` to `BankAccount` that is thrown when a withdrawal would cause the balance to go below zero. Test it by attempting to withdraw more than the available balance. Then write a `main` that catches both `WithdrawlException` and `OverdraftException` separately and prints a different message for each.
8. The propagation diagram in the chapter shows an exception passing through `method_b` before being caught in `method_a`. Implement this exact scenario in Java with three methods calling each other. Verify the output matches the diagram. Then add a try-catch inside `method_b` as well — does the exception still reach `method_a`?
9. The chapter notes that `"Exiting..."` was not printed in Listing 14 because the uncaught exception halted the program. Java provides a `finally` block that executes regardless of whether an exception was thrown or caught. Research `finally` and add one to Listing 13 that always prints `"Exiting..."`. Verify it prints even when the exception is not caught.
10. Write a class `SafeCalculator` with a static method `divide(int a, int b)` that returns the result of `a / b` but throws a custom `DivisionByZeroException` (which you must define) when `b` is zero. Write a `main` that calls `divide` with several inputs including zero, catches the exception, and continues running after each call. Make `DivisionByZeroException` an unchecked exception and explain why that is the appropriate choice here.

Inheritance

Introduction

When introducing objects, I discussed the idea that they are powerful tools of abstraction which allow for us to reprsent real-world things in the form of user-defined types that are subsequently managed by the Java runtime system. I had emphasized that there is a difference between a t-shirt-as-a-concept and a particular t-shirt-as-an-object. I had also mentioned that t-shirts can be abstracted away as *clothing*—a t-shirt **is** an article of clothing (it is made of some sort of fabric and it is worn to cover the body). Object-oriented programming allows us to construct types that are in *relation* to one another, that form a **hierarchy**. In Java, we call this hierarchy "**inheritance**".

What is Inheritance?

In Object-oriented programming inheritance refers to the concept of constructing objects which *take on* qualities of their predecessors. Recall what you know of basic biology. There are a variety of classification strata which denote where certain living organism reside in relation to one another. Let's take the humble dog or *canis lupus familiaris* as an example. A dog sits under a variety of hierarchical classifications:

```
Living Thing
└── Animal
    ├── Arthropod
    ├── Mollusk
    └── Chordate
        ├── Fish
        ├── Reptile
        ├── Bird
        └── Mammal
            ├── Rodent
            ├── Primate
            └── Carnivore
                ├── Cat
                ├── Bear
                └── Dog ← (Canis lupus familiaris)
```

A dog is a living thing, an animal, a mammal, and a carnivore. Within each grouping provides more specific information about the bodily makeup of the organism and the sorts of behaviors that it might express. A mammal breathes air, a carnivore eats meat, a chordate has a spinal chord (or a spinal chord-like structure), etc.

Java allows for us to construct similar groupings through the use of class hierarchies. For example I can create an `Animal` class:

Listing 1: Animal class

```
1:  class Animal{
2:      String name;
3:      Animal(String n){
4:          name = new String(n);
5:      }
6:      void eat(){
7:          System.out.printf("%s eats!%n", name);
8:      }
9:  }
```

I can then create a *subclass* of `Animal` called `Mammal`:

Listing 2: Mammal class

```
1:  class Mammal extends Animal{
2:      Mammal(String n){
3:          super(n);
4:      }
5:      void breathe(){
6:          System.out.printf("%s breathes!%n", name);
7:      }
8:  }
```

The `extends` keyword is what turns `Mammal` into a subclass of `Animal`. We say that `Animal` is the "superclass" of `Mammal` and `Mammal` is a subclass of `Animal`. It so happens that Java uses "single-inheritance" where a class may only `extend` a single class[1]. Furthermore, in the constructor on Line 4 of Listing 2 we see a new keyword, `super`. `super` explicitly references members of the super class—when in the context of the constructor it can be used as a method call to the "super constructor" (the constructor of the superclass).

How Inheritance Works

Although it may seem strange to conceptualize at first, the basics of inheritance are actually reasonably intuitive. A subclass will *inherit* all members (fields and methods) of the superclass. For example, `Mammal` gets `name` and `eat()` from `Animal`:

Listing 3: Using Mammal

```
1:  public class Farm{
2:      public static void main(String[] args){
3:          Mammal m = new Mammal("Lassie");
4:          m.eat();
5:          m.breathe();
6:      }
7:  }
```

After compiling and running we get:

Listing 4: Using Mammal results

```
Lassie breathes!josephraskind@stargazer:/tmp/inheritance$ java Farm.java
Lassie eats!
Lassie breathes!
```

We can see that even though `m` is only declared as a `Mammal` type we get the original `eat()` from `Animal` alongside the `breathe()` from `Mammal`! Again, this is because `Mammal` is a *subclass* of `Animal` and therefore inherits all of what `Animal` has available.

It is important to understand this **does not** work in the opposite direction. For example:

Listing 5: Using Animal like Mammal

```
1:  public class Farm{
2:      public static void main(String[] args){
3:          Animal a = new Animal("Lassie");
4:          a.eat();
5:          a.breathe();
6:      }
7:  }
```

After compiling the above we get:

Listing 6: Using Animal like Mammal results

```
josephraskind@stargazer:/tmp/inheritance$ javac Farm.java
Farm.java:5: error: cannot find symbol
        a.breathe();
         ^
  symbol:   method breathe()
  location: variable a of type Animal
1 error
error: compilation failed
```

We can see that `Farm` was incapable of compiling because the symbol `breathe` could not be found. This is because `Animal` does not have access to `breathe` as it is defined within a subclass---`Animal` has no understanding of its subclasses!

Subsubclasses

We can create arbitrarily many subclasses by adding more classes to the hierarchy chain:

Listing 7: Dog class

```
1:  class Dog extends Mammal{
2:      Dog(String n){
3:          super(n);
4:      }
5:      void bark(){
6:          System.out.printf("%s barks!%n", name);
7:      }
8:  }
```

`Dog`'s superclass is `Mammal` whose superclass is `Animal` therefore `Dog` can reference all three methods:

Listing 8: Using Dog

```
1:  public class Farm{
2:      public static void main(String[] args){
3:          Dog d = new Dog("Lassie");
4:          d.eat();
5:          d.breathe();
6:          d.bark();
7:      }
8:  }
```

After compiling and running we get:

Listing 9: Using Dog results

```
josephraskind@stargazer:/tmp/inheritance$ java Farm.java
Lassie eats!
Lassie breathes!
Lassie barks!
```

Variable Shadowing

Occassionally it may make sense to redifine fields within subclasses that share the names of fields in superclasses. The way that Java handles these identical symbol sets is through the use of variable **shadowing**. This means that the execution context informs the symbol that is being referenced. For example, if I alter `Animal` and `Mammal` to both have a `weight` field like so:

Listing 10: Adding shadowed variable to Animal and Mammal

```
1:  class Animal{
2:      String name;
3:      double weight;
4:      Animal(String n, double w){
5:          name = new String(n);
6:          weight = w;
7:      }
8:      void animalWeight(){
9:          System.out.printf("%s weighs %.2fkg!%n", name, weight);
10:     }
11: }
12: class Mammal extends Animal{
13:     double weight;
14:     Mammal(String n, double w){
15:         super(n, w);
16:     }
17:     void mammalWeight(){
18:         System.out.printf("%s weighs %.2fkg!%n", name, weight);
19:     }
20: }
```

There are now two separate `weight` fields: one for `Animal` and one for `Mammal`. We can test this out with the following:

Listing 11: Using both weights

```
1:  public class Farm{
2:      public static void main(String[] args){
3:          Mammal m = new Mammal("Lassie", 21.5);
4:          m.weight = 18.5;
5:          m.animalWeight();
6:          m.mammalWeight();
7:      }
8:  }
```

Compiling and running the above we get:

Listing 12: Using both weights results

```
josephraskind@stargazer:/tmp/inheritance$ java Farm.java
Lassie weighs 21.50kg!
Lassie weighs 18.50kg!
```

How can this be?! Lassie can't weight 21.5kg and 18.5kg at the same time! The answer has to do with *shadowed variables*. There are *actually* two `weight` fields contained within the `Mammal` object at the same time: one tied to the `Animal` context and the other tied to the `Mammal` context. The Java compiler does nothing to advise us on this fact because it takes for granted that we are aware of the possibility of variable shadowing. You should always be careful when shadowing variables!

Access Modifiers

We learned previously that we can tag *access modifiers* onto members of a class (as well as the classes themselves). Doing so restricts the visibility of those members in regards to other classes. Naturally, we can also use access modifiers within inherited classes as well. For example, I can set the `name` field to `private` in `Animal`:

Listing 13: Animal class

```
1:  class Animal{
2:      private String name;
3:      Animal(String n){
4:          name = new String(n);
5:      }
6:      void eat(){
7:          System.out.printf("%s eats!%n", name);
8:      }
9:  }
```

If we assume the default implementation of `Mammal` in Listing 2 and try to compile the code in Listing 3 we get:

Listing 14: Compiling with a private name

```
josephraskind@stargazer:/tmp/inheritance$ java Farm.java
/tmp/inheritance/Mammal.java:6: error: name has private access in Animal
        System.out.printf("%s breathes!%n", name);
                                            ^
1 error
error: compilation failed
```

Even though `name` is inherited by `Mammal` it cannot access it because it has been set to `private` within `Animal`.

The is-a Relationship

In OOP, inheritance establishes relationships between different classes. We can categorize these relationships into two camps the "is-a" and the "has-a". Let's take a look at the `Dog` hierarchy:

```
                  Animal
                +--------+
                | name   |  ← has-a String
                +--------+
                | eat()  |
                +--------+
                     △
                     |  is-a (extends)
                     |
                  Mammal
                +----------+
                | breathe()|
                +----------+
                     △
                     |  is-a (extends)
                     |
                    Dog
                +--------+
                | bark() |
                +--------+

is-a relationships:
  Dog    is-a Mammal
  Dog    is-a Animal
  Mammal is-a Animal

has-a relationships:
  Animal has-a String (name)
  Mammal has-a String (name)  ← inherited
  Dog    has-a String (name)  ← inherited
```

We can verify "is-a" relationships through the use of the `instanceof` operator:

Listing 15: Using instanceof to check is-a relationship

```
 1:   public class Farm{
 2:
 3:       public static void checkInstance(Animal a){
 4:           System.out.printf("%s instanceOf Animal: %b%n",
 5:                             a.name,
 6:                             a instanceof Animal);
 7:           System.out.printf("%s instanceOf Mammal: %b%n",
 8:                             a.name,
 9:                             a instanceof Mammal);
10:           System.out.printf("%s instanceOf Dog: %b%n",
11:                             a.name,
12:                             a instanceof Dog);
13:       }
14:
15:       public static void main(String[] args){
16:           Animal a = new Animal("Alfred");
17:           Mammal m = new Mammal("Margery");
18:           Dog d    = new Dog("Douglas");
19:
20:           checkInstance(a);
21:           checkInstance(m);
22:           checkInstance(d);
23:       }
24:   }
```

After compiling and running we get:

Listing 16: Using instanceof to check is-a relationship results

```
josephraskind@stargazer:/tmp/inheritance$ java Farm.java
Alfred instanceOf Animal: true
Alfred instanceOf Mammal: false
Alfred instanceOf Dog: false
Margery instanceOf Animal: true
Margery instanceOf Mammal: true
Margery instanceOf Dog: false
Douglas instanceOf Animal: true
Douglas instanceOf Mammal: true
Douglas instanceOf Dog: true
```

Note: Line 3 in Listing 15 might be a bit confusing, but the next chapter on polymorphism will explain how we can send in a Dog argument to an Animal parameter.

Why Inheritance?

We have seen that the advantage of inheritance comes in the form of passing down all of the members associated with a superclass, the superclass's superclass, and so on. But why is this advantageous? The most obvious benefit comes in the form of code reuse. When we create `Mammal` we do not need to rewrite `eat()` because we get it for free as `Mammal` is a subclass of `Animal`. As a corollary, if we want to change `eat()` we only have to change it in `Animal` and the change will be represented in `Mammal`, `Dog`, and any other class which extends `Animal`. We also get to leverage a powerful type system which can do things like runtime and static typechecking to make sure that certain behaviors are capable of being performed during the execution of our programs. When we write a program that uses a `Dog` object we know that a `Dog` should have a `name` and should be able to invoke the functions `eat()`, `breathe()`, and `bark()`.

Why Not Inheritance?

OOP is not without its faults and inheritance is something that is hotly contested as being a benefit to a language. Although code reuse can save time it can also cause massive problems. For example, if a superclass is changed in such a way that a new bug is introduced, all subclasses will now inherit that bug through no fault of their own! Many also find the "is-a" relationship too constricting as it locks a class down into a rigid hierarchy—classical solutions involve the championing of "has-a" relationships over "is-a" relationships. Encapsulation is also often violated through inheritance as subclasses will frequently demand total knowledge of their superclass ancestors.

Preventing Subclasses

Because of the pitfalls associated with inheritance, Java includes the option for programmers to prevent subclassing of certain classes. This can be achieved by tagging a class definition with `final`:

Listing 17: Final Dog class

```
1:  final class Dog extends Mammal{
2:      Dog(String n){
3:          super(n);
4:      }
5:      void bark(){
6:          System.out.printf("%s barks!%n", name);
7:      }
8:  }
```

Now if I try to extend `Dog`:

Listing 18: Extending final `Dog`

```
1:  class Collie extends Dog{
2:      Collie(String n){
3:          super(n);
4:      }
5:      void breed(){
6:          System.out.printf("%s is a Collie!%n", name);
7:      }
8:  }
```

and I compile the above I get the following error:

Listing 19: Extending /final `Dog` results

```
josephraskind@stargazer:/tmp/inheritance$ javac Collie.java
Collie.java:1: error: cannot inherit from final Dog
class Collie extends Dog{
                     ^
1 error
```

The `final` keyword makes it so `Dog` can **never** have any subclasses.

Exercises

1. Compile and run Listing 8 Then add a fourth class `Poodle` that extends `Dog` and adds a method `groom`. Instantiate a `Poodle` and verify it can call `eat`, `breathe`, `bark`, and `groom`. The chapter marks `Dog` as `final` — what change must you make first and why does `final` prevent this?
2. The chapter shows that `Dog` inherits `eat` from `Animal` without redefining it. Create an `Animal[]` containing one instance each of `Animal`, `Mammal`, and `Dog` and call `eat` on each through the array. What prints for each? What does this demonstrate about how inherited methods behave when called through a parent type reference?

3. The chapter introduces `super` to call parent constructors. Remove the `super(n)` call from `Mammal`'s constructor and attempt to compile. What error do you get? Now add a no-argument constructor to `Animal` that sets `name` to `"Unknown"` and recompile without `super`. What value does `name` have? What does this tell you about when `super` is required?
4. Change the `name` field in `Animal` from default (package-private) access to `public` and recompile. Does anything break? Now change it to `protected` and recompile. What is the practical difference between `public` and `protected` for an inherited field? Which would you choose and why?
5. The chapter states that `instanceof` checks the entire inheritance chain. Write a program that creates one instance each of `Animal`, `Mammal`, and `Dog` and tests every combination of `instanceof` (e.g. `dog instanceof Animal`, `animal instanceof Dog`, etc.). Print each result, predict it before running, and explain any results that surprise you.
6. Rewrite the `Animal`, `Mammal`, and `Dog` hierarchy replacing inheritance with composition: `Mammal` should contain an `Animal` field rather than extending it, and `Dog` should contain a `Mammal` field. Implement `eat`, `breathe`, and `bark` by delegating to the contained objects. Compare the two approaches — what do you gain and what do you lose by using composition instead of inheritance here?
7. The chapter marks `Dog` as `final`. Look up Java's `Math` class and `String` class — are they also `final`? Why might the designers of Java have made those classes `final`? Give one reason why marking a class `final` is a deliberate design decision rather than just a restriction.
8. Add a `Cat` class that also extends `Mammal` and has a `meow` method. Then create an `Animal[]` containing a `Dog` and a `Cat` and use an enhanced for loop with `instanceof` to call the appropriate sound method (`bark` for `Dog`, `meow` for `Cat`) on each element. What does this exercise demonstrate about the relationship between inheritance and polymorphism?
9. The chapter shows that `Mammal` calls `super(n)` to pass the name up to `Animal`. Add a second field `int age` to `Animal` and update its constructor to accept both `name` and `age`. Trace the changes that must be made down through `Mammal` and `Dog` to keep everything compiling. What does this exercise reveal about the fragile base class problem?
10. The chapter shows that Java uses the `extends` keyword to create a hierarchy of classes. Using only what you have learned so far, design a three-level class hierarchy of your own choosing that is different from the animal example. Write out the classes with appropriate fields and methods at each level, ensuring each subclass adds something new. Instantiate one object of each class and verify that the bottom-level class has access to all methods defined at every level above it.

Footnotes:

1 Other OOP languages like C++ and Python allow for multiple inheritance. Java eschewed its inclusion due to complications that can arise in highly complex class dependency hierarchies—often referred to as the *diamond problem*.

Polymorphism

Introduction

In biology, polymorphism is defined as the phenomenom where different phenotypes can appear within the population of a species—i.e., the difference between jaguars and panthers (panthers being melanated jaguars). It is a combination of the greak πολύ (polý), or "many", and μορφή (morfí), or "form". Computer scientists eventually repurposed the phrase to fit the phenomena in complex type systems whereby a single variable can take on many different type forms. Polymorphism is deeply embedded in the Java type system and is something that every Java programmer must be intimately familiar with.

The Object Class

It turns out that every single object in Java is a subclass of the `Object` class. If we take a look at the standard library documentation for Object we'll find that there are a handful of methods which all objects get for free: `clone()`, `equals(...)`, `hashCode()`, etc. This means that every single non-`Object` object in Java is inherently a polymorphic type. It contains within it the type `Object` and whatever else derived type was defined via the class definition. For example, let's go back to our `Animal` classes from the previous chapter:

Listing 1: Animal classes

```
class Animal{
    String name;
    Animal(String n){
        name = new String(n);
    }
    void eat(){
        System.out.printf("%s eats!%n", name);
    }
}
class Mammal extends Animal{
    Mammal(String n){
        super(n);
    }
    void breathe(){
        System.out.printf("%s breathes!%n", name);
    }
}
final class Dog extends Mammal{
    Dog(String n){
        super(n);
    }
    void bark(){
        System.out.printf("%s barks!%n", name);
    }
}
```

We can also throw in a few more:

Listing 2: Additional Animal classes

```
final class Cow extends Mammal{
    Cow(String n){
        super(n);
    }
    void moo(){
        System.out.printf("%s moos!%n", name);
    }
}
final class Pig extends Mammal{
    Pig(String n){
        super(n);
    }
    void oink(){
        System.out.printf("%s oinks!%n", name);
    }
}
```

Because Java allows for subtyping polymorphism, I can store three distinct `Animal`-derived instances in a single `Object`-based container:

Listing 3: Storing Animal-derived objects in an Object array

```
public class Farm{
    public static void main(String[] args){
        Object[] arr = {new Dog("Lassie"), new Cow("Bessie"), new
Pig("Porky")};
    }
}
```

If you send this code through the `javac` compiler you'll find no errors are thrown. This is because `Dog`, `Cow`, and `Pig` are ultimately all derived from `Object` and can therefore be stored as `Object`s. We'll investigate this a bit more in the section on polymorphic subtyping.

Method Overriding

One of the most significant advantages of polymorphism in object-oriented programming language is the ability to **override** ancestral methods. Let's alter the example shown in Listing 3 to try to have each animal make their trademark sound:

Listing 4: Making the animals sing

```
 1:   public class Farm{
 2:       public static void main(String[] args){
 3:           Animal[] arr = {new Dog("Lassie"), new Cow("Bessie"), new
Pig("Porky")};
 4:           for(Animal a : arr){
 5:               if(a instanceof Dog){
 6:                   ((Dog) a).bark();
 7:               }else if(a instanceof Cow){
 8:                   ((Cow) a).moo();
 9:               }else if(a instanceof Pig){
10:                   ((Pig) a).oink();
11:               }
12:           }
13:       }
14:   }
```

Yuck! But if we compile and run it we will find that it does the job just fine:

Listing 5: Making the animals sing results

```
josephraskind@stargazer:/tmp/polymorphism$ java Farm.java
Lassie barks!
Bessie moos!
Porky oinks!
```

We can see that Listing 4 works well enough, but it makes for some rather tiresome coding. Furthermore, anytime we define a new `Animal` we'll need to adjust `Farm` to check for another possible derived instance. A way to skirt that is to utilize *method overriding*. We can reconfigure our class definitions to allow for the technique:

Listing 6: Updating the Animal classes to include makeSound

```
class Animal{
    String name;
    Animal(String n){
        name = new String(n);
    }
    void eat(){
        System.out.printf("%s eats!%n", name);
    }
    void makeSound(){
        System.out.printf("%s makes a noise!%n", name);
    }
}
class Mammal extends Animal{
    Mammal(String n){
        super(n);
    }
    void breathe(){
        System.out.printf("%s breathes!%n", name);
    }
}
final class Dog extends Mammal{
    Dog(String n){
        super(n);
    }
    void makeSound(){
        System.out.printf("%s barks!%n", name);
    }
}
final class Cow extends Mammal{
    Cow(String n){
        super(n);
    }
    void makeSound(){
        System.out.printf("%s moos!%n", name);
    }
}
final class Pig extends Mammal{
    Pig(String n){
        super(n);
    }
    void makeSound(){
        System.out.printf("%s oinks!%n", name);
    }
}
```

Method overriding occurs when the exact same method (return type, name, and arguments) is defined in a subclass. The Java runtime will invoke whicher is the most recent definition of the method within the inheritance hierarchy. We can see how this plays out with an updated `Farm` class:

Listing 7: Making the animals sing with overriding

```
1:   public class Farm{
2:       public static void main(String[] args){
3:           Animal[] arr = {new Dog("Lassie"), new Cow("Bessie"), new
Pig("Porky")};
4:           for(Animal a : arr)
5:               a.makeSound();
6:       }
7:   }
```

After compiling and running we get the exact same results as Listing 5! How can that be the case? Whenever a method is invoked in Java, it will be checked to see if it has been overridden. If it has, then the "closest" definition relative to the instance type will be used. So even though the compile-time type of `a` on Line 5 is `Animal` the *actual*, *underlying* runtime-type is `Dog`, `Cow`, or `Pig`. This does not mean that `Animal` can at all times be a stand-in for any derived type or treated as such. For example, if `Dog` contained an instance method called `fetch` then `a` could not reference it as it would not also be shared by `Animal`.

Overriding vs Overloading

Students often get confused between method *overriding* and method *overloading*. Overriding occurs when a subtype reimplements a method defined above it in its class hierarchy—this requires declaring a method with the **exact same** method header. Overloading occurs whenever a method is declared with the same name, but a different set of arguments. It is always best practice to add an `@Override` annotation before functions you are trying to override. If the annotation is used and the function is not being overridden, but rather overloaded, the compiler will throw an error. Otherwise, the compiler will let you make the mistake of overloading instead of overriding a function! Below is a classic example taken from *Effective Java (3rd Edition)*:

Listing 8: Not quite overriding

```
1:  public class Bigram {
2:      private final char first;
3:      private final char second;
4:      public Bigram(char first, char second) {
5:          this.first = first;
6:          this.second = second;
7:      }
8:      public boolean equals(Bigram b) {
9:          return b.first == first && b.second == second;
10:     }
11:     public int hashCode() {
12:         return 31 * first + second;
13:     }
14:     public static void main(String[] args) {
15:         Set<Bigram> s = new HashSet<>();
16:         for (int i = 0; i < 10; i++)
17:             for (char ch = 'a'; ch <= 'z'; ch++)
18:                 s.add(new Bigram(ch, ch));
19:         System.out.println(s.size());
20:     }
21: }
```

Although it looks like `equals` is overriding `Object`'s equals method, it is not. This is because `equals` should take an `Object` type not a `Bigram` type. Java will have no problem letting this mistake be made. The solution? adding an `@Override` tag will prevent the above code from compiling. I leave it as an exercise to the reader to test that out.

Polymorphic Subtyping

As mentioned above, the ability for objects in Java to shift between types in the class hierarchy is called *polymorphic subtyping*. It is often a difficult topic for students to conceptualize as it appears to break the strong type system that has be presented heretofore. This sort of polymorphism **is not magic** neither is it completely lawless. You can follow a simple algorithm to determine whether or not a class can be treated like another class:

- A class may be treated as a superclass
- A superclass may be treated as a subclass

Examples and Oddities

Below are a few exampels of explicit conversions between different object types:

Listing 9: Explicit casting between different object types

```
 1:   public class Farm{
 2:       public static void main(String[] args){
 3:           Dog d = new Dog("Lassie");
 4:           Cow c = new Cow("Bessie");
 5:
 6:           Animal a = (Animal) c;                //   Legal conversion:
subclass to superclass
 7:           a = (Animal) d;                       //   Legal conversion:
subclass to superclass
 8:           d = (Dog) a;                          //   Legal conversion:
superclass to subclass (same instance)
 9:
10:           try{
11:               Dog fake_d = (Dog) ((Animal) c);  // Illegal conversion:
superclass to subclass (different instance)
12:           }catch(ClassCastException e){         // (Runtime Error)
13:               System.out.println(e);
14:           }
15:           try{
16:               Cow fake_c = (Cow) a;             // Illegal conversion:
superclass to subclass (different instance)
17:           }catch(ClassCastException e){         // (Runtime Error)
18:               System.out.println(e);
19:           }
20:           try{
21:               Cow fake_c = (Cow) d;             // Illegal conversion:
subclass to unrelated subclass
22:           }catch(ClassCastException e){         // (Compile-time Error)
23:               System.out.println(e);
24:           }
25:
26:       }
27:   }
```

The compiler will trust that the subtype conversions being performed are valid knowing that the runtime system will eventually catch any mistakes.

Why Polymorphism?

The largest advantage of polymorphism can be witnessed in Listing 7, we can store many different subtypes in a single container and have each perform a unique action using an overridden method. This behavior simplifies interfaces/APIs and makes it easier to reuse code wherever applicable. Polymorphism also grants the ability to replicate real-world "is-a" relationships: a `Student` is-a `Person`, a `Book` "is-a" `Item`, a `Pig` "is-a" `Animal`, etc.

Why Not Polymorphism?

Again, much like inheritance, polymorphism presents as a double-edged sword. It becomes difficult to trust the types being supplied when they can be absolutely any subtype. It also is unlikely that all subtypes *perfectly* reflect the is-a relationship that might be given by the parent (a `Square` "is-a"

`Rectangle`, but you cannot independently set the length and height). Finally, there are major performance overheads associated with managing a runtime typechecking system (which is mandatory in a type-safe polymorphic language)[1].

Exercises

1. The chapter introduces method overriding. Add an `eat` method to `Dog` that overrides the one defined in `Animal` and prints `"[name] wolfs down food!"`. Then add a `Cat` class that extends `Mammal` and also overrides `eat` to print `"[name] nibbles delicately!"`. Write a `main` that creates an `Animal[]` containing one instance each of `Animal`, `Mammal`, `Dog`, and `Cat`, and calls `eat` on each through the array. Which version of `eat` is called for each element? What does this confirm about how Java resolves method calls at runtime?
2. The chapter shows that `@Override` is optional but recommended. Remove `@Override` from `Dog`'s `eat` method and introduce a typo — change it to `public void eatt()`. Does it compile? Does it behave as expected? Now restore `@Override` and introduce the same typo. What does the compiler tell you? What does this demonstrate about the value of `@Override`?
3. The chapter states that polymorphism allows a single method to operate on any subtype. Write a static method `makeEat(Animal a)` that calls `a.eat()`. Call it with instances of `Animal`, `Mammal`, and `Dog`. Which version of `eat` is called in each case? Explain why in terms of dynamic dispatch.
4. The chapter shows that a method defined in a parent class is inherited by all subclasses. Add a `toString` method to `Animal` that returns a `String` describing the animal (e.g. `"Animal: Rex"`). Call `System.out.println` on instances of `Animal`, `Mammal`, and `Dog` — Java automatically calls `toString` when an object is passed to `println`. What prints for each instance? What does this confirm about how inherited methods behave across the hierarchy?
5. The chapter shows that a subclass can call its parent's constructor using `super`. Add a second field `String sound` to `Animal` and update its constructor to accept both `name` and `sound`. Trace the changes needed through `Mammal` and `Dog` to keep everything compiling. Then print the `sound` field from a `Dog` instance to verify it was correctly passed up the chain.
6. The chapter states that a reference of a parent type can hold an instance of any subtype. Write a program that assigns a `Dog` instance to an `Animal` reference and attempt to call `bark` through that reference. What does the compiler say? What does this tell you about the tradeoff between generality and specificity when using parent type references?
7. The chapter shows that overriding allows subclasses to specialize inherited behavior. Add a `speak` method to `Animal` that prints `"..."`, override it in `Mammal` to print `"[name] makes a sound!"`, and override it again in `Dog` to print `"[name] barks!"`. Create one instance of each class, store them all in an `Animal[]`, and call `speak` on each. Verify the most specific version is called in each case.

8. Write a static method `describeAnimal(Animal a)` that prints the name of the animal and calls `eat`. Call it with instances of `Animal`, `Mammal`, and `Dog`. Which version of `eat` is called in each case and why? What does this demonstrate about the value of writing methods that accept parent types rather than specific subtypes?
9. The chapter introduces polymorphism as a way to write more general code. Without polymorphism, a program that needed to feed every animal on a farm would require a separate method for each type: `feedDog`, `feedCat`, `feedBird`, and so on. Write both versions — one with a separate method per type and one with a single `feed(Animal a)` method — for at least three animal types. Count the number of lines of code in each version. What happens to the non-polymorphic version when you add a fourth animal type? What happens to the polymorphic version?
10. The chapter presents polymorphism as resolving the tension between generality and specificity. Revisit the composition-based hierarchy you wrote in the inheritance exercises and consider: could you achieve the same polymorphic behavior (storing different types in a single array and calling a shared method) using composition alone, without inheritance or interfaces? Try it and describe what you find. What does this tell you about when polymorphism through subtyping is genuinely necessary?

Footnotes:

1 C++ skirts some of this overhead by allowing for undefined behavior.

Abstract Classes

Introduction

Classes act as the blueprints for objects. Objects in turn act as the in-memory representations of things in the real world, or at the very least, things we wish to perform computations on. If we pair that with our understanding of inheritance and polymorphism we'll quickly run into situations where behaviors become a bit too difficult to generalize regardless of the fact that they may be present in many sorts of the same category of object. For example, we know all animals make sounds, but those sounds will differ between each animal. We might want to enforce the *contract* of the ability to make sounds without establishing a generic, shared sound. Java provides the ability to declare **abstract classes** for exactly this scenario.

What is an Abstract Class?

According to the Java Language Specfication (§ 8.1.1.1):

> *An abstract class is a class that is incomplete, or to be considered incomplete.*

This does not sound desireable at all! However, it's the semantic behavior of abstract classes that provides its usefulness. Because an abstract class is considered "incomplete" by Java that means they *cannot be instantiated*. We can define an abstract class `Animal` like so:

Listing 1: Defining Animal as an abstract class

```
 1:  abstract class Animal{
 2:      String name;
 3:      Animal(String n){
 4:          name = new String(n);
 5:      }
 6:      void eat(){
 7:          System.out.printf("%s eats!%n", name);
 8:      }
 9:      abstract void makeSound();
10:  }
```

We can see the use of a new keyword, `abstract`, above. `abstract` is used as a class modifier and as a method modifier. All classes with `abstract` methods must themselves be `abstract` classes. Classes which inherit from abstract classes naively, must also necessarily be abstract:

Listing 2: Defining Mammal as a subclass of abstract Animal

```
1:  class Mammal extends Animal{
2:      Mammal(String n){
3:          super(n);
4:      }
5:      void breathe(){
6:          System.out.printf("%s breathes!%n", name);
7:      }
8:  }
```

If I try to compile the above I get:

Listing 3: Compiling the Mammal definition

```
josephraskind@stargazer:/tmp/abstract_classes$ javac Mammal.java
Mammal.java:1: error: Mammal is not abstract and does not override abstract
method makeSound() in Animal
class Mammal extends Animal{
^
1 error
error: compilation failed
```

What does the above error mean? First, all classes are abstract which have *at least one* abstract method. Since `Mammal` is a subclass of `Animal` it has one abstract method `makeSound()`. Second, if a class is abstract it must be modified with the `abstract` keyword.

What's the usefulness of an abstract class if I can never instantiate it and all subclasses are abstract? Let's take a look at the error message in Listing 3: "Mammal is not abstract and **does not override abstract method makeSound() in Animal**". Ah ha! We must *override* an abstract method to dissolve its abstract*ness*. `Mammal` is still a bit abstract to deserve an implemenation of `makeSound()`, so we can tag is as abstract and move on to `Dog`:

Listing 4: Defining Mammal as abstract and Dog as concrete

```
1:  abstract class Mammal extends Animal{
2:      Mammal(String n){
3:          super(n);
4:      }
5:      void breathe(){
6:          System.out.printf("%s breathes!%n", name);
7:      }
8:  }
9:
10: final class Dog extends Mammal{
11:     Dog(String n){
12:         super(n);
13:     }
14:     void eat(){
15:         System.out.printf("%s eats meat!%n", name);
16:     }
17:     void makeSound(){
18:         System.out.printf("%s barks!%n", name);
19:     }
20: }
```

The above compiles nicely! This is because `Dog` finally provides an implementation for `makeSound()` which decrements the number of fully abstract methods to 0. That turns `Dog` from a potentially abstract class to a fully concretized class.

As seen on Lines 6-8 in Listing 1, abstract classes may have concrete implementations of methods when shared behavior is desired.

Abstract Class Members and Instantiation

As seen above in Listing 1, abstract classes *are* allowed to have instance fields and instance methods. However, that does not mean that an abstract class can be instantiated in its own right:

Listing 5: Instantiating abstract Animal

```
1:  public class Farm{
2:      public static void main(String[] args){
3:          Animal a = new Animal("Lassie");
4:      }
5:  }
```

If we try to compile this we get:

Listing 6: Instantiating abstract Animal compilation results

```
josephraskind@stargazer:/tmp/abstract_classes$ javac Farm.java
Farm.java:51: error: Animal is abstract; cannot be instantiated
        Animal a = new Animal("Lassie");
                   ^
1 error
```

This should be somewhat intuitive. `Animal` is still missing an implementation of `makeSound()` so if it were invoked the Java runtime would have no idea what to do—it would be undefined behavior, something which the language specification avoids as much as possible. However, that does not mean we cannot ever have variables of type `Animal`:

Listing 7: Placing Dog instance in Animal variable

```
1:  public class Farm{
2:      public static void main(String[] args){
3:          Animal a = new Dog("Lassie");
4:          a.makeSound();
5:      }
6:  }
```

When we compile and run this we get:

Listing 8: Placing Dog instance in Animal variable result

```
1:  josephraskind@stargazer:/tmp/abstract_classes$ java Farm.java
2:  Lassie barks!
```

Why does this work? Refer back to the chapter on polymorphism.

Interfaces

Java allows for the declaration of an even-more-abstract class in the form of **interfaces**. An interface is a totally abstract class where there is absolutely no concrete implemenation at all. Every method within an interface is by definition abstract. We can alter `Animal` to transform it into an interface:

Listing 9: Animal interface

```
1:  interface Animal{
2:      void eat();
3:      void makeSound();
4:  }
```

Notice what's happened: we have removed the instance field `name` and we have gotten rid of the `eat()` implementation. These are both necessary steps in turning `Animal` into an interface as they may not have any sort of concrete implementation[1]. An interface may not have an instance field as an interface is meant to purely as a set of guide rails for further class definitions. Let us do the name to `Mammal`:

Listing 10: Mammal interface

```
1:  interface Mammal extends Animal{
2:      void breathe();
3:  }
```

We can see that `extends` works just the same when defining an interface. Here `Mammal` is defined as a subinterface of `Animal` which means that it inherits all of the abstract methods associated with `Animal`. I can now redefine `Dog` which will implement `Mammal`:

Listing 11: Dog implementing Mammal

```
 1:  final class Dog implements Mammal{
 2:      String name;
 3:      Dog(String n){
 4:          name = new String(n);
 5:      }
 6:      public void breathe(){
 7:          System.out.printf("%s breathes air!%n", name);
 8:      }
 9:      public void eat(){
10:          System.out.printf("%s eats meat!%n", name);
11:      }
12:      public void makeSound(){
13:          System.out.printf("%s barks!%n", name);
14:      }
15:  }
```

We can then write some code to test out this new `Dog` class:

Listing 12: Testing new Dog implementation

```
1:  public class Farm{
2:      public static void main(String[] args){
3:          Dog d = new Dog("Lassie");
4:          d.breathe();
5:          d.eat();
6:          d.makeSound();
7:      }
8:  }
```

Compiling and running the above gives us:

Listing 13: Testing new Dog implementation results

```
josephraskind@stargazer:/tmp/abstract_classes$ java Farm.java
Lassie breathes air!
Lassie eats meat!
Lassie barks!
```

Because `Dog` `implements` `Mammal` and `Dog` is not tagged with `abstract`, `Dog` must provide a concrete implementation for all abstract methods associated with the `Mammal` interface.

Why Abstract Classes/Interfaces?

As shown in Listing 7 we can still leverage polymorphism with abstract classes. This is also true with interfaces. For example, if I redefined the `Cow` and `Pig` classes from the previous chapter to implement `Animal`:

Listing 14: Defining Cow and Pig as implementations of Animal

```
 1:   final class Cow implements Animal{
 2:       String name;
 3:       Cow(String n){
 4:           name = new String(n);
 5:       }
 6:       public void eat(){
 7:           System.out.printf("%s eats grass!%n", name);
 8:       }
 9:       public void makeSound(){
10:           System.out.printf("%s moos!%n", name);
11:       }
12:   }
13:   final class Pig implements Animal{
14:       String name;
15:       Pig(String n){
16:           name = new String(n);
17:       }
18:       public void eat(){
19:           System.out.printf("%s eats everything!%n", name);
20:       }
21:       public void makeSound(){
22:           System.out.printf("%s oinks!%n", name);
23:       }
24:   }
```

We can use the old polymorphic example below:

Listing 15: Using the classes that implement Animal

```
1:   public class Farm{
2:       public static void main(String[] args){
3:           Animal[] arr = {new Dog("Lassie"), new Cow("Bessie"), new
Pig("Porky")};
4:            for(Animal a : arr)
5:                a.makeSound();
6:       }
7:   }
```

Which gives us the same output as before:

Listing 16: Using the classes that implement Animal results

```
josephraskind@stargazer:/tmp/abstract_classes$ java Farm.java
Lassie barks!
Bessie moos!
Porky oinks!
```

Our use of interfaces solves the problem of upstream classes ruining downstream subclasses. Each implementation is guaranteed to have their own `makeSound()` method, but each class has its own unique implementation. Of course, what we gain in control over implemenation, we lose in code reuse.

Exercises

1. The chapter shows that `Animal` cannot be instantiated directly once made abstract. Attempt to compile and run the following and record the error:

```
Animal a = new Animal("Generic");
```

   Then assign a `Dog` instance to an `Animal` reference and verify it compiles. In your own words explain why instantiating an abstract class is prohibited while assigning a subclass instance to an abstract type reference is permitted.

2. The chapter states that a class containing any abstract method must itself be declared abstract. Attempt to compile the following and record the error:

```
class Shape {
    abstract double area();
}
```

   Add the `abstract` keyword to `Shape` and recompile. Then create a concrete subclass `Circle` with a `double radius` field that implements `area`. What happens if you declare `Circle` abstract as well?

3. The chapter introduces abstract classes as a way to share concrete method implementations while forcing subclasses to define certain behaviors. Add a concrete method `describe` to `Animal` that prints `"I am [name] and I say: "` followed by the result of `makeSound`. Instantiate a `Dog` and call `describe`. How is it possible for `Animal`'s concrete `describe` method to call `makeSound` when `Animal` itself provides no implementation of `makeSound`?
4. The chapter shows that `Mammal` remains abstract because it does not implement `makeSound`. Add a concrete `breathe` method to `Mammal` and verify that `Dog` inherits it. Then attempt to instantiate `Mammal` directly and record the error. What does this confirm about the relationship between having concrete methods and being instantiable?
5. Abstract classes can have constructors even though they cannot be instantiated. Add a constructor to `Animal` that prints `"An animal is being created!"` and verify it is called when a `Dog` is instantiated. Explain in your own words why an abstract class needs a constructor if it can never be directly instantiated.
6. The chapter motivates abstract classes as a tool for enforcing a contract across a hierarchy. Design your own three-level abstract hierarchy on a topic of your choosing. The top level should be fully abstract, the middle level should implement some but not all abstract methods and add at least one new concrete method, and the bottom level should be fully concrete. Instantiate the bottom level class and verify it has access to all methods defined at every level.

Footnotes:

1 This is not entirely true. Interfaces may define `default` method implementations, but I have chosen to ignore this feature.

Generics

Introduction

So far we've dealt with defining classes where absolutely everything about the class was known at the moment we finished writing the code. For example, when we wrote the `BankAccount` class we defined it as having a `name` field of type `String` and a `balance` field of type `double`. This meant that for every single instantiation of `BankAccount` there would always be a `String` and a `double` contained within the object. That structure worked well because a `BankAccount` object only ever represented a programmatic version of a bank account. However, there are certain classes where it would be more advantageous for the definition allow for the use of any number of types. The simplest analogue would be an array—we can declare an array to be of *any type*. If we could only declare arrays that stored `int`s they wouldn't be nearly as useful to us. It turns out that Java allows us to have that sort of type flexibility with our own user-defined classes in the form of **generics**.

Note: This is often a *very* challenging subject for students. If you are struggling with generics that's okay! Keep practicing and it *will* begin to make sense.

Declaring a Generic Class

The Java Language Specification (§ 8.1.2) states that:

> *A class is generic if the class declaration declares one or more type variables.*

Which, of course, isn't very helpful in and of itself. Like with most things programming-wise, an example will prove useful:

Listing 1: Declaring a simple generic class

```
1:  public class MyObject<T>{
2:      public T genericObject;
3:  }
```

This, mostly, looks exactly the same to what we've seen before in class definitions. The biggest difference is we now see `<>` after the class name and a symbol, `T`, in between them which is used in the next line before the variable `genericObject`. The `T` in this instance is a **type variable**. Type variables are essentially stand-ins for the type that will eventually be filled in when the object is instantiated. Looking at how we can create an instance of `MyObject` will show what I mean:

Listing 2: Instantiating MyObject

```
1:   public class UsingGenerics{
2:       public static void main(String[] args){
3:           MyObject<Integer> m_i = new MyObject<Integer>();
4:           m_i.genericObject = 10;
5:           System.out.println(m_i.genericObject);
6:       }
7:   }
```

You can see that in Line 3 the `T` has been replaced by `Integer`. `T` in the class definition was a type variable which was assigned the "value" of the type `Integer` in Listing 2. If we compile and run the above code we get:

Listing 3: Instantiating MyObject results

```
josephraskind@stargazer:/tmp$ java UsingGenerics.java
10
```

The output shouldn't be too surprising. We create an instance of `MyObject` whose type variable is set to `Integer`. This means that `genericObject`, a type-parameterized instance field, will also be set to `Integer` and therefore can be used as if it was an `Integer` in Line 4.

One of the benefits of using generics is that one definition will work for any type. Let's modify `MyObject` to include slightly more functionality:

Listing 4: Adding a toString and constructor to MyObject

```
 1:   public class MyObject<T>{
 2:       public T genericObject;
 3:       MyObject(T obj){
 4:           genericObject = obj;
 5:       }
 6:       @Override
 7:       public String toString(){
 8:           return String.format("This MyObject holds %s",
genericObject.getClass().getSimpleName());
 9:       }
10:   }
```

I've added a `toString` method to `MyObject` alongside the ability to construct it with an object. We can test out this functionality using the code below:

Listing 5: Testing out with multiple MyObjects

```
 1:  public class UsingGenerics{
 2:      public static void main(String[] args){
 3:          MyObject<Integer> m_i = new MyObject<>(10);
 4:          System.out.println(m_i);
 5:
 6:          MyObject<String> m_s = new MyObject<>("Hello World");
 7:          System.out.println(m_s);
 8:
 9:          MyObject<MyObject<Integer>> m_mo = new MyObject<>(m_i);
10:          System.out.println(m_mo);
11:      }
12:  }
```

You'll notice some new syntax on the instantiation lines. The `<>` on either side do not precisely match. This is because Java allows the user to eschew a set of type parameters on the right side of an assignment statement as the compiler can infer that they should both match. Compiling and running this code we get:

Listing 6: Testing out with multiple MyObjects results

```
josephraskind@stargazer:/tmp$ java UsingGenerics.java
This MyObject holds Integer
This MyObject holds String
This MyObject holds MyObject
```

We can see that we were able to instantiate *three* different `MyObject` objects that hold three different types!

Generic Methods

Java also allows the declaration of generic methods. Generic methods are of a more limited utility, but they work just the same:

Listing 7: Declaring and using a generic method

```
class MyInt{
    int i;
    MyInt(int i){
        this.i = i;
    }
    @Override
    public String toString(){
        return String.format("MyInt[%d]", i);
    }
}

public class GenericMethods{

    public static <T> void foo(T var){
        System.out.println(var);
    }

    public static void main(String[] args){
        GenericMethods.<String>foo("Hello World");
        GenericMethods.<Double>foo(2.5);
        GenericMethods.<MyInt>foo(new MyInt(17));
    }
}
```

After compiling and running we get:

Listing 8: Declaring and using a generic method results

```
josephraskind@stargazer:/tmp$ java GenericMethods.java
Hello World
2.5
MyInt[17]
```

Although it appears that three different versions of `foo` are being called, it turns out that only one version is.

Using Wildcards and Boundaries

It turns out that generics with total freedom with regard to the sort of type parameters sent in can sometimes hamper the usefulness of a particular class. For example, let's imagine an alternate version of `MyObject` called `MyNumber`. If we used the exact same definition as `MyObject` in Listing 1 it would be far too easy to instantiate a non-number:

Listing 9: Naive MyNumber class

```
public class MyNumber<T>{
    public T genericNumber;
    public static void main(String[] args){
        MyNumber<String> m_n = new MyNumber<>();
        m_n.genericNumber = "Hello World";
    }
}
```

The above code will compile just fine which is a problem if I only what `MyNumber` to contain `Number` objects. I can employ the use **boundaries** to provide restrictions to the possible type parameters that can be sent in to the type declaration:

Listing 10: Naive MyNumber class with bounds

```
1:   public class MyNumber<T extends Number>{
2:       public T genericNumber;
3:       public static void main(String[] args){
4:           MyNumber<String> m_n = new MyNumber<>();
5:           m_n.genericNumber = "Hello World";
6:       }
7:   }
```

If I try to compile the above I get:

Listing 11: Naive MyNumber class with bounds results

```
josephraskind@stargazer:/tmp$ javac MyNumber.java
MyNumber.java:4: error: type argument String is not within bounds of type-
variable T
        MyNumber<String> m_n = new MyNumber<>();
                 ^
  where T is a type-variable:
    T extends Number declared in class MyNumber
MyNumber.java:4: error: incompatible types: cannot infer type arguments for
MyNumber<>
        MyNumber<String> m_n = new MyNumber<>();
                                           ^
    reason: inference variable T has incompatible bounds
      equality constraints: String
      upper bounds: Number
  where T is a type-variable:
    T extends Number declared in class MyNumber
2 errors
```

I can no longer use *just any type*. I must use a type that can be matched to the boundaries provided. In Listing 10, `T` has to be a type that *extends*, or is a subclass of, the `Number` type!

Non-class type parameters can also leverage lower bounds in the form of the `super` keyword, as well as utilize **wildcards**. In Java, the wildcard symbol for type parameters is `?`. It acts as a standin to be used in place of a single type. Here is a set of generic methods which leverage the wildcard symbol and boundary keywords for type parameters:

Listing 12: Wildcard examples

```java
import java.util.ArrayList;
import java.util.List;

public class WildcardExamples {

    // 1. Unbounded wildcard -- accepts a list of any type
    //    can only read as Object, cannot write
    static void printAll(List<?> list){
        for(Object o : list)
            System.out.println(o);
    }

    // 2. Upper bounded wildcard (? extends) -- accepts List<Number>
    //    or any subtype: List<Integer>, List<Double> etc.
    //    can READ as Number, cannot write (except null)
    static double sum(List<? extends Number> list){
        double total = 0;
        for(Number n : list)
            total += n.doubleValue();
        return total;
    }

    // 3. Lower bounded wildcard (? super) -- accepts List<Integer>
    //    or any supertype: List<Number>, List<Object>
    //    can WRITE Integers into it, can only READ as Object
    static void addIntegers(List<? super Integer> list){
        list.add(1);
        list.add(2);
        list.add(3);
    }

    public static void main(String[] args){

        // unbounded wildcard
        List<String> strings = List.of("a", "b", "c");
        printAll(strings);   // fine
        List<Integer> ints = List.of(1, 2, 3);
        printAll(ints);      // also fine

        // upper bounded wildcard
        System.out.println(sum(List.of(1, 2, 3)));       // 6.0
        System.out.println(sum(List.of(1.5, 2.5, 3.0))); // 7.0

        // lower bounded wildcard
        List<Number> numbers = new ArrayList<>();
        addIntegers(numbers);
        System.out.println(numbers);  // [1, 2, 3]

        List<Object> objects = new ArrayList<>();
        addIntegers(objects);         // also fine
        System.out.println(objects);  // [1, 2, 3]

    }
}
```

Type Erasure and Raw Types

Java **does not** allow the use of primitives in place for type parameters in generics. The Java Language Specification does not justify this choice, but it is clear that it comes as a result of what's called **type erasure**. Essentially, the type information of a generic is erased at compile-time. When we write code like `ArrayList<Integer> l = new ArrayList<>()` the `<Integer>` is dropped during the runtime of the application leaving the JVM with only the `ArrayList` type. Here's an example that shows how strange type erasure can be:

Listing 13: Type erasure example

```
 1:   import java.util.*;
 2:   public class TypeErasure{
 3:       public static void main(String[] args){
 4:           final List<Integer> li = new ArrayList<>(); //<> tells the
compiler to infer the type as <Integer>
 5:           final List<Float> lf = new ArrayList<>();
 6:           if (li.getClass() == lf.getClass()) { // evaluates to true
 7:               System.out.println("Equal");
 8:           }
 9:       }
10:   }
```

after compiling and running the above:

Listing 14: Type erasure example results

```
josephraskind@stargazer:/tmp$ java TypeErasure.java
Equal
```

Despite the fact that at compile-time it seems like the two `ArrayList` objects have different types, they actually have the same type during the runtime!

Why Generics?

If Java did not provide generics the only other option would be to use the `Object` type exclusively when defining container classes. In fact, this was exactly how Java handled such classes before generics were introduced in Java 5 (2004). The major use-case comes in the form of compilation-type checking. Take the following example sourced from Wikipedia:

Listing 15: Using an ArrayList<Object>

```
1:   import java.util.ArrayList;
2:   import java.util.List;
3:
4:   List v = new ArrayList();
5:   v.add("test");                // A String that cannot be cast to an
Integer
6:   Integer i = (Integer)v.get(0); // Run time error
```

The problem with the code above is that the `String` content of `"test"` is lost the moment it is added to the `List`. Although it is clear to us that it cannot be converted to an `Integer` at compile time, the Java compiler cannot make the same logical leap. For all it "knows", what is stored in `v` is an `Object` and an `Object` is allowed to be cast down to an `Integer` as `Integer` is a subclass of `Object`. If we introduce generics:

Listing 16: Using an ArrayList<String>

```
1:   import java.util.ArrayList;
2:   import java.util.List;
3:
4:   List<String> v = new ArrayList<String>();
5:   v.add("test");                // A String that cannot be cast to an
Integer
6:   Integer i = (Integer)v.get(0); // (type error) compilation-time error
```

The above code will be stopped at compile-time. Why? It is because the `List` now has the extra type information from the generic to know that anything stored within it must be a `String`. This means that the compiler can be confident that `v.get(0)` will return a `String` and check that `String` cannot be converted to an `Integer`. Ostensibly, the extra type information is what makes Java an attractive OOP language to be writing in the first place.

Exercises

1. The chapter shows that without generics, a `MyObject` class would need to use `Object` as its field type. Write a non-generic `MyObject` class with an `Object value` field and `getValue~/ ~setValue` methods. Then write a program that stores a `String` in it and retrieves it, casting back to `String`. Now store an `Integer` and accidentally cast it to `String`. What happens and when does the error occur? Compare this to the generic `MyObject<T>` — at what point does the equivalent error occur?
2. Compile and run Listing 1. Then attempt to instantiate `MyObject<int>` instead of `MyObject<Integer>`. Record the compiler error. Explain in your own words why primitives cannot be used as type parameters and what you must use instead.
3. The chapter introduces bounded type parameters with `extends`. Write a generic method `max` that takes two arguments of type `T extends Comparable<T>` and returns the larger of the two. Test it with `Integer`, `Double`, and `String` arguments. Why is the `Comparable<T>` bound necessary — what would go wrong without it?

4. The chapter shows that `T` is erased to `Object` at runtime. Write a generic class `Pair<A, B>` that holds two values of potentially different types and provides `getFirst` and `getSecond` methods. Then call `getClass().getSimpleName()` on both fields after retrieving them from a `Pair<String, Integer>`. What do you observe about the runtime types?
5. The chapter introduces the unbounded wildcard `<?>`. Write a method `printMyObject` that accepts a `MyObject<?>` and prints its contents. Verify it works with a `MyObject<String>`, a `MyObject<Integer>`, and a `MyObject<Dog>`. Then attempt to call `setValue` inside `printMyObject` with a `String` argument. What does the compiler say and why?
6. The chapter explains the upper bounded wildcard `<? extends T>`. Write a method `sumMyObjects` that takes a `List<? extends Number>` and returns the sum of all values as a `double`. Test it with a `List<MyObject<Integer>>`... wait — does `List<MyObject<Integer>>` satisfy `List<? extends Number>`? Why or why not? What would the correct bound need to be?
7. The chapter explains the lower bounded wildcard `<? super T>`. Write a method `fillWithZero` that takes a `List<? super Integer>` and adds ten zeros to it. Call it with a `List<Integer>`, a `List<Number>`, and a `List<Object>`. Then attempt to call it with a `List<Double>`. What happens and why?
8. The chapter states that a class type parameter can use `extends` but not `super`. Write a generic class `Cage<T extends Animal>` that holds an `Animal` subtype and has a method `release` that calls `eat` on the contained animal. Instantiate it with `Dog` and `Mammal`. Then attempt to instantiate it with `String` and record the compiler error.
9. The chapter introduces generic methods with their own type parameters independent of the class. Write a generic method `swap` that takes an array of type `T[]` and two indices and swaps the elements at those indices. Test it with a `String[]` and an `Integer[]`. Could you write the same method without generics using `Object[]`? What would you lose?
10. The chapter shows that `MyObject<Dog>` is not a subtype of `MyObject<Animal>` even though `Dog` is a subtype of `Animal`. Write a program that demonstrates this by attempting to assign a `MyObject<Dog>` to a `MyObject<Animal>` reference. Record the compiler error. Then rewrite the assignment using a wildcard `MyObject<? extends Animal>` and verify it compiles. Explain in your own words why this invariance exists and what problem it prevents.

JVM

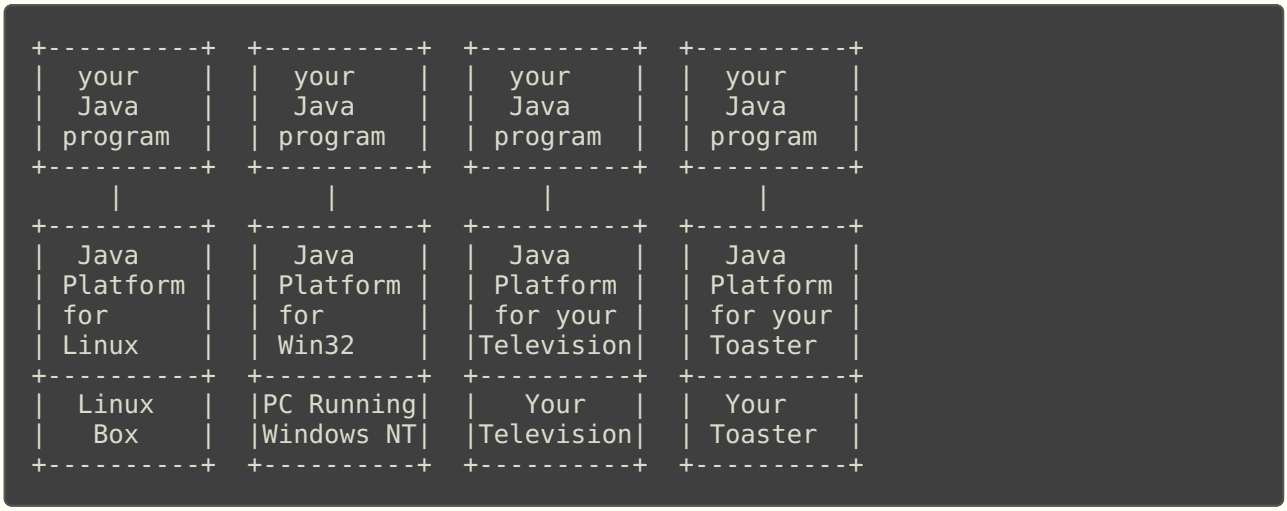
Introduction

In the chapter on compilation we learned that Java does not create fully formed native binary executables, but instead generates **.class** files which are used when executing a program. It was mentioned that these class files consisted of "bytecode" which would be fed into Java's runtime engine. We've been using this runtime engine constantly throughout this course, but we've not investigated exactly what *it is*. Now, we'll be taking a deep-ish dive into Java's runtime engine, the Java Virtual Machine (JVM).

Note: This chapter relies heavily on information provided by the *Java Virtual Machine Specification* which I have hosted on this website for convenience.

Java's Architecture

Java began development as an object-oriented programming language that would act as a competitor to languages like C++. It initially focused on building software for networked consumer devices. This meant that code written in Java would ideally work on any platform that supported it, without the code itself needing to be targeted to fit platform-specific differences. Java's desire to be inherently cross-platform made it fundamentally distinct from C++, where code had to be compiled into an executable for a particular architecture before it is run. The idea can be visuallized as follows:



As long as the Java program was written correctly in the first place, it should be capable of being run, without changes, on every single valid Java platform.

The Java ecosystem can be broken down into a few component parts:

- Java programming language
- Java class file format
- Java API
- Java Virtual Machine

Putting it all together we get:

Listing 1: Java's Architecture

The Java Virtual Machine

We are particularly interested in the last link on the chain, the Java Virtual Machine, or JVM for short. Every single time we've invoked `java` to

run one of our programs we've been spinning up a JVM! At base, the JVM is an **abstract computing machine**. It is abstract because it is not literally a machine, but a computational model that represents one. The JVM, much like a real computer, "has an instruction set and manipulates various memory areas at run time" (§ 1.2, JVM spec). The major advantage of a machine-independent computing engine is that the code that is written for it can completely ignore machine-specific details if desired.

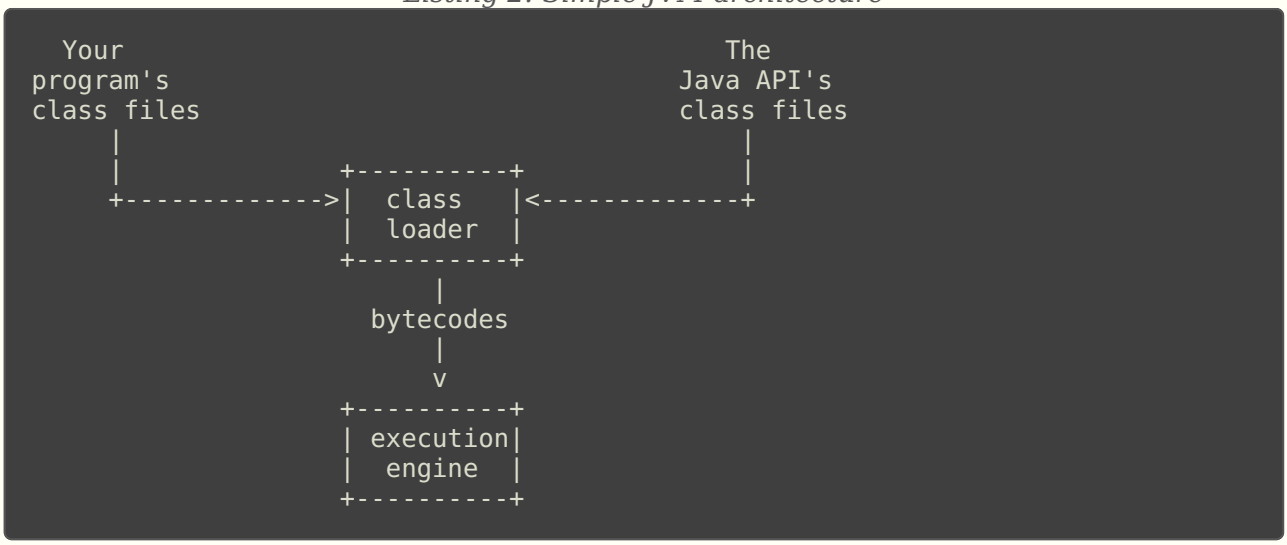
Note: Reflect on all of the code you have written so far for this course, have you ever written any code that specifically targetted x86_64 systems?

One thing about the JVM that might seem incredibly strange is it **knows nothing about the Java programming language**! The JVM has been developed to be a totally independent virtual machine. In fact, there have been many non-Java JVM-based languages created that make use of this reality—the most famous being kotlin, clojure, and scala.

JVM Architecture

Taking a bird's eye view of the JVM, we can represent its architecture visually like so:

Listing 2: Simple JVM architecture



The JVM can be broken down into two major parts: something that loads class files (the class loader) and something that executes the instructions contained in those class files (the execution engine).

JVM Implementation

The *Java Virtual Machine Specification* describes in meticulous detail the internals of class files and the way in which they must be processed by an aspiring JVM. What the specification does not describe is *how exactly* that must be gone about, neither does it describe the various extra features that

can be filled by a JVM developer. This means that much of the JVM specification *implementation agnostic*. As long as class files can be read and the class libraries of the Java SE Platform can be executed, then the program in question is a valid JVM.

HotSpot JDK

There are many JVMs that have been in use historically and that are in use today. In this class, all of our calls to `java` have actually been calls to the *HotSpot JVM*. You can verify this with a call to your `java` executable:

Listing 3: My java installation

```
josephraskind@stargazer:/tmp/jvm$ java --version
openjdk 22.0.1 2024-04-16
OpenJDK Runtime Environment (build 22.0.1+8-16)
OpenJDK 64-Bit Server VM (build 22.0.1+8-16, mixed mode, sharing)
```

We can see above that right now I have `openjdk 22.0.1` installed on my machine which is using the *OpenJDK Runtime Environment*. The internal VM implementation of the *OpenJDK Runtime Environment* is called HotSpot.

Different JVM interpretations will focus on different features/ development environments. HotSpot is used due to its position as the primary reference implementation. GraalVM has its own unique JIT compiler which rivals HotSpot's. Eclipse OpenJ9, originally developed for IBM, has a low memory footprint and fast startup and sees adoption in cloud environments. Azul Zulu provides a OpenJDK-verified JVM implementation which does not require the same commercial licensing agreements as HotSpot.

JVM as Java Interpreter

It is often said that the JVM is a "java interpreter". This is not true in the general case as not all JVMs are strictly interpreters. JVMs read JVM bytecode, but how they go about reading that bytecode is up to the developers. Many JVMs actually compile JVM bytecode down to native code as needed in a process called Just-in-Time compilation (more on this later). There are other JVMs which actually directly run JVM bytecode on hardware, although this is rare. As was mentioned before, the JVM is also not Java-specific—all valid JVMs should be able to process code generated from clojure, scala, kotlin, etc.

The class File

The major "atomic" unit in the JVM is the class file. About ~280 pages of the JVM spec is dedicated to detailing the innerworkings of the class file structure and how it should be processed within a JVM. Each class file contains the definition of a single class, interface, or module. Class files are distinct from executables that are generated from compiling C or C++

insofar as they are meant to be loaded and executed by a JVM whereas C/C++ binaries are meant to run directly on the target machine.

Bytecode

We can take a look at the internal of a class file by using the `javap` program which comes with an installation of the OpenJDK. First we write a simple Java program:

Listing 4: Simple Java program

```
1:   public class Add{
2:       public int add(int x, int y){
3:           int sum = x + y;
4:           return sum;
5:       }
6:   }
```

We then compile it with `javac`:

Listing 5: Simple Java program compiled

```
josephraskind@stargazer:/tmp/jvm$ javac Add.java
josephraskind@stargazer:/tmp/jvm$ ls
Add.class  Add.java
```

This will create a class file whose bytecode we can investigate with `javap -c`:

Listing 6: Add bytecode

```
1:  Compiled from "Add.java"
2:  public class Add {
3:    public Add();
4:      Code:
5:         0: aload_0
6:         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
7:         4: return
8:
9:    public int add(int, int);
10:     Code:
11:        0: iload_1
12:        1: iload_2
13:        2: iadd
14:        3: istore_3
15:        4: iload_3
16:        5: ireturn
17: }
```

What we find is something that looks vaguely like assembly code, but is missing a lot of th e more verbose syntax associated with x86 or arm. We see that `Add` is provided a default constructor (Lines 3-7) and we see that the `add` method is represented as a set of integer loads, an integer addition, an integer store, and return. But how does this tell the JVM how to load this class? In reality, there's more information in the class file than this dump is showing. If we add the `-v` flag to the command we get:

Listing 7: Add bytecode with verbose flag

```
1:  Classfile /tmp/jvm/Add.class
2:    Last modified Jul 31, 2026; size 242 bytes
3:    MD5 checksum 11450389ce554c0e8392091021b12d00
4:    Compiled from "Add.java"
5:  public class Add
6:    minor version: 0
7:    major version: 66
8:    flags: (0x0021) ACC_PUBLIC, ACC_SUPER
9:    this_class: #7                          // Add
10:   super_class: #2                         // java/lang/Object
11:   interfaces: 0, fields: 0, methods: 2, attributes: 1
12: Constant pool:
13:    #1 = Methodref          #2.#3          // java/lang/Object."<init>":()V
14:    #2 = Class              #4             // java/lang/Object
15:    #3 = NameAndType        #5:#6          // "<init>":()V
16:    #4 = Utf8               java/lang/Object
17:    #5 = Utf8               <init>
18:    #6 = Utf8               ()V
19:    #7 = Class              #8             // Add
20:    #8 = Utf8               Add
21:    #9 = Utf8               Code
22:   #10 = Utf8               LineNumberTable
23:   #11 = Utf8               add
24:   #12 = Utf8               (II)I
25:   #13 = Utf8               SourceFile
26:   #14 = Utf8               Add.java
27: {
28:   public Add();
29:     descriptor: ()V
30:     flags: (0x0001) ACC_PUBLIC
31:     Code:
32:       stack=1, locals=1, args_size=1
33:          0: aload_0
34:          1: invokespecial #1                  // Method java/lang/Object."<init>":()V
35:          4: return
36:       LineNumberTable:
37:         line 1: 0
38:
39:   public int add(int, int);
40:     descriptor: (II)I
41:     flags: (0x0001) ACC_PUBLIC
42:     Code:
43:       stack=2, locals=4, args_size=3
44:          0: iload_1
45:          1: iload_2
46:          2: iadd
47:          3: istore_3
48:          4: iload_3
49:          5: ireturn
50:       LineNumberTable:
51:         line 3: 0
52:         line 4: 4
53: }
54: SourceFile: "Add.java"
```

We now see there's a lot more going on! We can see that the class file gives the JVM information on the accessibility of the class (`ACC_PUBLIC` on Line 8), it collects all of the constants that can be referenced (Lines 12-26), it provides type information on all of the methods (for example (II)I for a method that takes two ints and returns an int on Line 40), and much more.

The Class Loader

How exactly do classes enter the JVM's runtime environment? It's done through the **class loader**. The class loader takes the requisite class files, interprets them, and turns them into an in-memory representation that functions as the classes you have defined.

When the JVM is launched one of the first things it does is it begins to load all of the Java standard library files. If you downloaded a version of the jdk onto your machine you can see the actual files that the JVM loads in `path-to-jdk/jmods/java.base.jmod`:

Listing 8: ls of my jdk jmod files

```
josephraskind@stargazer:/tmp/jvm$ ls ~/jdk22/jdk-22.0.1/jmods/
java.base.jmod            jdk.attach.jmod
jdk.jdwp.agent.jmod
java.compiler.jmod        jdk.charsets.jmod                  jdk.jfr.jmod
java.datatransfer.jmod    jdk.compiler.jmod                  jdk.jlink.jmod
                                  ...
jdk.accessibility.jmod    jdk.jdi.jmod                       jdk.zipfs.jmod
```

What we see above are the collection of java packages that come pre-archived in a jdk installation. The first file `java.base.jmod` contains all of the essential class files that the JVM needs to get started. We can list them out with `jmod list`:

Listing 9: Listing all java.base class files

```
josephraskind@stargazer:/tmp/jvm$ jmod list ~/jdk22/jdk-22.0.1/jmods/
java.base.jmod
classes/module-info.class
classes/META-INF/services/java.nio.file.spi.FileSystemProvider
classes/com/sun/crypto/provider/AEADBufferedStream.class
  ...
lib/tzdb.dat
```

There are about 7.6k files so it wouldn't do any good to list them here. If we unzip the file we can find the `Object.class` file buried within:

```
josephraskind@stargazer:/tmp/jvm$ jmod extract ~/jdk22/jdk-22.0.1/jmods/
java.base.jmod --dir ./class_files
```

Now we can look at the `Object.class` file:

```
josephraskind@stargazer:/tmp/jvm$ javap ./class_files/classes/java/lang/
Object.class
Compiled from "Object.java"
public class java.lang.Object {
  public java.lang.Object();
  public final native java.lang.Class<?> getClass();
  public native int hashCode();
  public boolean equals(java.lang.Object);
  protected native java.lang.Object clone() throws
java.lang.CloneNotSupportedException;
  public java.lang.String toString();
  public final native void notify();
  public final native void notifyAll();
  public final void wait() throws java.lang.InterruptedException;
  public final void wait(long) throws java.lang.InterruptedException;
  public final void wait(long, int) throws java.lang.InterruptedException;
  protected void finalize() throws java.lang.Throwable;
}
```

A full bytecode dump would be hundreds of lines, so I leave that as an exercise for the reader.

We can test that standard library methods are loaded before anything else using the following code:

Listing 10: Using Java's Reflection API to get information about the class loader

```
 1:   public class ClassLoaderDemo {
 2:       public static void main(String[] args){
 3:
 4:           // Our own class -- loaded by the application class loader
 5:           System.out.printf("ClassLoaderDemo loaded from: %s%n",
ClassLoaderDemo.class.getClassLoader());
 6:
 7:           // A core Java class -- loaded by the bootstrap loader (null)
 8:           System.out.printf("String loaded from: %s%n",
String.class.getClassLoader());
 9:       }
10:   }
```

If we compile and run the above:

Listing 11: Seeing who loaded what

```
loaded from: nulljosephraskind@stargazer:/tmp/jvm$ java ClassLoaderDemo.java
ClassLoaderDemo loaded from:
com.sun.tools.javac.launcher.MemoryClassLoader@1d7acb34
String loaded from: null
```

We see that our `ClassLoaderDemo` was loaded in from an in-memory process that exists during the JVM's runtime whereas `String` seems to have existed primordially. This is because the boostrap class loader works before anything else to set up the Java Standard Environment.

Java vs C++

Java belongs to the branch of languages called "managed runtime languages". This is because Java was built on the idea that it would be run in tandem with the JVM. Other languages that fit the bill would be Python, Lisp, Perl, Bash, etc. These languages are in direct contrast to "unmanaged runtime languages" like C, C++, Rust, etc. Managed runtime languages utilize *powerful abstractions* to separate the programmer as much as possible from low-level implementation details. For example, in this course we have never once seriously thought about how memory is freed up after we allocate it. This is a major benefit of Java and other managed runtime languages—the technique is called "garbage collection". We also get other goodies like runtime type-checking, array bounds checking, I/O exceptions, and much more. Not to mention the fact that Java has an incredibly robust "reflection" API that lets programmers manipulate and inspect types on the fly.

What we gain in functionality, we lose in speed. Unlike C++, and other unmanaged runtimes, there is massive overhead associated with running the JVM. Although it appears to be non-existant for our simple programs, it can really start to crop up on enterprise-level applications. JVMs have different

ways of combatting this problem, but I will mention only one here: Just-in-Time compilation.

Just-in-Time Compilation

Just-in-Time, or JIT, compilation refers to a technique whereby a JVM identifies portions of code which are running "hot" and converting them directly to native binary. "Hotness" varies by JVM settings, but it essentially boils down to how many times a method, or block of code, is executed within a certain span of time. Take for example a `MatrixMult` class which contains a static method:

Listing 12: MatrixMult class

```
 1:   public class MatrixMult{
 2:       static int[][] matrixMult(int[][]mat1, int[][]mat2){
 3:           final int M = mat1.length;
 4:           final int N = mat2.length;
 5:           final int K = mat2[0].length;
 6:           int[][] matRes = new int[M][K];
 7:           for(int i = 0; i < M; i++){
 8:               for(int j = 0; j < K; j++){
 9:                   matRes[i][j] = 0;
10:                   for(int k = 0; k < N; k++){
11:                       matRes[i][j] += mat1[i][k] * mat2[k][j];
12:                   }
13:               }
14:           }
15:           return matRes;
16:       }
17:   }
```

We can then write a `Driver` class which sets up using the multiplication method for X number of times:

Listing 13: Using MatrixMult X number of times

```
 1:   public class Driver{
 2:       public static void main(String[] args){
 3:           final int X = Integer.parseInt(args[0]);
 4:           int[][] mat = {{1,2,3,4,5},{1,2,3,4,5},{1,2,3,4,5},{1,2,3,4,5},
{1,2,3,4,5}};
 5:           long startTime = System.currentTimeMillis();
 6:           for(int i = 0; i < X; i++)
 7:               MatrixMult.matrixMult(mat, mat);
 8:           System.out.printf("Time Elapsed: %fs%n",
((System.currentTimeMillis()-startTime)/1000.0));
 9:       }
10:   }
```

After compiling and running the code normally we get:

Listing 14: Running normally (with JIT)

```
josephraskind@stargazer:/tmp/jvm$ java Driver.java 10000000
Time Elapsed: 4.625000s
```

Not too shabby for running 10 million matrix multiplication method calls. A lot of this has to do with the fact that JITting is turned on by default. If I run it with JIT explicitly turned off:

Listing 15: No JIT

```
josephraskind@stargazer:/tmp/jvm$ java -Xint Driver.java 10000000
Time Elapsed: 42.209000s
```

we see that the above takes 42s, a nearly 10X speed difference! We can confirm that the JIT compiler made the difference using the `-XX:+PrintCompilation` flag with `java`:

Listing 16: Collecting JIT compiled methods

```
josephraskind@stargazer:/tmp/jvm$ java -XX:+PrintCompilation Driver.java
10000000 > compilation.txt
josephraskind@stargazer:/tmp/jvm$ grep -e "MatrixMult" ./compilation.txt
737 1537       3       MatrixMult::matrixMult (105 bytes)
741 1538 %     4       MatrixMult::matrixMult @ 52 (105 bytes)
750 1539       4       MatrixMult::matrixMult (105 bytes)
756 1537       3       MatrixMult::matrixMult (105 bytes)   made not entrant
```

We can see that `MatrixMult` was natively compiled during the execution which heavily impacted the running time.

Exercises

1. The chapter describes three distinct stages a Java program passes through before producing output: writing source code, compiling to bytecode, and executing on the JVM. For each of the following errors, identify which stage it would be caught at and explain why:
    - A missing semicolon at the end of a statement
    - A `NullPointerException` when calling a method on an uninitialized variable
    - Passing a `String` where an `int` is expected
    - Dividing by zero
    - Referencing a variable that was never declared
2. The chapter draws a distinction between the compile-time environment and the runtime environment. In your own words explain why these are two separate environments rather than one. What is the practical benefit of separating compilation from execution? Think about what this means for distributing a Java program to someone who uses a different operating system than you.
3. The chapter states that the JVM is responsible for executing bytecode regardless of the underlying hardware or operating system. In your own words explain what the JVM must know about the machine it is running on, and what it deliberately hides from your program. Why is it significant that the same `.class` file produced on a Linux machine can run on a Windows machine without recompilation? What would break this guarantee?

4. The chapter states that the bootstrap class loader returns `null` when queried in Java. Run the `ClassLoaderDemo` program from the chapter and observe the output. Why does Java represent the bootstrap loader as `null` rather than as a proper object? What does this tell you about the relationship between the bootstrap loader and the Java object system?
5. The chapter contrasts interpretation and JIT compilation as two strategies for executing bytecode. In your own words describe the tradeoff between the two. Under what circumstances would a pure interpreter be preferable to a JIT compiler? Under what circumstances would the reverse be true? Where does HotSpot's mixed-mode approach sit on this spectrum and why?
6. The chapter explains that Java's "write once, run anywhere" guarantee is made possible by the JVM acting as an intermediary between bytecode and the underlying hardware. Using the platform diagram from the chapter as a reference, explain what would have to be true of a new device (say, a smartwatch) for it to run Java programs. Who would be responsible for making that possible and what would they need to produce?
7. The chapter mentions that the bootstrap class loader loads `java.base` which contains classes like `Object`, `String`, and `Thread`. Consider what would happen if a programmer were able to replace the bootstrap loader's version of `String` with their own custom implementation. What could go wrong?
8. The chapter describes the JVM as an abstraction layer that hides the details of the underlying hardware. This is a recurring theme throughout the textbook — abstraction allowing programmers to work at a higher level without worrying about lower-level details. List three other abstractions you have encountered in this course and for each one identify what lower-level detail it is hiding from the programmer.

Multithreading

Introduction

Modern day processors have more than just one single CPU core. In fact, many consumer grade processors have anywhere between 2 to 24 cores on a single chip! Comparing these chips to hardware that was available at the start of Java's lifecycle would be like comparing a flintlock pistol to a Stinger missile. As a result, programmers, alongside Java's ecosystem itself, have had to adapt to these changes in hardware by developing new techniques that can properly leverage the additional computational firepower. The primary innovation comes in the form of **multithreading** which aims to take advantage of these multiple cores by running tasks simultaneously.

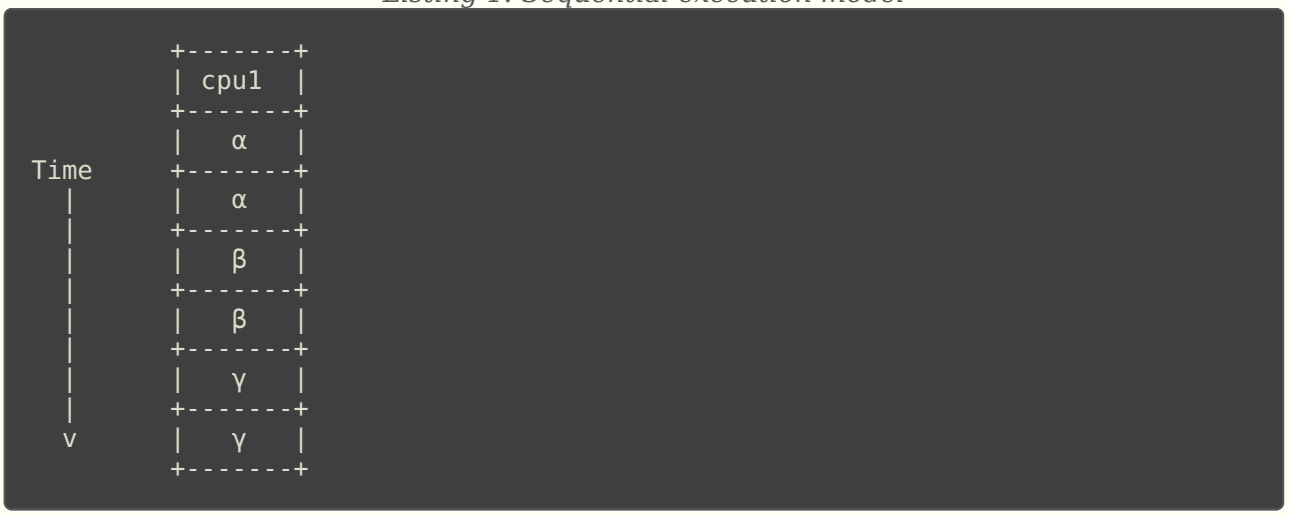
Execution Models

An *execution model* describes the behavior of how a program runs. If we focus less on the semantics/syntax of the language, but rather on the order in which instructions are processed we can define three distinct, but somwhat dependent, categories: sequential, concurrent, and parallel execution models.

Sequential

The sequential execution model is what we have been focusing on up until this point. The idea is simple, if we have three tasks, α, β, and γ, then we process α first, then β, then γ. Each task is executed until they have been completed in the order in which they arrived. We can visualize that as follows:

Listing 1: Sequential execution model



Concurrent

The concurrent execution model spices things up a little bit. Now instead of tasks being completed from beginning to end tasks can be interleaved with one another:

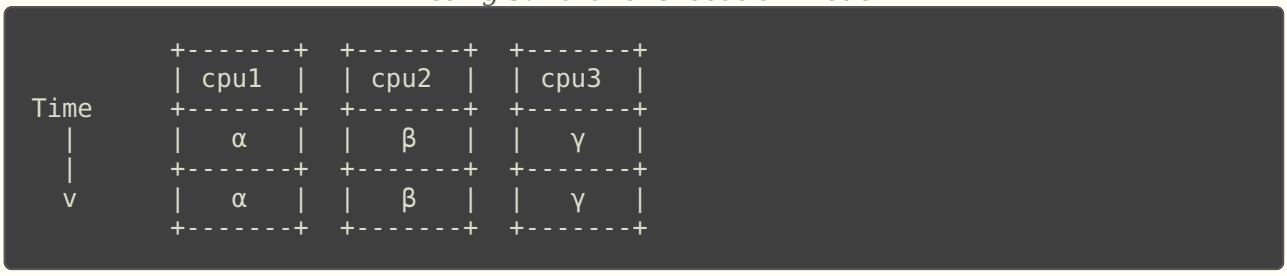
Listing 2: Concurrent execution model

This is, more or less, what is happening behind the scenes in your operating system! A little bit of your web browser task is being executed, then a background system power governer task, then your text editor task, and so on. Each portion of each task is being processed so fast that it looks like they are all happening at exactly the same time.

Parallel

Our final execution model, and the one most relevant to this chapter, is the parallel execution model. Parallel execution means that multiple tasks can be processed *at exactly the same time*. This execution model requires hardware that can handle such behavior:

Listing 3: Parallel execution model



Tasks do not need to be separate programs. They can be parts of the *same program* split along computational units.

Parallel Execution Example

Note: I highly suggest running these examples on your own machine. It is not enough to see static renders of multithreading, you need to see it running to get a feel for it.

Below is an example of parallel execution using Java's Thread API:

Listing 4: HelloWorlds prints each message in its own thread

```
 1:    import java.lang.Thread;
 2:   public class HelloWorlds extends Thread{
 3:       int id;
 4:
 5:       public void run(){
 6:           System.out.println(String.format("Thread %d: Hello Worlds!",
this.id));
 7:       }
 8:
 9:       public HelloWorlds(int id){
10:           this.id = id;
11:       }
12:
13:       public static void main(String[] args) throws Exception{
14:           int worlds = Integer.parseInt(args[0]);
15:           Thread[] ts = new Thread[worlds];
16:           for(int i = 0; i < worlds; i++){
17:               ts[i] = new HelloWorlds(i);
18:               ts[i].start();
19:           }
20:           for(Thread t : ts){
21:               try{
22:                   t.join();
23:               }catch(Exception e){
24:                   System.out.println(e);
25:                   System.exit(-1);
26:               }
27:           }
28:       }
29:   }
```

There's a lot going on here which probably makes this look insane at first glance. Let's break it down into chunks.

First, I define a class `HelloWorlds`. `HelloWorlds` is a subclass of the `Thread` class. `Thread` is how Java abstracts a *thread* of execution in a program. In sequential execution (Listing 1) there is only ever a single thread. In parallel execution there is *at least* two threads. Because `Thread` implements the `Runnable` interface it provides an implementation for the `run()` method—all user-defined worker threads override `run()` to replace it with useful work associated with its class. In `HelloWorlds`, the "useful work" is printing out a "Hello World" message.

Second, in the main method I instantiate an array of `worlds` number of `Thread` objects. Each object is instantiated with a unique `id` and stored in the

array. Each `Thread` is then started up using the `start()` method. The `start()` method informs the JVM that it is time to create a new thread of execution. This new execution thread is then queued up and will begin at the `run()` method.

Third, I call `join()` on all of the `Thread` objects. `join()` halts the calling thread's exeuction until the target thread has finished executing. The default, main thread of execution is just like any other potentially schedulable thread, so we want to make sure it doesn't stop executing before any of the child threads.

Note: Try seeing what happens when you comment out the call to `join()`!

We can see what happens below when I set `worlds` equal to 2, 5, and 10, respectively:

Listing 5: Using HelloWorlds

```
josephraskind@stargazer:/tmp/multithreading$ java HelloWorlds.java 2
Thread 0: Hello Worlds!
Thread 1: Hello Worlds!
josephraskind@stargazer:/tmp/multithreading$ java HelloWorlds.java 5
Thread 3: Hello Worlds!
Thread 0: Hello Worlds!
Thread 1: Hello Worlds!
Thread 2: Hello Worlds!
Thread 4: Hello Worlds!
josephraskind@stargazer:/tmp/multithreading$ java HelloWorlds.java 10
Thread 0: Hello Worlds!
Thread 3: Hello Worlds!
Thread 5: Hello Worlds!
Thread 7: Hello Worlds!
Thread 8: Hello Worlds!
Thread 9: Hello Worlds!
Thread 1: Hello Worlds!
Thread 2: Hello Worlds!
Thread 4: Hello Worlds!
Thread 6: Hello Worlds!
```

One thing you'll notice is that the higher thread counts are all out of order. This is proof that our tasks ran in a parallel/concurrent fashion. Multithreaded scheduling is deeply `non-deterministic`—from the perspective of the programmer there's no telling which thread will process before another!

Sychronization

Multithreading is an incredibly powerful tool that can greatly speed up the execution of complex programs, but it often comes at a significant cost to developers. The cost is that it is very difficult to write consistent, thread-safe multithreaded processes. The principle reason for this is due to the phenomena of *race conditions*.

Race Conditions

Below is a multithreaded Java program which increments a counter `upperBound` times in `numThreads` number of threads:

Listing 6: UnsafeCounter

```
 1:  import java.lang.Thread;
 2:  public class UnsafeCounter extends Thread{
 3:      static int counter;
 4:      int countTo;
 5:
 6:      public UnsafeCounter(int i){
 7:          countTo = i;
 8:      }
 9:
10:      public void run(){
11:          for(int i = 0; i < this.countTo; i++){
12:                  counter++;
13:          }
14:      }
15:
16:      public static void main(String[] args){
17:          int upperBound = Integer.parseInt(args[0]);
18:          int numThreads = Integer.parseInt(args[1]);
19:          Thread[] threads = new Thread[numThreads];
20:          for(int i = 0; i < numThreads; i++)
21:              threads[i] = new UnsafeCounter(upperBound);
22:
23:          for(Thread t : threads)
24:              t.start();
25:          try{
26:              for(Thread t : threads)
27:                  t.join();
28:              System.out.printf("Finished!\nCounter = %d\n", counter);
29:          }catch(Exception e){
30:              System.err.println(e);
31:          }
32:
33:      }
34:  }
```

Below I run `UnsafeCounter` with an `upperBound` = *10*, *1,000*, and *1,000,000* and `numThreads` = *2*:

Listing 7: UnsafeCounter

```
josephraskind@stargazer:/tmp/multithreading$ java UnsafeCounter.java 10 2
Finished!
Counter = 20
josephraskind@stargazer:/tmp/multithreading$ java UnsafeCounter.java 1000 2
Finished!
Counter = 2000
josephraskind@stargazer:/tmp/multithreading$ java UnsafeCounter.java 1000000 2
Finished!
Counter = 1969985
```

You'll notice that the call with `upperBound` equal to *10* and *1,000* worked as expected, but the call with *1,000,000* did not. Why? Something called a

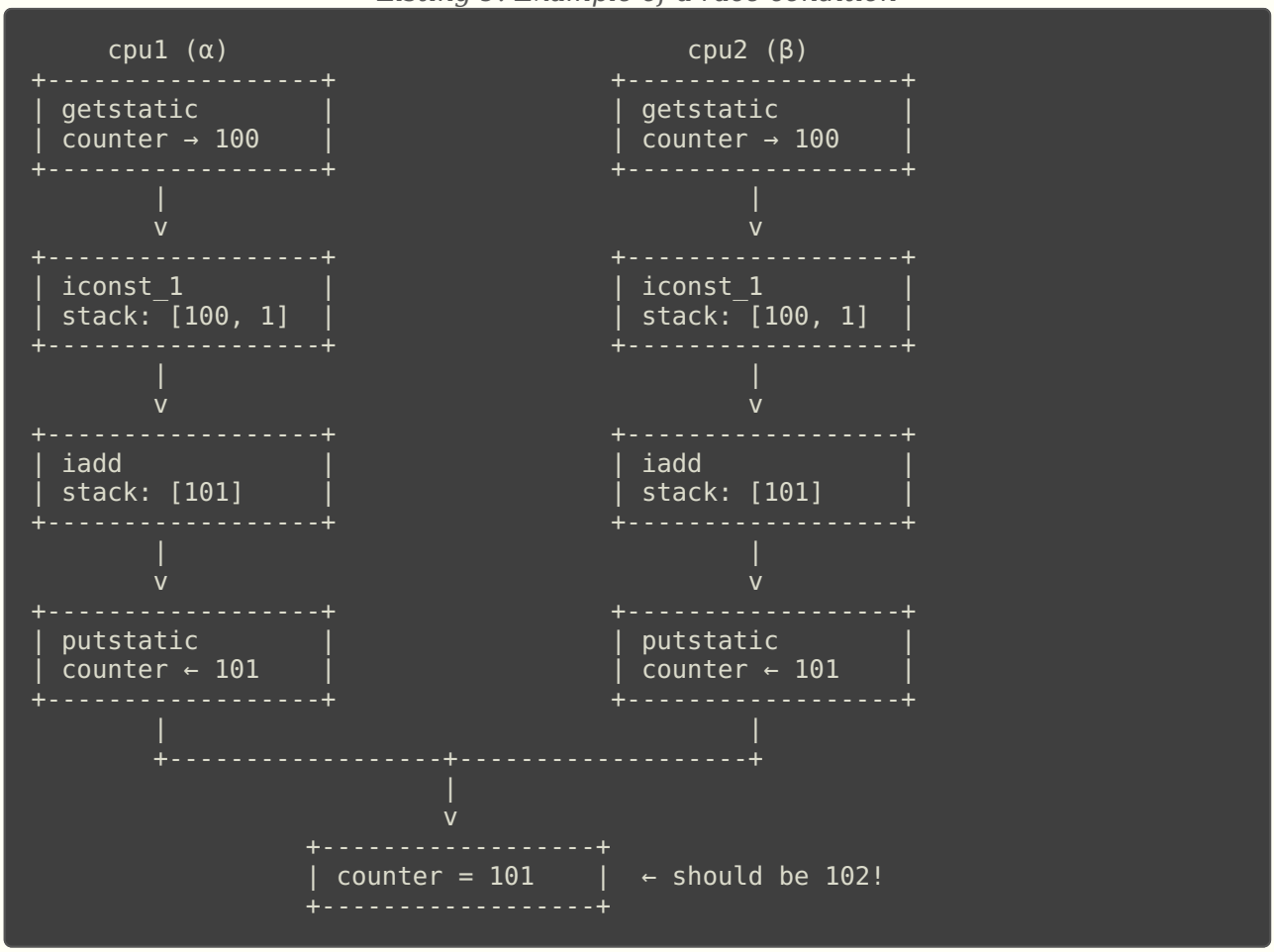
race condition occurred. The two worker threads are both updating a shared resource in the form of a class field, `counter`. The update occurs on Line 12 where `counter++` is processed. `counter++` is actually four separate instructions, which we can see in JVM bytecode:

Listing 8: The run function as JVM bytecode

```
1:  public void run();
2:    Code:
3:       0: iconst_0
4:       1: istore_1
5:       2: iload_1
6:       3: aload_0
7:       4: getfield      #7                  // Field countTo:I
8:       7: if_icmpge     24
9:      10: getstatic     #13                 // Field counter:I
10:     13: iconst_1
11:     14: iadd
12:     15: putstatic     #13                 // Field counter:I
13:     18: iinc          1, 1
14:     21: goto          2
15:     24: return
```

Lines 9-12 describe the process. This in and of itself is not a problem in a single thread of execution, but it becomes a **massive** problem with multiple threads of execution. For example, imagine a scenario where threads α and β have been scheduled to run at the same time on two separate CPUs. α processes the `getstatic` instruction and the exact same time as β—that means both threads will retrieve the same value from `counter`, say 100. α then loads a 1 on the operator stack, adds 100 to 1, and stores it in `counter`. *At the same time*, β does *exactly the same thing*. That means `counter` will store 101 despite the fact two separate additions have been executed! We can represent this visually as follows:

Listing 9: Example of a race condition



Note: What we see here is a pedagogical example. Real race conditions are often not this obvious and can be rather sinister.

How do we fix this? One way would be through the use of *synchronization techniques*.

Snychronization Techniques

Dealing with concurrency issues in computer science is not anything new. Many classic techniques that are still in use today were thought up in the 1960s and 70s. The most obvious solution to the race condition presented in Listing 9 is to enforce the rule of **mutual exclusion**. Mutual exclusion refers to using synchronization primitives to guarantee that only one thread executes within a region of that is operating on shared memory, often called the **critical section**. Using Listing 7 as an example, the critical section would be on Line 12 as that is where a resource shared between threads is being updated.

Java provides many ways of enforcing mutual exclusion. Below we will look at two: using the `synchronized` keyword and using standard library locks.

Using *synchronized*

As we have already established earlier, every object in Java is a subclass of `Object`. Every `Object` contains within it a **monitor**. This "monitor" acts as a built in synchronization primitive for every single object. Without digging into the details to much, we will briefly say that the monitor acts as a sort of talking stick: whoever holds it is allowed to speak. After unmixing our metaphores, we can say instead that whichever thread holds an object's monitor in a `synchronized` block is allowed to execute that block, whereas every other thread must **wait**. Let us alter `UnsafeCounter` to now rely on the `synchronized` keyword:

Listing 10: SafeCounter now uses synchronized

```
1:  import java.lang.Thread;
2:  public class SafeCounter extends Thread{
3:      static int counter;
4:      static Object lock;
5:      static{
6:       lock = new Object();
7:      }
8:      int countTo;
9:
10:     public SafeCounter(int i){
11:         countTo = i;
12:     }
13:
14:     public void run(){
15:         for(int i = 0; i < this.countTo; i++){
16:             synchronized(lock){
17:                 counter++;
18:             }
19:         }
20:     }
21:
22:     public static void main(String[] args){
23:         int upperBound = Integer.parseInt(args[0]);
24:         int numThreads = Integer.parseInt(args[1]);
25:         Thread[] threads = new Thread[numThreads];
26:         for(int i = 0; i < numThreads; i++)
27:             threads[i] = new SafeCounter(upperBound);
28:
29:         for(Thread t : threads)
30:             t.start();
31:         try{
32:             for(Thread t : threads)
33:                 t.join();
34:             System.out.printf("Finished!\nCounter = %d\n", counter);
35:         }catch(Exception e){
36:             System.err.println(e);
37:         }
38:
39:     }
40: }
```

Let's run the same tests we did on `UnsafeCounter`:

Listing 11: Testing SafeCounter with synchronized

```
josephraskind@stargazer:/tmp/multithreading$ java SafeCounter.java 10 2
Finished!
Counter = 20
josephraskind@stargazer:/tmp/multithreading$ java SafeCounter.java 1000 2
Finished!
Counter = 2000
josephraskind@stargazer:/tmp/multithreading$ java SafeCounter.java 1000000 2
Finished!
Counter = 2000000
```

Now we can see that setting the `upperBound` to *1,000,000* no longer produces non-determinant output. Why? This is because of two key factors. (1) We have a new static object that is shared by all instances of the SafeCounter class in the form of `lock`. (2) We define a `synchronized` block which wraps around the critical section and holds `lock`'s monitor. The Java runtime guarantees mututal exclusion in a `synchronized` block, so we can be confident that a race condition will not occur no matter how high `upperBound` is set or how many threads we create:

Listing 12: Testing SafeCounter with synchronized and large inputs

```
josephraskind@stargazer:/tmp/multithreading$ java SafeCounter.java 10000000 17
Finished!
Counter = 170000000
```

Note: You may be confused by Lines 5-7. That is what is called a "static block". It is a portion of code which is executed whenever the class is loaded in by the JVM for the first time. I could have used a simple assignment statement on Line 4, but I chose to show this syntax instead.

- Using Local Objects as Locks

  One thing to note is that the `lock` field in Listing 10 is static for a reason. For a locking mechanism to work it must be shared between all threads of execution that need to abide by mutual exclusion laws. Let's see what would happen if I tried to use a local lock instead:

Listing 13: Using a local object as a lock

```
 1:  import java.lang.Thread;
 2:  public class SafeCounter extends Thread{
 3:      static int counter;
 4:      int countTo;
 5:
 6:      public SafeCounter(int i){
 7:          countTo = i;
 8:      }
 9:
10:      public void run(){
11:          Object lock = new Object();
12:          for(int i = 0; i < this.countTo; i++){
13:              synchronized(lock){
14:                  counter++;
15:              }
16:          }
17:      }
18:
19:    // main...
20:  }
```

If I try to run my original tests:

Listing 14: Using a local object as a lock results

```
josephraskind@stargazer:/tmp/multithreading$ java SafeCounter.java 10 2
Finished!
Counter = 20
josephraskind@stargazer:/tmp/multithreading$ java SafeCounter.java 1000 2
Finished!
Counter = 2000
josephraskind@stargazer:/tmp/multithreading$ java SafeCounter.java
1000000 2
Finished!
Counter = 1332926
```

We can see that my final test breaks again! This is because each `SafeCounter` object now has its own `lock`. That means there are two, distinct `lock` variables and thus two distinct *monitors*. There is nothing to synchronize with as each instance is synchronizing with itself, solpisistically.

Standard Java Locks

There is nothing inherently *wrong* or *unsafe* about using Java's built-in `Object` monitor locks; however, they are often not the first choice of locking mechanisms that Java developers reach for in production-level projects. Instead many developers will flip through the handful of lock-centric objects that are a part of the concurrent.locks package in the standard library. Here, we will take a look at the `ReentrantLock` object:

Listing 15: Using the ReentrantLock

```
 1:   import java.lang.Thread;
 2:   import java.util.concurrent.locks.ReentrantLock;
 3:   public class SafeCounter extends Thread{
 4:       static int counter;
 5:       int countTo;
 6:       static ReentrantLock lock = new ReentrantLock(true); // Creating
lock with fair policy
 7:
 8:
 9:       public SafeCounter(int i){
10:           countTo = i;
11:       }
12:
13:       public void run(){
14:           for(int i = 0; i < this.countTo; i++){
15:               lock.lock();   // Acquiring the lock before critical section
16:               counter++;
17:               lock.unlock(); // Releasing the lock after critical section
18:           }
19:       }
20:
21:     // main...
22:   }
```

Now, instead of using the `synchronized` keyword, I've created a locking object which has explicit methods for acquiring and relinquishing the lock—what was done automatically by entering and exiting the block I must now do manually.

The advantage of these specialty locks is that they will often provide added functionality. For example, the `ReentrantLock` has an optional "fairness" policy which will do its best to grant the lock to the longest running thread. This is simply not possible when using an object's monitor. It also can be used as a "try lock" by invoking the `tryLock` method. If the lock cannot be grabbed, the thread in question can try to do some other work in the meantime.

Why Multithreading?

Some tasks are inherently suited to multithreading, such as web browsers and database management systems, and others can be broken down into smaller tasks which could benefit from multithreaded execution. Let's take an example of summing up the values in a very large array. Below we have both a sequential *and* a parallel solution:

Listing 16: ArraySummation

```java
import java.util.Arrays;
public class ArraySummation extends Thread{
    static int SIZE = 500_000_000;

    public static int parallelSum(int[] arr) {
        int numThreads = Runtime.getRuntime().availableProcessors();
        int chunkSize  = arr.length / numThreads;
        int[] partial  = new int[numThreads];

        Thread[] threads = new Thread[numThreads];

        for(int t = 0; t < numThreads; t++){
            final int threadIndex = t;
            final int start = t * chunkSize;
            final int end   = (t == numThreads - 1) ? arr.length : start
+ chunkSize;

            threads[t] = new Thread(() -> {
                    int localSum = 0;
                    for(int i = start; i < end; i++)
                        localSum += arr[i];
                    partial[threadIndex] = localSum;
            });
            threads[t].start();
        }

        try{
            for(Thread thread : threads)
                thread.join();
        }catch(Exception e){
            System.out.println(e);
            System.exit(-1);
        }
        int total = 0;
        for(int p : partial)
            total += p;
        return total;
    }

    public static int sequentialSum(int arr[]){
        int sum = 0;
        for(int v : arr)
            sum += v;
        return sum;
    }

    public static void main(String[] args){
        int arr[] = new int[SIZE];
        Arrays.fill(arr, 1);

        long start = System.currentTimeMillis();
        int sum = sequentialSum(arr);
        double duration = (System.currentTimeMillis() - start) / 1000.0;
        System.out.printf("Sequential Sum: %d (%.2f s)%n", sum,
duration);

        start = System.currentTimeMillis();
        sum = parallelSum(arr);
        duration = (System.currentTimeMillis() - start) / 1000.0;
        System.out.printf("Parallel Sum: %d (%.2f s)%n", sum,
duration);
    }
}
```

If we run both in interpretation mode:

Listing 17: ArraySummation results

```
josephraskind@stargazer:/tmp/multithreading$ java -Xint ArraySummation.java
Sequential Sum: 500000000 (6.45 s)
Parallel Sum: 500000000 (2.70 s)
```

We can see a ~2.4X speedup with the parallel execution method.

Exercises

1. Compile and run Listing 16 on your machine. Record both the sequential and parallel times. How many logical processors does your machine have? Run `Runtime.getRuntime().availableProcessors()` to find out. Is the speedup you observe close to that number? Based on what the chapter says about memory bandwidth and Amdahl's Law, explain why the speedup is lower than the number of cores.
2. The chapter shows that `counter++` compiles to four bytecode instructions. Before running Listing 7, predict what value `counter` will hold after two threads each count to 1000000. Will it be exactly 2000000, less, or more? Run it several times and record the results. Are they consistent? Explain why the output varies between runs in terms of thread scheduling.
3. The chapter introduces `synchronized` as a fix for the race condition. Add `synchronized` to the `run` method in Listing 7 and rerun it with `upperBound` set to 1000000. Does the result now equal 2000000 consistently? Then measure the time taken with and without `synchronized`. What cost does synchronization impose and why?

4. Predict the output of the following program before running it:

```
public class Predict {
    static int x = 0;
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> { for(int i = 0; i < 1000; i++) x+
+; });
        Thread t2 = new Thread(() -> { for(int i = 0; i < 1000; i++) x+
+; });
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(x);
    }
}
```

   Run it ten times and record each result. Is the output always 2000? Always less? Could it ever be more than 2000? Explain each answer.

5. The chapter states that `join` causes the calling thread to wait until the joined thread finishes. Remove both `join` calls from Listing 16 and

rerun it. What happens to the result? Explain in terms of what the main thread does after calling `start` without `join`.

6. The chapter motivates multithreading through hardware: modern CPUs have multiple cores that can execute instructions simultaneously. In your own words explain the difference between **concurrency** and **parallelism**. Is the sequential version of `ArraySummation` concurrent? Is the parallel version? Can a program be concurrent without being parallel?
7. The chapter shows that each thread in `parallelSum` writes to its own slot in the `partial` array. Why is this safe from race conditions when a shared `counter` field is not? What property of the `partial` array makes it thread-safe without needing `synchronized`?
8. The chapter introduces `Runnable` as the interface used to define thread tasks. Write a program with three threads, each printing its own name and a count from 1 to 5. Run it several times. Is the output always in the same order? What does the variability in output tell you about how the OS schedules threads?
9. Thread creation carries overhead. Design an experiment that measures this overhead by creating a thread that does nothing (`() -> {}`) and measuring how long it takes to start and join it. Run the experiment 1000 times and compute the average. How does this overhead compare to the time saved by parallelizing `ArraySummation`? Under what circumstances would the overhead make parallelism not worth it?
10. The chapter presents multithreading as a way to leverage multiple CPU cores for performance. Consider a program that reads a large file from disk and processes each line. Would multithreading help this program? What would the bottleneck be? Compare this to `ArraySummation` and explain why some workloads benefit more from multithreading than others.

I/O

Introduction

A computer would be all but useless if there was no way of retrieving the information it produced. Similarly, a computer would be incredibly difficult to use if there was no way to send information into it. This is why all computers provide components for input and output (IO). Java supplies a variety of abstractions to facilitate taking input and giving output. In this chapter, we'll go over a sampling of different objects built into the standard library.

PrintStream

First up is the PrintStream. Java's `PrintStream` is used for printing characters to files. You've actually been using a `PrintStream` since your first program! `System.out` is actually a `PrintStream` object:

Listing 1: Showing System.out's class

```
1:  public class SystemOut{
2:      public static void main(String[] args){
3:          System.out.println(System.out.getClass());
4:      }
5:  }
```

If we compile and run the above we get:

Listing 2: System.out is a PrintStream object

```
josephraskind@stargazer:/tmp/IO$ java SystemOut.java
class java.io.PrintStream
```

We can easily create a new `PrintStream` object, attach it to a file, and write to it:

Listing 3: Instantiating and using a PrintStream

```
1:  import java.io.PrintStream;
2:  import java.io.FileNotFoundException;
3:  public class UsingPrintStream{
4:      public static void main(String[] args) throws FileNotFoundException{
5:          PrintStream myStream = new PrintStream("./output.txt");
6:          myStream.println("Hello CS210! I'm in a file!");
7:      }
8:  }
```

After compiling and running we get:

Listing 4: Instantiating and using a PrintStream results

```
josephraskind@stargazer:/tmp/IO$ ls
UsingPrintStream.java
josephraskind@stargazer:/tmp/IO$ java UsingPrintStream.java
josephraskind@stargazer:/tmp/IO$ ls
output.txt  UsingPrintStream.java
josephraskind@stargazer:/tmp/IO$ cat output.txt
Hello CS210! I'm in a file!
```

During the instantiation a file descriptor is opened up by Java and is wrapped around in the trappings of the `PrintStream` object. We can see this play out in (somewhat) real time in Linux:

Listing 5: PrintStream file descriptor

```
 1:   import java.io.PrintStream;
 2:   import java.io.FileNotFoundException;
 3:   import java.lang.Thread;
 4:   public class UsingPrintStream{
 5:       public static void main(String[] args) throws FileNotFoundException,
InterruptedException{
 6:           PrintStream myStream = new PrintStream("./output.txt");
 7:           myStream.println("Hello CS210! I'm in a file!");
 8:           while(true){
 9:               Thread.sleep(10000);
10:           }
11:       }
12:   }
```

In Linux, we can track open file descriptors of processes if we have their process id's:

Listing 6: Showing file descriptors with PrintStream

```
josephraskind@stargazer:/tmp/IO$ java UsingPrintStream.java &
[1] 6515
josephraskind@stargazer:/tmp/IO$ ls /proc/6515/fd/
0  1  2  3  4
josephraskind@stargazer:/tmp/IO$ ls -l /proc/6515/fd/4
l-wx------ 1 josephraskind josephraskind 64 Aug  2 17:41 /proc/6515/fd/4 -> /
tmp/IO/output.txt
```

We can see that the latest file descriptor associated with the `UsingPrintStream` program is `4` which is tied to `/tmp/IO/output.txt`!

InputStream

`PrintStream` is an example of one of Java's `OutputStream` variants. The flip-side to the `OutputStream` is the `InputStream`. `System.in` is an example of an `InputStream` object:

Listing 7: Showing System.in's class

```
public class SystemIn{
    public static void main(String[] args){
        System.out.println(System.in.getClass());
    }
}
```

After compiling and running we get:

Listing 8: System.in is a BufferedInputStream object

```
josephraskind@stargazer:/tmp/IO$ java SystemIn.java
class java.io.BufferedInputStream
```

We can see that `System.in` is a `BufferedInputStream` object which is a descendant of `InputStream`. A quick trip to the documentation tells us that `BufferedInputStream` implements a `read()` method which will read a single byte from the connected stream. We can use `System.in` to check that out:

Listing 9: Using an InputStream

```
1:  import java.io.IOException;
2:  public class UsingInputStream{
3:      public static void main(String[] args) throws IOException{
4:          System.out.println(System.in.read());
5:      }
6:  }
```

After compiling and running:

Listing 10: Using an InputStream results

```
1:  josephraskind@stargazer:/tmp/IO$ java UsingInputStream.java
2:  a
3:  97
```

Line 2 shows my input, Line 3 shows the `println` output. We can see that the "a" character value matches the equivalent value in ASCII[1].

FileReader

The `InputStream` variants are nice, but they're often not that convenient for handling files. That's because they were made for other IO objects to use, like `FileReader`. `FileReader`, as the name suggests, is explicitly meant to be used to read files. Let's take a look at an example of using it in tandem with `BufferedReader` to print out one of my favourite Shakespeare sonnets:

Listing 11: Using FileReader to read Shakespeare

```
 1:   import java.io.*;
 2:   public class UsingFileReader{
 3:       public static void main(String[] args){
 4:           String filename = args[0];
 5:           String input;
 6:           try(BufferedReader fp = new BufferedReader(new
FileReader(filename))){
 7:               while((input = fp.readLine()) != null){
 8:                   System.out.println(input);
 9:               }
10:           }catch(IOException e){
11:               System.err.println(e);
12:           }
13:       }
14:   }
```

There's some new syntax being introduced here, so I'll explain that first. Line 6 shows a `try` keyword with a set of parentheses. Inside those parentheses we can see a variable declaration and assignment. Variables which are declared within the `try()` are guaranteed to be closed regardless of whether or not an error occurs—every declared variable must be an AutoCloseable. It's considered good style to use that syntax whenever applicable. Line 7 invokes the `BufferedReader` instance method `readLine()` which does what it says on the tin.

Here's what happens when we run the code:

Listing 12: Using FileReader to read Shakespeare results

```
josephraskind@stargazer:/tmp/IO$ java UsingFileReader.java ./sonnet_60.txt
Like as the waves make towards the pebbl'd shore,
So do our minutes hasten to their end;
Each changing place with that which goes before,
In sequent toil all forwards do contend.
Nativity, once in the main of light,
Crawls to maturity, wherewith being crown'd,
Crooked eclipses 'gainst his glory fight,
And Time that gave doth now his gift confound.
Time doth transfix the flourish set on youth
And delves the parallels in beauty's brow,
Feeds on the rarities of nature's truth,
And nothing stands but for his scythe to mow:
And yet to times in hope my verse shall stand,
Praising thy worth, despite his cruel hand.
```

The `BufferedFileReader` recognizes the end of a line due to the special `"\n"` character and converts it to a Java `String` which is what allows us to print it using `println()`.

Exercises

1. Compile and run Listing 11. Then modify it to print the total number of lines in the file alongside the contents. Next, modify it again to print

only lines that contain a specific word of your choosing. What class and method are you using to check whether a line contains the word?

2. The chapter shows that `FileReader` throws a checked `IOException`. Write a program that attempts to open a file that does not exist and catches the exception gracefully, printing a friendly error message rather than a stack trace. Then write a second version using `throws` instead of try-catch. In your own words explain the practical difference between the two approaches from the perspective of whoever calls your method.
3. The chapter introduces `BufferedWriter` for writing to files. Write a program that takes the contents of one file, converts every line to uppercase, and writes the result to a new file. Verify the output by reading the new file back with `BufferedReader` and printing its contents. What would happen if you forgot to call `close` on the writer?
4. The chapter shows that `BufferedReader` wraps a `FileReader` to add buffering. Write a program that reads the same file twice: once using `FileReader` directly character by character, and once using `BufferedReader` with `readLine`. Measure the time taken for each approach on a large file. What difference do you observe? In your own words explain what buffering does and why it improves performance when reading from disk.
5. The chapter introduces the try-with-resources syntax as a cleaner alternative to manually closing streams. Rewrite Listing 11 using try-with-resources. Then deliberately throw an exception inside the try block and verify that the file is still closed correctly by adding a print statement to a `finally` block. Compare the two approaches and explain what try-with-resources guarantees that a manual `close` call does not.

Footnotes:

1 You can use `man ascii` to see for yourself!

Vectors

Introduction

Computer programming, in the form we're familiar with it, has been around for over eighty years. All of that time has come with advancements not just in technology, but also in the way that we think about and write compute programs. The main aspect we'll be concerned with throughout the rest of this course is in *data structures*. A data structure is a way to organize and orient data within a computer program. There have been many data structures theorized, formalized, and utilized throughout the long history of computing, so we can only focus on a few in a reasonable amount of time. We'll begin with a simple, intuitive data structure which acts as a useful supplement to a basic array.

Note: Many of the remaining exercises will be having you implement the data structures in Java yourself. This means the only code I will provide for the remainder of this course is *pseudocode*.

Where Do Arrays Fail?

Arrays are incredibly useful for storing and retrieving a fixed number of elements. For example, if I'm writing software that catalogues all of the titles in a library, and I know the number of books contained within that library, it's trivial to represent every item as an element in an array, albeit a bit tedious. But what happens the moment a new book comes out and the library adds it to its physical collection? Already the old program is out of sync with reality and must be reworked to allow for the additional item. What we need is a data structure that can grow on demand without the programmer having to manage the resizing manually.

What is a Vector?

A **vector** is the exact response to the above problem. The idea is simple: combine the simplicity and contiguous behavior of arrays with automatic resizing logic. Vectors are often also called **resize-able arrays**.

Vector API

Every data structure we will be discussing has historically been defined and implemented in one form or another. Regardless of particular historical machines/libraries, there are always key functions and behaviours the remain

constant across all implementations. We call the relevant functions and their behaviors tied to a data structure its application programmer interface (API). We list the API for a vector below:

- `init`
  - Initializes a vector
- `add`
  - Appends an element to the vector
- `get`
  - Gets an element at a given index
- `set`
  - Sets an element at a given index
- `size`
  - Returns the number of elements in the vector
- `capacity`
  - Returns the underlying size of the vector
- `is_empty`
  - Indicates whether the vector is empty or not
- `free`
  - Frees the vector

Implementing a Vector

There is no one, true way of implementing any data structure—there will always be differences depending on goals, target language quirks, and programming styles. When implementing a vector it is always advisable to use the target language's array data type as the underlying array that be resized. The essential logic is quite simple:

```
1:  add(elem):
2:    if size = capacity:
3:      resize()
4:    array[size] ← elem
5:    size ← size + 1
6:    return
```

When adding elements to the array you must first check to see if there's any space. If so, the element can be added; if not, the array must be grown. The way the array is resized is down to implementation. What matters is that the original elements must be copied over into the new array.

I leave it as an exercise to the reader to further flesh out the Vector implementation in C.

Note: Remember that if arrays are going to be passed around multiple functions you should make sure to allocate them on the heap!

Vector: Pros & Cons

The choice to use one data structure over another is always an *engineering* decision. It is up to the programmer to weight the pros and cons of all of the possible data structures that could be used for a given task. Here, we enumerate the pros and cons for vectors:

Pros:

+ Random access

Cons:

- Must copy everything over on size up
- Wasted space if anomalous growth occurs

Common Vector Implementations

Vectors are very common throughout the standard libraries of programming languages, although they are often given different names. Java calls its vector an **ArrayList**, Python a **list**, Rust a **Vec**, Go a **slice**, and so on. C++ sticks to **vector** which is why I used that terminology here. Each of these languages implement their vectors differently, although the core functionality remains. The major difference will come in the form of the default array size and the algorithm which determines the new resized array length.

Exercises

1. Java's `ArrayList` is the standard library implementation of a vector. Before using it, design your own generic `Vector<T>` class in Java. It should contain at minimum an `Object[]` array field, a `size` field, and a `capacity` field. Why must the array be declared as `Object[]` rather than `T[]`? What initial values should `size` and `capacity` have after construction?
2. Implement a constructor and a `free` equivalent for your `Vector<T>`. Since Java has garbage collection there is no `free` — instead write a `clear` method that sets all array slots to `null` and resets `size` to 0. Why is setting slots to `null` important rather than just setting `size` to 0?
3. Implement `size`, `capacity`, and `isEmpty` for your `Vector<T>`. These should be straightforward one-liners. Write a small `main` that constructs a `Vector<Integer>`, then prints the result of all three methods before any elements are added.
4. Implement `get` and `set` for your `Vector<T>`. What should happen if the index provided is out of bounds? Java's `ArrayList` throws an `IndexOutOfBoundsException` — do the same in your implementation and verify it is thrown correctly with a test call.

5. Implement the `resize` helper method. It should allocate a new `Object[]` of larger capacity, copy all existing elements over using `System.arraycopy`, and update the capacity field. What factor should you grow by and why? Look up how Java's `ArrayList` grows its internal array and compare to your choice.
6. Implement `add` using the pseudocode from the chapter as a guide. Test it by adding 20 integers to a `Vector<Integer>` with an initial capacity of 4 and printing `size` and `capacity` after each insertion. How many times does `resize` get called?
7. The chapter states that "the way the array is resized is down to implementation." Implement two versions of `resize`: one that doubles the capacity and one that grows by a fixed amount of 4. Add 100 elements to each and count the total number of copy operations performed. Which strategy results in fewer copies overall?
8. The chapter lists random access as a pro of vectors. Write a program that uses `get` to access every element of a `Vector<Integer>` containing 1000 elements by index and prints their sum. Then write a second version that iterates through the same elements sequentially from index 0 to 999. Are the two approaches equivalent in terms of the number of operations performed? What does this tell you about why random access is considered a strength of the vector data structure?
9. The chapter notes that wasted space is a con of vectors. Write a program that adds 5 elements to a `Vector<Integer>` with a doubling resize strategy and prints `size` and `capacity`. How much space is wasted? Under what circumstances would this wasted space become a practical problem?
10. The chapter distinguishes between `size` and `capacity` as two separate concepts. Write a program that creates a `Vector<Integer>` with an initial capacity of 4 and adds elements one at a time, printing both `size` and `capacity` after each insertion until the vector has been resized at least three times. In your own words, explain the difference between what `size` and `capacity` represent and why a vector needs to track both.

Sorting Algorithms

Introduction

I think just about everyone reading this chapter will have, at some point, gone on an online shopping portal—whether it was for clothes, school supplies, books, or anything else is of little consequence. Often major shopping websites will have hundreds or thousands of unique items for sale, which can make it difficult to find something you *actually want to buy*. Because of this, online retailers will allow users to apply filters to their catalogue, letting them look for only such-and-such size or such-and-such brand. Additionally, these websites will also include the ability to order the items by price from highest-to-lowest or lowest-to-highest. This dynamic ordering of items within a program is handled by **sorting algorithms**—routines which have been created to place elements in either decreasing or increasing magnitude. There are many different sorting algorithms in use today, but in this class we will only focus on three: *selection sort*, *insertion sort*, and *bubble sort*.

Selection Sort

Selection sort, like all of the sorts shown in this chapter, is an intuitive sorting algorithm with a simple implementation. We divide an array of values into a sorted and unsorted portion and keep track of each portion. The sorted portion holds a space for a new value to be swapped into it and the unsorted portion is swept to find the smallest element in it. The index of the smallest element is recorded so it can be swapped into the smallest portion. A visualization can be found below:

Listing 1: Selection sort on [12, 5, 3, 10, 7]

```
Initial:  [ 12 |  5 |  3 | 10 |  7 ]
           i=0

Pass 1: find minimum in [0..4]
          [ 12 |  5 |  3 | 10 |  7 ]
            i=0  j=1  j=2  j=3  j=4
                      ^
                      min found at j=2 (value 3)

          swap arr[i=0] with arr[j=2]

          [  3 |  5 | 12 | 10 |  7 ]
            *              (sorted)

Pass 2: find minimum in [1..4]
          [  3 |  5 | 12 | 10 |  7 ]
                 i=1  j=2  j=3  j=4
                 ^
                 min found at j=1 (value 5, already in place)

          swap arr[i=1] with arr[j=1] (no change)

          [  3 |  5 | 12 | 10 |  7 ]
             *    *             (sorted)
```

In the above visualization, `i` denotes the index of the last element of the sorted portion, and `j` denotes indices of the unsorted portion. The *selection* in *selection sort* is the finding of the minimum and *selecting* it for swapping.

I leave it as an exercise for the reader to finish out the above visualization, as well as to try and implement *selection sort* in Java.

Insertion Sort

Insertion sort is very similar to *selection sort*. Here, we also partition the array into a sorted and unsorted portion. The biggest difference is that we are not selecting an element to end up at the end of the sorted portion, but we actually update the sorted portion with a new element which is propogated down stream if necessary. We can see insertion sort playing out visually below:

Listing 2: Insertion sort on [12, 5, 3, 10, 7]

```
Initial:  [ 12 |  5 |  3 | 10 |  7 ]

Pass 1: insert arr[1]=5 into sorted region [0..0]
          [ 12 |  5 |  3 | 10 |  7 ]
            i=0  j=1
          5 < 12, shift 12 right
          [  5 | 12 |  3 | 10 |  7 ]
             *             (sorted)

Pass 2: insert arr[2]=3 into sorted region [0..1]
          [  5 | 12 |  3 | 10 |  7 ]
                    j=2
          3 < 12, shift 12 right
          [  5 |  3 | 12 | 10 |  7 ]
                j=1
          3 < 5, shift 5 right
          [  3 |  5 | 12 | 10 |  7 ]
            j=0 (reached beginning, insert here)
             *    *             (sorted)

Pass 3: insert arr[3]=10 into sorted region [0..2]
          [  3 |  5 | 12 | 10 |  7 ]
                         j=3
          10 < 12, shift 12 right
          [  3 |  5 | 10 | 12 |  7 ]
                    j=2
          10 > 5, insert here
          [  3 |  5 | 10 | 12 |  7 ]
             *    *    *             (sorted)
```

`i` and `j` function identically to what they did in *selection sort*, the unsorted and sorted indices, respectively. The difference here is that you can see that when a value is placed in the sorted portion, it is swapped with other elements downstream. This guarantees that at the end of a full pass the sorted portion actually is sorted.

Again, I leave it as an exercise for the reader to finish out the above visualization, as well as to try and implement *insertion sort* in Java.

Bubble Sort

Bubble sort is likely the most intuitive sorting algorithm out there. Most beginner programmers implement bubble sort naturally without referring to any textbooks on the matter. All we do is progressively swap values of the array if they need to be swapped. Essentially we start from the beginning of the array and "float" values up. We continue to do this until we do a full pass of the array without a single swap:

Listing 3: Bubble sort on [12, 5, 3, 10, 7]

```
Initial:  [ 12 |  5 |  3 | 10 |  7 ]

Pass 1: bubble largest element to end
          [ 12 |  5 |  3 | 10 |  7 ]
            j=0  j=1
          12 > 5, swap
          [  5 | 12 |  3 | 10 |  7 ]
                 j=1  j=2
          12 > 3, swap
          [  5 |  3 | 12 | 10 |  7 ]
                       j=2  j=3
          12 > 10, swap
          [  5 |  3 | 10 | 12 |  7 ]
                            j=3  j=4
          12 > 7, swap
          [  5 |  3 | 10 |  7 | 12 ]
                                  *   (sorted)

Pass 2: bubble largest remaining to end
          [  5 |  3 | 10 |  7 | 12 ]
            j=0  j=1
          5 > 3, swap
          [  3 |  5 | 10 |  7 | 12 ]
                 j=1  j=2
          5 < 10, no swap
          [  3 |  5 | 10 |  7 | 12 ]
                       j=2  j=3
          10 > 7, swap
          [  3 |  5 |  7 | 10 | 12 ]
                            j=3  j=4
          10 < 12, no swap
          [  3 |  5 |  7 | 10 | 12 ]
                             *    *  (sorted)
```

Again, I leave it as an exercise for the reader to finish out the above visualization, as well as to try and implement *bubble sort* in Java.

Why Are There Different Algorithms?

We've seen above that every algorithm shown above *does exactly the same thing* in terms of results. We always end with an array that was sorted in increasing order. Why then do we bother using more than one algorithm? The answer is found in an analysis of *algorithmic complexity*, which I will have more to say on a bit later. Suffice it to say that sometimes a certain algorithm will be better for one job in certain situations than another.

As an example, we will compare *selection sort* and *bubble sort* in two different contexts. One where our numbers are already sorted, and another where they are sorted in reverse order. Each case will look at arrays with half a million elements (from [1-500,000] and [500,000, 1]).

You can see the Java program I used to test the algorithms here:

Listing 4: Testing Sorting algorithms

```
 1:   import java.util.Arrays;
 2:   public class Sorting{
 3:       static final int SIZE = 500_000;
 4:
 5:       public static void swap(int[] arr, int a, int b){
 6:           int temp = arr[a];
 7:           arr[a] = arr[b];
 8:           arr[b] = temp;
 9:       }
10:
11:       public static void main(String[] args){
12:           if(args.length != 2){
13:               System.out.printf("\"selection\" for Selection
Sort%n\"insertion\" for Insertion Sort%n\"bubble\" for Bubble Sort%n\"desc\"
for descending array %n \"asc\" for ascending array%n");
14:               System.exit(1); //Error out
15:           }
16:           int[] arr = new int[SIZE];
17:           switch(args[1]){
18:           case "desc":
19:               for(int i = 0; i < SIZE; i++){
20:                   arr[i] = SIZE - (i);
21:               }
22:               break;
23:           case "asc":
24:               for(int i = 0; i < SIZE; i++){
25:                   arr[i] = i+1;
26:               }
27:               break;
28:           default:
29:               throw new RuntimeException("desc or asc!");
30:           }
31:
32:           long start = System.currentTimeMillis();
33:           switch(args[0]){
34:           case "selection":
35:               System.out.printf("Selection Sort: ");
36:               SelectionSort.sort(arr);
37:               break;
38:           case "insertion":
39:               System.out.printf("Insertion Sort: ");
40:               InsertionSort.sort(arr);
41:               break;
42:           case "bubble":
43:               System.out.printf("Bubble Sort: ");
44:               BubbleSort.sort(arr);
45:               break;
46:           default:
47:               throw new RuntimeException("No valid sort selected");
48:           }
49:           double duration = (System.currentTimeMillis() - start) / 1_000.0;
50:           System.out.printf("%.2f s%n", duration);
51:       }
52:   }
```

Let's see how *bubble sort* and *selection sort* do with a descending list:

Listing 5: Descending list sort

```
josephraskind@stargazer:/tmp/sorting$ javac Sorting.java; java Sorting
selection desc
Selection Sort: 117.79 s
josephraskind@stargazer:/tmp/sorting$ javac Sorting.java; java Sorting bubble
desc
Bubble Sort: 93.51 s
```

We can see there's some parity in the performance. Bubble sort is a bit faster but proportionally only a ~1.3X speedup. If I were to run this test 100 times we would likely see that decrease on average as well. What about on an ascending list:

Listing 6: Ascending list sort

```
josephraskind@stargazer:/tmp/sorting$ javac Sorting.java; java Sorting
selection asc
Selection Sort: 33.76 s
josephraskind@stargazer:/tmp/sorting$ javac Sorting.java; java Sorting bubble
asc
Bubble Sort: 0.02 s
```

Wow, that's a ~1700X speedup! This is because the list is already sorted, so the bubble sort never has to float up a single value, we only loop through the array once whereas selection sort still has to loop through every element in the array.

All sorting algorithms have this kind of give and take.

Exercises

1. The chapter shows the first three passes of selection sort on `[12, 5, 3, 10, 7]`. Complete the visualization for passes 4 and 5 by hand, showing the values of `i` and `j` at each step. Then implement selection sort in Java and verify your hand trace matches the output of your program.
2. The chapter shows the first three passes of insertion sort on `[12, 5, 3, 10, 7]`. Complete the visualization for passes 4 and 5 by hand. Then implement insertion sort in Java and verify your hand trace matches the output of your program.
3. The chapter shows the first two passes of bubble sort on `[12, 5, 3, 10, 7]`. Complete the visualization for passes 3, 4, and 5 by hand, noting which comparisons result in swaps and which do not. Then implement bubble sort in Java and verify your hand trace matches the output of your program.
4. Implement all three sorting algorithms in a single Java program. Add a counter to each that tracks the total number of comparisons made when sorting `[12, 5, 3, 10, 7]`. Which algorithm makes the fewest comparisons on this input? Does the winner change if you sort the already-sorted array `[3, 5, 7, 10, 12]`?
5. The chapter states that bubble sort performs poorly on nearly-sorted arrays in its basic form. Add an optimization: if no swaps were made

during an entire pass, the array is already sorted and the algorithm can terminate early. Implement this optimization and verify it causes bubble sort to terminate in a single pass on `[3, 5, 7, 10, 12]`.

6. Implement selection sort on a `String[]` instead of an `int[]`. Use `String`'s `compareTo` method to compare elements. Sort the array `["banana", "apple", "date", "cherry", "elderberry"]` and print the result. What does `compareTo` return and how do you use its return value to determine ordering?
7. The chapter motivates sorting through the online shopping analogy. Consider a list of `BankAccount` objects from earlier in the course. Write a program that sorts an array of `BankAccount` objects by balance from lowest to highest using insertion sort. What challenge do you encounter that did not exist when sorting integers?
8. The chapter shows that sorting requires comparing elements against each other. For each of the three algorithms, count the exact number of comparisons made when sorting `[12, 5, 3, 10, 7]` by hand using the visualizations from the chapter. Then count the comparisons again on the reverse-sorted array `[12, 10, 7, 5, 3]` and on the already-sorted array `[3, 5, 7, 10, 12]`. Which algorithm's comparison count changes the most between the best and worst case? Which changes the least? What does this suggest about which algorithm you would choose if you knew your data was already nearly sorted?
9. The chapter shows that sorting is motivated by the need to order data for human consumption. Consider the following scenario: a teacher has an array of student grades and wants to find the median grade. Write a program that uses one of the three sorting algorithms to find the median of `[72, 95, 61, 88, 79, 55, 90]`. Why is sorting a useful first step for finding the median?
10. The chapter introduces three sorting algorithms that all sort in place—they rearrange the elements of the original array rather than creating a new one. Write a version of selection sort that instead returns a new sorted array without modifying the original. What additional steps are required? Verify that the original array is unchanged after calling your method.

Stacks & Queues

Introduction

Imagine a scenario where you've put off doing the dishes for far too long and you need a plate to eat your daily bread. You think, "Oh God, I can't keep living like this." Before it's time to begin running the water you pull your sleeves up to your forearms, wearing out the last bit of plasticity remaining in your elastic cuffs. The joyful mundanity of scrubbing a week's worth of solidifying crust feels like a secular baptism. Every dish you polish and dry is sat atop one another such that the most recently cleaned is at the peak. Once you're done, you take the plate resting at the summit, the one you just finished drying off. You have just implemented and utilized a **stack**.

Stacks

Stacks are one of the simplest and most foundational data structures in computing. In fact, I've already used the word "stack" many times already throughout this course when referring to dynamic memory allocated by the compiler. A stack guarantees that the order in which elements are retrieved, or *popped* off, are in the reverse order in which they were placed, or *pushed*, on the stack.

Stack API

Like vectors, stacks can be implemented a variety of ways and may have different features depending on the library being used. The basic functionality is as follows:

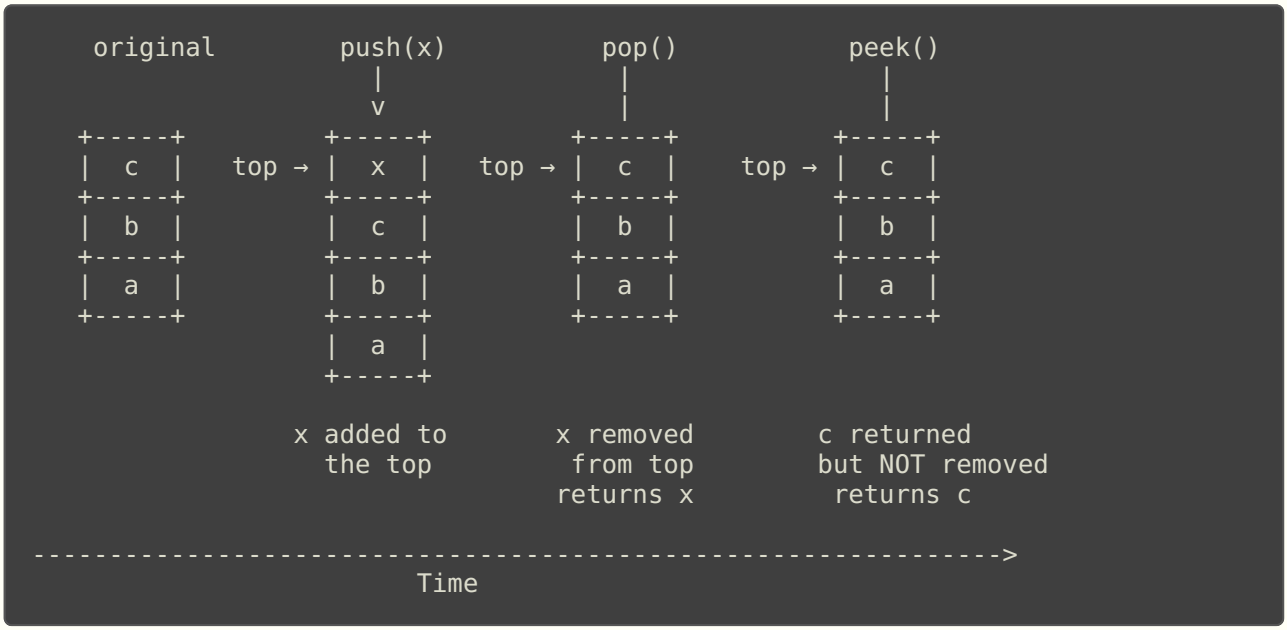
- `init`
  - Initializes a stack
- `push`
  - Pushes an element on the top of the stack
- `peek`
  - Retrieves the top element on the stack but does not remove it
- `pop`
  - Removes and returns the top element on the stack
- `isEmpty`
  - Indicates whether the stack is empty or not
- `free`
  - Frees the stack

Implementing a Stack

All a stack must do is maintain last-in-first-out (LIFO) order. This gives a wide variety of possible backend implementations for the stack in question. Stacks may have a fixed or a dynamic capacity—again, this is decided by need; traditionally, stacks in standard libraries will have dynamic capacity logic so as to best apply to all general cases. It is left as an exercise for the reader to implement the stack logic.

Using a Stack

The diagram below shows the behavior of the main three stack functions (push, pop, and peek):



We begin with a stack that has the elements `(c, b, a)` where `c` is at the top of the stack (the elements were pushed in alphabetical order). A call to `push(x)` places `x` at the top of the stack. A call to `pop()` removes the element at the top of the stack, now `x`, and returns it. A call to `peek()` takes a look at the new element at the top of the stack, now `c` again, and returns that element *without removing it from the stack*.

Case Study: Method Calls, Parsing and Reverse Polish Notation

When I first started teaching data structures years ago, I found that stacks came across as far too abstract for students to connect with as something genuinely useful. Why would I ever need anything in the *reverse* order in which I added them? It's not immediately intuitive. First, think back to how method calls work in Java:

Listing 1: Method call order

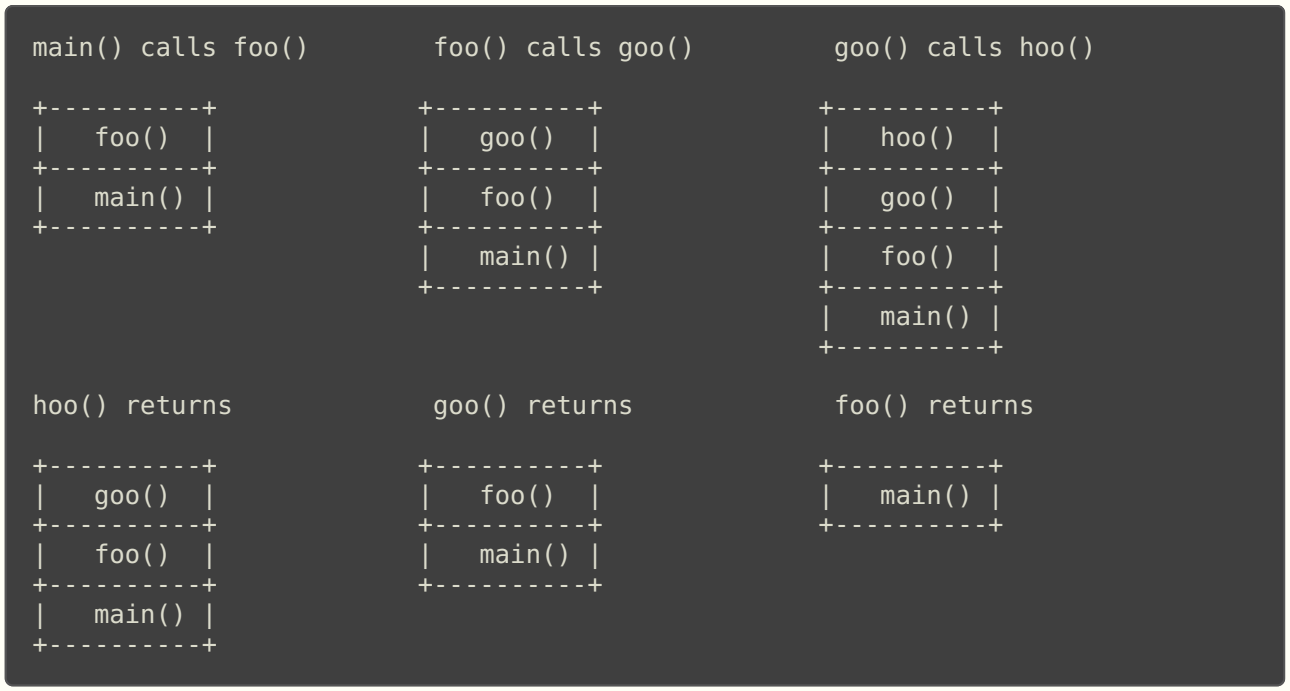
```
1:  public class MethodCalls{
2:      public static void hoo(){
3:          System.out.printf("entering hoo()\n");
4:          System.out.printf("leaving hoo()\n");
5:      }
6:
7:      public static void goo(){
8:          System.out.printf("entering goo()\n");
9:          hoo();
10:         System.out.printf("leaving goo()\n");
11:     }
12:
13:     public static void foo(){
14:         System.out.printf("entering foo()\n");
15:         goo();
16:         System.out.printf("leaving foo()\n");
17:     }
18:
19:     public static void main(String[] args){
20:         System.out.printf("entering main()\n");
21:         foo();
22:         System.out.printf("leaving main()\n");
23:
24:     }
25: }
```

You should know the order in which Listing 1 prints, but I'll print out the results anyway:

Listing 2: Method call order results

```
1:  josephraskind@stargazer:/tmp/stacks_and_queues$ java MethodCalls.java
2:  entering main()
3:  entering foo()
4:  entering goo()
5:  entering hoo()
6:  leaving hoo()
7:  leaving goo()
8:  leaving foo()
9:  leaving main()
```

We get a nice symmatry in the "entering" and "leaving" print outs. Why? Well, it's because the compiler maintains a *function call* ***stack***. We can see a visualization of this below:



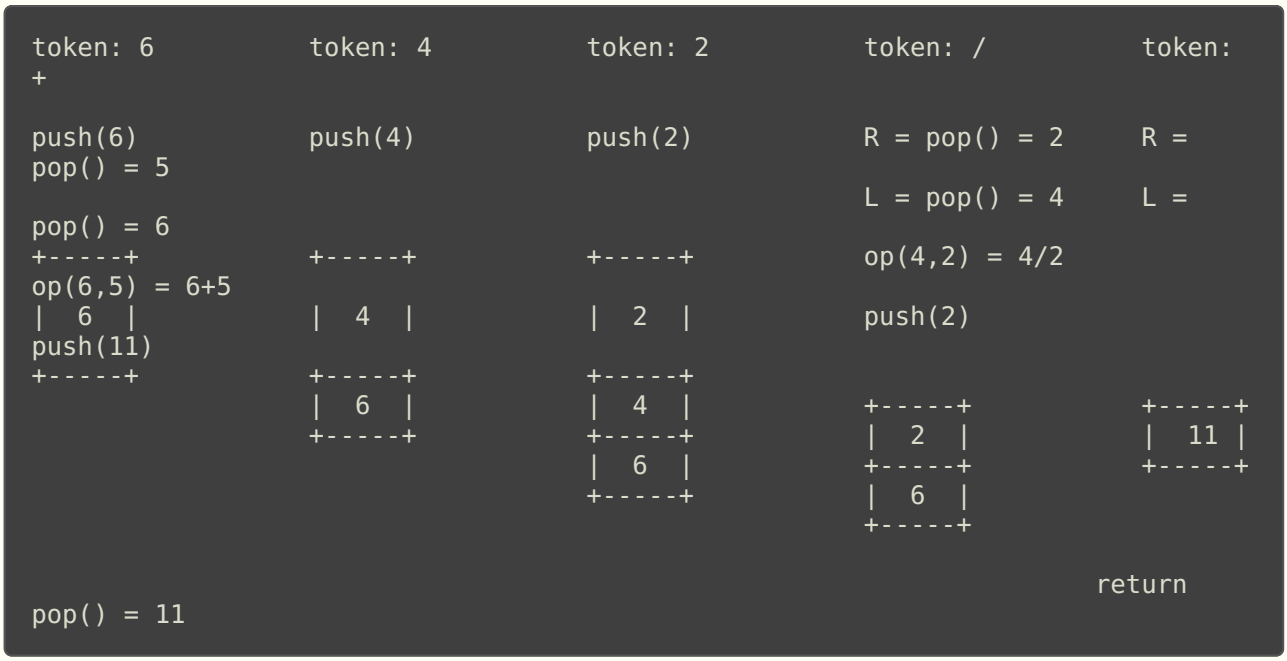
Each subsequent call *pushes* the previous function's context onto the stack. Every return *pops* the recent call off of the stack. The same is done with local variables which is why they go out of scope within the lifetime memory semantics.

Stacks can also be used for parsing tasks. If you remember back to when we discussed operators, I mentioned there were three ways we can represent the syntax for binary operators: prefix, infix, and postfix. We'll focus on postfix notation, or *Reverse Polish Notation* (RPN), as it is perfectly suited for being handled by a stack.

Take the string `6 + 4 / 2`. Without being aware of operator precedence, which a computer is not natively aware of, it is impossible to discern whether or not the user wanted to add 6 and 4 and then divide by 2, or to divide 4 by 2 then add 6. In RPN, there is no ambiguity. If we take classic precedence as a given then we can rewrite that string `6 4 2 / +`. The algorithm for RPN is simple:

```
 1:  evaluate(tokens):
 2:    for token in tokens:
 3:      if token is VALUE:
 4:        stack.push(token)
 5:      if token is OPERATOR:
 6:        op ← token
 7:        operand_R ← stack.pop()
 8:        operand_L ← stack.pop()
 9:        push(op(operand_L, operand_R))
10:    return stack.pop()
```

We can see this evaluation play out visually:

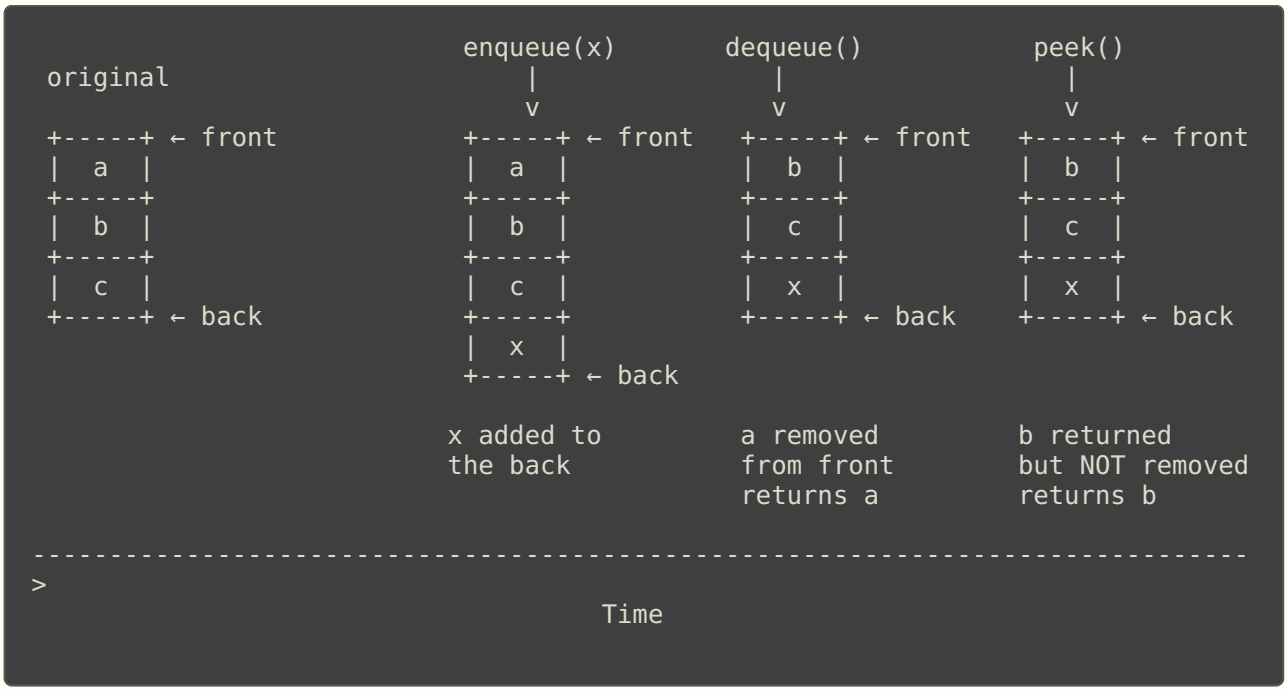


I leave it as an exercise to implement RPN in Java.

Note: RPN was used in one of the first computers, Konrad Zuse's Z3!

Queues

Queues are nearly identical to stacks with one key difference: instead of maintaining LIFO order they maintain FIFO order, or *first*-in-*first*-out. The use-case for a queue are a bit more intuitive. For example, if there are many users who wish to run their programs on a remote server each request could be entered into a queue such that the requests are processed in the order in which they were received.



It does not take much turn a stack into a queue. I leave that as an exercise for the reader.

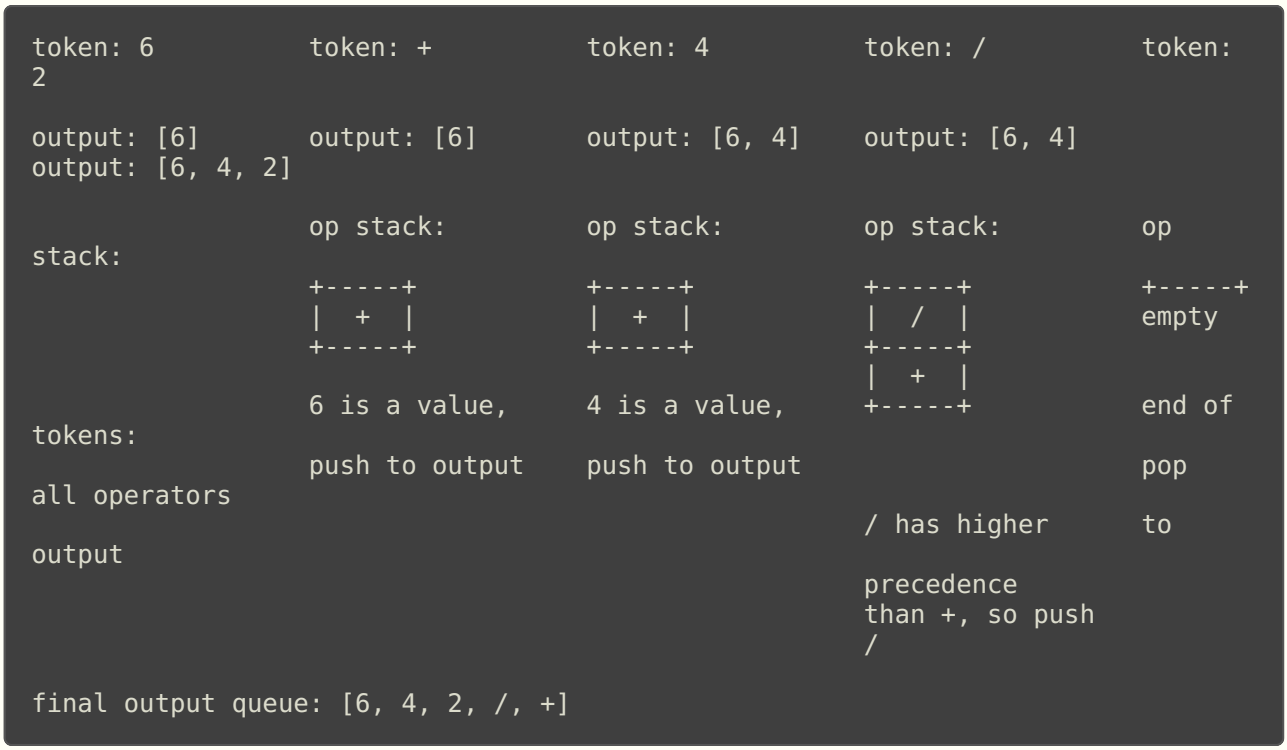
Case Study: The Shunting Yard Algorithm

Following up on the case study for parsing postfix operators with a stack, we can use a stack in tandem with a queue to convert infix notation to postfix notation. The following algorithm was developed by Edsger Dijkstra in the early 1960s[1]:

Listing 3: Shunting Yard Algorithm

```
 1:   while there are tokens to be read:
 2:       read a token
 3:       if the token is:
 4:       - a number:
 5:           put it into the output queue
 6:       - a function:
 7:           push it onto the operator stack
 8:       - an operator o1:
 9:           while (
10:               there is an operator o2 at the top of the operator stack
which is not a left parenthesis,
11:               and (o2 has greater precedence than o1 or (o1 and o2 have
the same precedence and o1 is left-associative))
12:           ):
13:               pop o2 from the operator stack into the output queue
14:           push o1 onto the operator stack
15:       - a ",":
16:           while the operator at the top of the operator stack is not a
left parenthesis:
17:                pop the operator from the operator stack into the output
queue
18:       - a left parenthesis (i.e. "("):
19:           push it onto the operator stack
20:       - a right parenthesis (i.e. ")"):
21:           while the operator at the top of the operator stack is not a
left parenthesis:
22:               {assert the operator stack is not empty}
23:               /* If the stack runs out without finding a left parenthesis,
then there are mismatched parentheses. */
24:               pop the operator from the operator stack into the output
queue
25:           {assert there is a left parenthesis at the top of the operator
stack}
26:           pop the left parenthesis from the operator stack and discard it
27:           if there is a function token at the top of the operator stack,
then:
28:               pop the function from the operator stack into the output
queue
29:   /* After the while loop, pop the remaining items from the operator stack
into the output queue. */
30:   while there are tokens on the operator stack:
31:       /* If the operator token on the top of the stack is a parenthesis,
then there are mismatched parentheses. */
32:       {assert the operator on top of the stack is not a (left) parenthesis}
33:       pop the operator from the operator stack onto the output queue
```

Below represents the transformation of the above mentioned `6 + 4 / 2` into RPN:



I leave it as an exercise for the reader to implement this in Java.

Stack/Queue: Pros & Cons

Pros:

+ Keeps well-defined order

Cons:

- Elements cannot be easily retrieved without breaking order

Exercises

1. Implement a stack in Java using the `Vector<T>` you implemented in the previous chapter as the underlying data structure. Your stack should support all of the operations listed in the Stack API: a constructor, `push`, `peek`, `pop`, and `isEmpty`. Since your `Vector<T>` already handles dynamic resizing, your stack implementation should require very little additional logic. Make your stack generic: `Stack<T>`.
2. Using your stack implementation, write a program that replicates the function call order behavior shown in Listing 1. Each time a function is "entered" push its name onto the stack and print the stack contents. Each time a function "leaves" pop it off and print the stack contents. Verify the output matches the diagram in the chapter.

3. Implement the RPN evaluator from the pseudocode in the chapter using your `Stack<Integer>`. Test it with the following expressions and verify each result:
    - `6 4 2 / +` → 8
    - `3 4 * 2 +` → 14
    - `5 1 2 + 4 * + 3 -` → 14
4. Implement a queue in Java. Consider what changes need to be made relative to your stack implementation to enforce FIFO rather than LIFO order. Your queue should support a constructor, `enqueue`, `dequeue`, `peek`, and `isEmpty`. Make your queue generic: `Queue<T>`.
5. The chapter states that it "does not take much to turn a stack into a queue." Explain in your own words what the key structural difference is between the two. What changes in terms of where elements are added and removed?
6. Using your `Queue<String>` implementation, simulate a print queue. Write a program that enqueues five print jobs represented as strings, then dequeues and prints them one at a time. Verify that they are processed in the order they were submitted.
7. Implement the Shunting Yard Algorithm from Listing 3 in Java using your `Stack` and `Queue` implementations. Test it on `6 + 4 / 2` and verify the output queue matches `[6, 4, 2, /, +]`. Then pipe the result into your RPN evaluator from exercise 3 and verify the final result is 8.
8. The chapter lists "elements cannot be easily retrieved without breaking order" as a con of stacks and queues. Write a program that demonstrates this limitation by attempting to find a specific element in a `Stack<Integer>` without permanently destroying its contents. What strategy can you use to restore the stack after searching it? How does this compare to finding an element in a `Vector<Integer>` using `get`?
9. The chapter presents two case studies showing stacks used for real computational tasks: function call management and RPN evaluation. Write a third use case of your own by implementing a program that uses your `Stack<Character>` to check whether a string of parentheses is balanced. For example `(((())))` and `(()(()))` are balanced while `(()` and `)(` are not. The algorithm is: push every opening parenthesis onto the stack and pop when a closing parenthesis is encountered — if the stack is empty when you try to pop, or non-empty when the string ends, the parentheses are unbalanced.
10. Implement a circular buffer in Java with a fixed capacity of your choosing. Your implementation should support `enqueue`, `dequeue`, `peek`, `isEmpty`, and `isFull` operations. Unlike your `Vector<T>` implementation, the circular buffer should not resize — when it is full, attempting to enqueue should throw an exception. Use two integer indices, a read pointer and a write pointer, to track the front and back of the buffer, and use the modulo operator to wrap them around the array. Test your implementation by filling the buffer to capacity, dequeuing a few elements, and then enqueuing new ones into the freed slots to verify the wrap-around behavior works correctly.
11. Java's standard library provides `java.util.Stack` and `java.util.ArrayDeque` which implement stack and queue behavior respectively. Rewrite exercise 3 using `java.util.Stack<Integer>` instead of your own implementation. Then rewrite exercise 6 using `java.util.ArrayDeque<String>`. Compare the API of the standard

library classes to your own—what methods do they provide that yours does not? What does this tell you about the value of implementing data structures yourself before using library versions?

Footnotes:

1 This psuedocode was taken from the Wikipedia article on the topic.

Linked Lists

Introduction

Every data structure we've encountered so far has shared one serious limitation: they have all needed to be defined with contiguous stretches of memory. This can be a major issue when dealing with variable sized data—remember back to one of the major cons of vectors. That then raises the question: "Can we create a data structure which can grow arbitrarily large, but does not require large swathes of connected strips of memory?" The answer: *linked lists*.

What is a Linked List?

Linked lists were first formalized in a work published by Allen Newell, Cliff Shaw, and Herbert A. Simon in 1957 titled *Programming the Logic Theory Machine*. Newell, Shaw, and Simon had previously worked on establishing a heuristic search-based algorithm which would mimic human-like proof discovery with symbolic symbol sets. The biggest barrier they encountered when trying to practically implement the Logic Theory Machine was that, by definition, the search would take an amount of time and space not known *a priori*—human beings never know how long it might take to discover proofs, so the algorithmic representation was equally haunted by that reality. This meant that storing all of the different possible branches and twists the search algorithm went on was beyond the capabilities of known data structures and programming language constructs. Their solution was to craft a new programming language, IPL, which was focused around the maintenance of a revolutionary data structure they called a "list".

The IPL list had to satisfy the following constraints:

1. "There should be no restriction on the number of different lists of items of information to be stored. This number should not have to be decided in advance; that is, it should be possible to create new lists at will during the course of computation."
2. "There should be no restriction on the nature of the items in a list. These might range from a single symbol or number to an arbitrary list. Thus, it should be possible to make lists, lists of lists, lists of lists of lists, etc."
3. "It should be possible to add, delete, insert, and rearrange items of information in a list at any time and in any way. Thus, for example, one should be able to add to the front of a list as well as to the end."
4. "It should be possible for the same item to appear on any number of lists simultaneously."

And thus, what we call a "linked list" was born!

Anatomy of a Linked List

I've found that, for many students, linked lists mark a major point where data structures enter a level of what can feel like incomprehensible abstraction. The most important thing to keep in mind when dealing with linked lists is that they are really only constituted of two components: **data** and a *pointer* to the **next** element in the list. The "elements" that make up linked lists are called **nodes**. A node looks like the following:

```
+--------+--------+
|  data  |  next  |
+--------+--------+
```

The node stores two values: the key **data** associated with that node and the pointer that will be used to connect it to the **next** node in the chain.

Now that we know what a node looks like, we can visualize what a full linked list looks like:

Listing 1: Singly linked list diagram

```
+------+------+    +------+------+    +------+------+    +------+------+
+------+------+
|  17  |   *--+--->|   4  |   *--+--->|  91  |   *--+--->|  62  |   *--+---
>|   1  | NULL |
+------+------+    +------+------+    +------+------+    +------+------+
+------+------+
 head
```

Here we see a linked list which is storing integer values. This list is comprised of five nodes all of which have up to a single connection to another node. We call this sort of linked list a *singly*-linked list for that very reason. The advantage of the linked list is that, since it is entirely pointer, or *reference*, based none of the nodes must be connected contiguously in memory. For example, Listing 1 could be represented in memory like this:

Listing 2: Singly linked list in memory

```
Address
0x008  +------+------+
       |  17  |   *--+---+
       +------+------+   |
                         |
0x031  +------+------+<--+
       |   4  |   *--+---+
       +------+------+   |
                         |
0x057  +------+------+<--+
       |  91  |   *--+---+
       +------+------+   |
                         |
0x102  +------+------+<--+
       |  62  |   *--+---+
       +------+------+   |
                         |
0x210  +------+------+<--+
       |   1  | NULL |
       +------+------+

head → 0x008
```

Again, in a linked list nodes can be connected between disparate locations in the address space.

One thing that you will have noticed is the use of the phrase **head**. The head pointer refers to the start of the linked list. Without a head pointer there would be no way to "enter" into the linked list.

Linked List API

We can define a simple linked list API as follows:

- `init`
  - Initializes a linked list
- `append`
  - Adds an element to the end of the linked list
- `insert`
  - Inserts an element at the desired index of the linked list
- `search`
  - Retrieves the desired element from the linked list
- `get`
  - Retrieves the element at the desired index of the linked list
- `remove`
  - Removes the element at the desired index of the linked list
- `isEmpty`
  - Indicates whether the linked list is empty or not
- `free`
  - Frees the linked list

Adding Elements to a Linked List

Linked list implementations often allow for two ways of adding elements: appending and inserting.

Append

Appending elements to a linked list is very simple. The following pseudocode describes the process:

Listing 3: append pseudocode

```
append(elem):
  current_node ← head
  while current_node.next not = null:
    current_node ← current_node.next
  current_node.next ← elem
```

We begin at the head, or the entrace to the linked list, work our way to the last node, and add the desired element to the chain. I leave it as an exercise to the reader to implement the function in Java.

Note: The pseudocode in the append function is missing an edge case check. Can you figure out what it is?

Insert

Appending elements to the list is a special case of `insert` where the insertion index is the beyond the end of the list. Here I will define the pseudocode for insertion in the middle of a linked list:

Listing 4: insert pseudocode — here we assume the head is at index 0

```
insert(index, elem):
  current_node ← head
  prev_node ← null
  c_i ← 0
  while c_i not = index:
    prev_node ← current_node
    current_node ← current_node.next
    c_i ← c_i + 1
  prev_node.next = elem
  elem.next = current_node
```

I leave it as an exercise to the reader to implement this in Java.

Note: Like the append function, the insert function is missing two distinct case checks. Can you figure out what they are?

Searching a Linked List

Searching a linked list follows the exact same algorithm as appending/inserting with the exception of checking equivalency at each data item.

Removing Elements from a Linked List

Removing elements from a linked list follow a similar algorithm to insertion. The only difference being is you want to break the next node chain and free up the node that is marked for deletion.

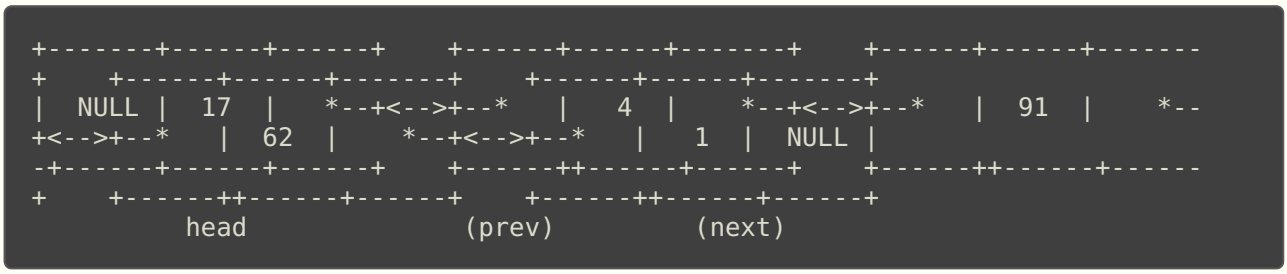
Listing 5: delete pseudocode — here we assume the head is at index 0

```
 1:   delete(index):
 2:     current_node ← head
 3:     prev_node ← null
 4:     c_i ← 0
 5:     while c_i not = index:
 6:       prev_node ← current_node
 7:       current_node ← current_node.next
 8:       c_i ← c_i + 1
 9:     prev_node.next = current_node.next
10:     free(current_node)
```

Note: Again, the delete function is missing two distinct case checks. Can you figure out what they are?

Doubly Linked Lists

The references to linked lists above were all to *singly*-linked lists as each node had only one next pointer. We can increase the complexity of our linked lists by adding a second pointer to each node which points to the previous node in the chain. The diagram below shows a *doubly*-linked list variant of the singly-linked list diagram used above:



The introduction of a second pointer requires slightly more upkeep and overhead from the algorithm perspective. Again, I leave it as an exercise to the reader to implement a doubly-linked list using the singly-linked list as a guide.

Linked List: Pros & Cons

Pros:

+ Can grow arbitrarily large
+ Appending can be made a constant time operation

Cons:

- Can be costly to search through

Exercises

1. Implement a singly linked list node as an inner class in Java. It should contain an integer data field and a `next` reference. Then implement a `LinkedList` class with a `head` field initialized to `null` and a constructor that sets up an empty list.
2. Implement `append` from Listing 3 in Java. The chapter notes that the pseudocode is missing an edge case check — identify what it is and handle it in your implementation. Test by appending the values `(17, 4, 91, 62, 1)` and printing each node.
3. The chapter notes that Listing 4 is missing two distinct case checks. Identify both before implementing `insert` in Java. Hint: think about what happens when inserting at index 0 and when inserting beyond the end of the list.
4. Implement `delete` from Listing 5 in Java. The chapter notes it is also missing two case checks — identify and handle them. Test by deleting the head, the tail, and a middle element from your list.
5. Implement `search` for your linked list. It should traverse the list and return a reference to the first node whose data matches the search value, or `null` if not found. What is the worst case number of nodes that must be visited to find an element?
6. The chapter lists "appending can be made a constant time operation" as a pro of linked lists. Currently `append` traverses the entire list to find the last node. Add a `tail` reference to your `LinkedList` class that always points to the last node and update `append`, `insert`, and `delete` to maintain it correctly. Verify that `append` no longer needs to traverse the list.
7. The chapter shows in Listing 2 that linked list nodes can be stored at non-contiguous memory addresses. Write a program that allocates five nodes and prints the identity hash code of each using `System.identityHashCode(node)`. Are they contiguous? What does this confirm about how linked lists use memory compared to arrays?

8. Java's garbage collector reclaims unreachable objects automatically. When you delete a node from your linked list by removing all references to it, does it need to be explicitly freed? Verify by deleting all nodes from your list one at a time and confirming that your `LinkedList` class correctly sets `head` to `null` when empty. What would happen in C if you forgot to `free` a deleted node?
9. Implement a doubly linked list in Java using the singly linked list as a guide. Each node should have both a `next` and a `prev` reference. Update `append`, `insert`, and `delete` to maintain both references correctly. What additional steps are required compared to the singly linked list versions?
10. The chapter lists "can be costly to search through" as a con of linked lists. Write a program that creates a linked list of 1000 integers and searches for a value stored at the very end. Count the number of nodes visited during the search and print it. Then repeat the same experiment with a value stored at the front. What does the difference tell you about the relationship between where an element is stored and how long it takes to find it? How does this compare to retrieving an element from a `Vector<Integer>` using `get` at any index?

Algorithmic Complexity

Introduction

None of the data structures or algorithms that we cover in this course exist in a vacuum. Historically, all of them have had some reason to exist for one reason or another. Practically, every algorithm also must be processed **through time** and with **real hardware**. What I mean by that is, if we run *selection sort* on some dataset we must send real digital signals through transistors which must be sent through more transistors and so on. All of this *takes time*, there is a temporal and spatial dimension to every single thing we do as computer programmers. As such, it behooves us to understand the implications of choosing one algorithm, or one data structure over another by analyzing how these abstractions perform both in *theory* and in *practice*. We will see that two algorithms solving the exact same problem can differ significatly in performance due to their different approaches.

Examples of Complexity

Before we get into some classical computer science abstractions, I would like to go through some "real world" examples that will help provide intuition on what algorithmic complexity is.

Constant Time Operations

Think about the last time you needed to flip to a particular chapter in a book. Likely the first thing you did was take a look at the *table of contents* and found the chapter name you were looking for. Next to the name was the page number associated with the start of the chapter. This tells you *immediately* where you need to open the book to get the information you needed. We would call that immediate jump to the chapter start a *constant time operation*. It is constant because there is no longer any need to *search through* pages in the book to find the chapter. The table of contents gives us the answer in a single step regardless of how long the book is.

Variable Time Operations

Let's say the book didn't come with a table of contents, most novels do not. You still want to find a particular chapter, but you don't really know where it is. The natural act is to begin scanning through the book front to back to find the chapter you're looking for. If you flip through the book one page at a time you are guaranteed to *eventually* find the page you are looking for, but you do not know *exactly when* that might happen. We would call this a *variable-time operation* as the time it takes to complete the task would vary

with where exactly the key chapter is and how many pages the book has in the first place. Looking page-by-page, it will take longer to find the last chapter of a book than it would to find the first chapter—similarly, it will likely take longer to find a chapter in a 900 page book than a 50 page book.

Big O Notation

Many years prior to the establishment of computer science, Paul Bachmann and Edmund Landau developed a mathematical formalism for the analysis of the asymptotic behavior of functions. Bachmann-Landau notation was constructed to describe how functions "grew" at both similar and different rates. Using their notation, the formula $f(n) = O(g(n))$ is equivalent to saying that "$f(n)$ grows no faster than $g(n)$ does" such that there exists some constant value M that $f(n) \leq Mg(n)$. If we take $f(n)$ to be the identity function $id(n) = n$ and $g(n)$ to be equivalent to $square(n) = n^2$ we could say that $id = O(square)$. There is some constant that can be applied to *square* which maintains that every value of *id* is either equal to or less than it (in this case M could be set to 1).

As computer science tends to bend towards the practical rather than the purely theoretical, we also tend to be a bit looser with the above mathematical formalism. This comes as a direct consequence of Donald Knuth's seminal work *The Art of Computer Programming* where he adopted Bachmann-Landau notation for his discussions on how various algorithms and data structures can be compared in terms of both time and space complexity. We often follow his lead of dropping constant values in complexity comparisons. For example, if an algorithm has been found to be described by the formula $f(n^2 + 2n)$ we would say that $f(n^2 + 2n) = O(n^2)$ even though that's not valid for the strict notation. This is because in computer science we are interested in how algorithms *scale* and the limiting scaling factor is the dominant term.

The best way to get a feel for algorithmic complexity is to see how different algorithms stack up against one another. First we'll look at two different sorting algorithms, *bubble sort* and *merge sort*, then we'll look at two different data structures, *arrays* and *lists*.

Comparing Sorting Algorithms (Bubble vs Merge Sort)

From the chapter on sorting algorithms, we're already familiar with *bubble sort*. *Bubble sort* has us "floating" larger values "up" to the end of an array. *Merge sort* is an entirely different beast. It is a recursive algorithm which works by progressively splitting an array in two and sorting each smaller array within each split. I've provided a Java implementation of both below:

Listing 1: Bubble Sort

```
1:  public class BubbleSort{
2:      public static void swap(int a, int b, int[] arr){
3:          int temp = arr[a];
4:          arr[a] = arr[b];
5:          arr[b] = temp;
6:      }
7:      public static void sort(int[] arr){
8:          boolean swapped = false;
9:          int j = arr.length;
10:         do{
11:             swapped = false;
12:             for(int i = 1; i < j; i++){
13:                 if(arr[i-1] > arr[i]){
14:                     swap(i-1, i, arr);
15:                     swapped = true;
16:                 }
17:             }
18:             j--;
19:         } while(swapped == true);
20:     }
21: }
```

Listing 2: Merge Sort

```
 1:   public class MergeSort{
 2:       public static void mergesortMerge(int[]b, int begin, int middle, int
end, int[] a){
 3:           int i = begin;
 4:           int j = middle;
 5:           for(int k = begin; k < end; k++){
 6:               if(i < middle && (j >= end || a[i] <= a[j])){
 7:                   b[k] = a[i];
 8:                   i += 1;
 9:               }else{
10:                   b[k] = a[j];
11:                   j += 1;
12:               }
13:           }
14:       }
15:       public static void mergesortSplit(int[] b, int begin, int end, int[]
a){
16:           if((end - begin) <= 1){
17:               return; // size == 1 -> sorted
18:           }
19:           int middle = (end + begin) / 2;
20:
21:           mergesortSplit(a, begin, middle, b); //left half
22:           mergesortSplit(a, middle, end, b); //right half
23:
24:           mergesortMerge(b, begin, middle, end, a);
25:       }
26:       public static void mergesort(int[]a, int[]b){
27:           for(int i = 0; i < a.length; i++){
28:               b[i] = a[i];
29:           }
30:           mergesortSplit(a, 0, a.length, b);
31:       }
32:       public static void sort(int[] arr){
33:           mergesort(arr, new int[arr.length]);
34:       }
35:   }
```

Note: You are not expected to memorize the merge sort algorithm.

I have also provided a program to test these implementations:

Listing 3: Benchmarking Sorting Algorithms

```
 1:   import java.io.*;
 2:   import java.util.Arrays;
 3:   public class BigO{
 4:       public static class InvalidSort extends RuntimeException{
 5:           InvalidSort(String msg){
 6:               super(msg);
 7:           }
 8:       };
 9:       public static int[] loadValues(String filename){
10:           int[] arr = null;
11:           try (BufferedReader buffer = new BufferedReader(new
FileReader(filename))) {
12:               int size = Integer.parseInt(buffer.readLine());
13:               arr = new int[size];
14:               String num;
15:               int i = 0;
16:               while ((num = buffer.readLine()) != null){
17:                   arr[i] = Integer.parseInt(num);
18:                   i += 1;
19:               }
20:           }catch(Exception e){
21:               System.err.println(e);
22:           }
23:           return arr;
24:       }
25:       public static void printArr(int[] arr){
26:           System.out.print("[");
27:           for(int e : arr){
28:               System.out.print(e + ",");
29:           }
30:           System.out.println("]");
31:       }
32:       public static void main(String[] args){
33:           int[] arr = loadValues(args[0]);
34:           String sortChoice = args[1];
35:           Runtime runtime = Runtime.getRuntime();
36:           System.gc();
37:           long beforeUsedMem=runtime.totalMemory()-runtime.freeMemory();
38:           long start = System.nanoTime();
39:           if(sortChoice.equals("bubble")){
40:               BubbleSort.sort(arr);
41:           }else if(sortChoice.equals("merge")){
42:               MergeSort.sort(arr);
43:           }else{
44:               throw new InvalidSort("No Valid Sort Found!");
45:           }
46:           long end = System.nanoTime();
47:           System.gc();
48:           long afterUsedMem=runtime.totalMemory()-runtime.freeMemory();
49:           double seconds = (end-start) / (1000000000.0);
50:           long actualMemUsed=afterUsedMem-beforeUsedMem;
51:           System.out.println(String.format("Execution time: %.2fs",
seconds));
52:           System.out.println(String.format("Memory Used: %d",
actualMemUsed));
53:       }
54:   }
```

Here is a python script that can generate the input used by this program:

Listing 4: Input generation script

```
import random
import sys
numInts = int(sys.argv[2])
with open(sys.argv[1], "w") as fp:
    fp.write(f"{numInts}\n");
    for i in range(numInts):
        fp.write(f"{random.randint(0, 1000)}\n");
```

We can then run `BigO` on both algorithms with *100* integers:

Listing 5: Running with 100 integers

```
josephraskind@stargazer:/tmp/bigO/bigO$ java BigO.java input/100.txt bubble
Execution time: 0.04s
Memory Used: -560
josephraskind@stargazer:/tmp/bigO/bigO$ java BigO.java input/100.txt merge
Execution time: 0.07s
Memory Used: -744
```

There's not a lot of difference between the two. Let's try *100,000* integers:

Listing 6: Running with 100,000 integers

```
josephraskind@stargazer:/tmp/bigO/bigO$ java BigO.java input/100000.txt bubble
Execution time: 16.99s
Memory Used: 1392
josephraskind@stargazer:/tmp/bigO/bigO$ java BigO.java input/100000.txt merge
Execution time: 0.07s
Memory Used: 1688
```

We see that *merge sort* has massive speedup when compared to *bubble sort*. This has to do with the fact that merge sort's recursive algorithm breaks the sorting problem down into smaller parts that are all looked at independently of each other prior to being merged. *Bubble sort* has no choice but to look at the *entire array of values* every loop! We say that *bubble sort* has $O(n^2)$ whereas *merge sort* has $O(nlogn)$.

Note: When speaking, we say that *bubble sort* has a "big O of n squared".

Something that is not as obvious, as the way of getting memory usage during the Java execution is a bit crude, is that the merge sort gains are not without consequences. *Merge sort* takes up *way more memory* than *bubble sort* does. Every algorithm has both time and space tradeoffs. Beyond that, it's clear that it's also much harder to write *merge sort* than it is to write *bubble sort*. What we gain in speed we lose in implementation complexity.

Comparing Data Structures (Arrays vs Linked Lists)

Now let's compare the complexity of two classic data structures: arrays and linked lists. If we take a look at insertion between arrays and linked lists

it's clear that the winner would have to be linked lists. Linked lists can add elements in constant time whereas arrays must be recreated and old elements must be copied over before another element can be added in. However, arrays have the advantage of constant time access that linked lists don't. It is impossible to search through a linked list any faster than looking at each element one at a time; however, it *is possible* to write a search algorithm for a linked list that can make use of constant time access (we'll soon learn about the binary search algorithm).

I leave it as an exercise for the reader to benchmark these realities on their own.

Exercises

1. The chapter introduces Big O notation as a way to describe how algorithms scale. For each of the following operations, identify the Big O complexity and justify your answer using the real-world analogies from the chapter:
    - Retrieving an element from a `Vector<T>` by index
    - Searching for an element in an unsorted `LinkedList`
    - Bubble sort on an array of n elements
    - Appending to a `LinkedList<T>` that has a `tail` pointer
    - Finding the minimum element in an unsorted array
2. The chapter shows that `f(n² + 2n) = O(n²)` because the dominant term is `n²`. Simplify each of the following to its Big O class and explain which term dominates and why:
    - `f(5n³ + 100n² + n)`
    - `f(n + log n)`
    - `f(2ⁿ + n¹⁰⁰)`
    - `f(n log n + n²)`
    - `f(1000)`
3. Compile and run `BigO.java` with both bubble sort and merge sort on a small input (*10* elements), a medium input (*5,000* elements), and a large input (*100,000* elements). Record the execution time for each combination. Does the execution time scale as you would expect from the Big O class of each algorithm? At what input size does the difference between the two algorithms become clearly visible?
4. The chapter motivates algorithmic complexity through the observation that all computation takes real time on real hardware. Consider two algorithms: one that runs in `O(n²)` and one that runs in `O(n log n)`. For small inputs the `O(n²)` algorithm might actually be faster due to lower constant overhead. Design an experiment using `BigO.java` that finds the crossover point—the input size at which merge sort becomes faster than bubble sort on your machine. What does this tell you about the relationship between theoretical complexity and practical performance?

Binary Trees

Introduction

If you went to kindergarten/grade school in the United States there's a good chance you had to complete a school project where you sketched out a family tree. The top of the tree would consist of distant, or not *so* distant, ancestors and it would eventually pass through your parents and end up at you. Each person's name in the tree would have two branches going up, representing parents[1], but could have many branches going down, representing children. If we prune the nodes that are not strictly ancestral by only keeping parents and no siblings, what you drew was a **binary tree** and it is one of the most used, fundamental data structures in computing.

What is a Binary Tree?

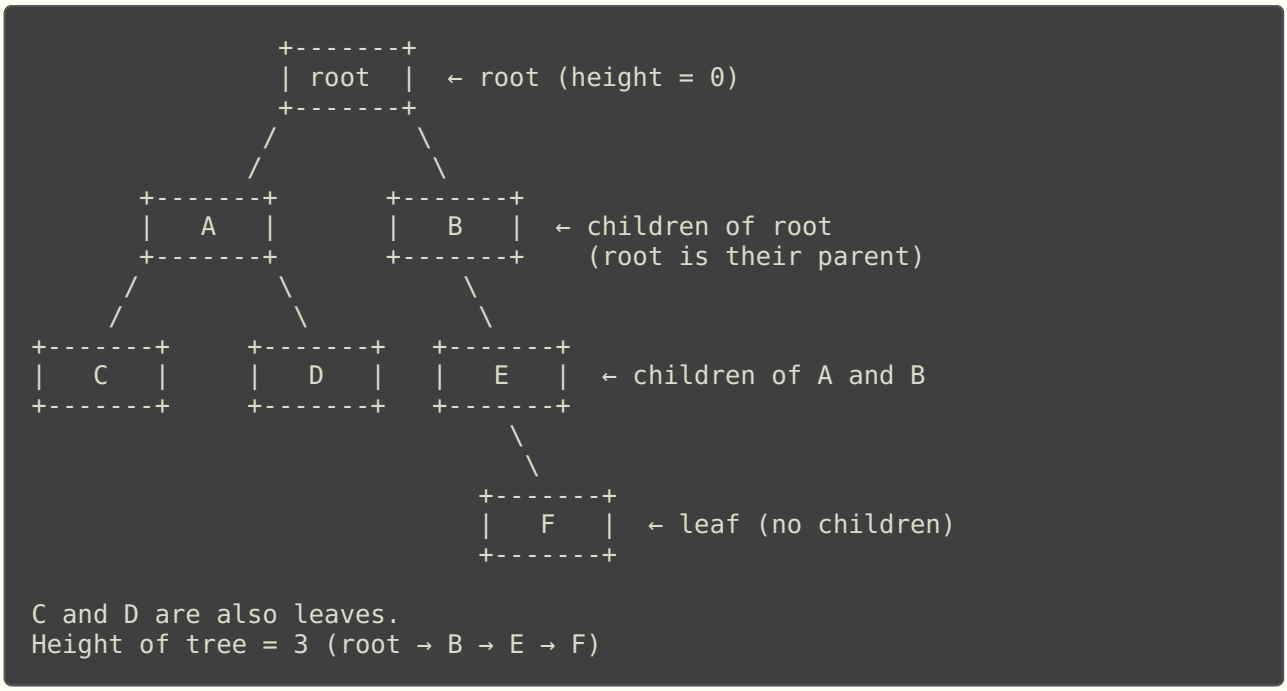
A binary tree is a data structure which, like linked lists, must satisfy a handful of logical constraints:

1. All binary trees must have a starting node called the root node
2. All nodes in the tree can have at most two children.
3. All nodes may have at most one parent.

Any tree-like structure which breaks these rules are by-definition not *binary* trees. For example, the Linux filesystem can easily be represented as a *tree*, but it is not a *binary* tree because a single directory (node) may have more than two files (children).

Binary Tree Terminology

The **root** node is the starting node in a tree ("root" for the roots, or anchoring point, of a tree). Nodes can connect to other nodes. Nodes that are connected to a given node that go up the tree are called **parents**. Nodes that are connected to a given node that go down the tree are called **children**. Nodes that have no children are called **leaves**. The **height** of a tree indicates how many jumps from the root need to be made to reach the farthest leaf. The following diagram visualizes this terminology:



We call a binary tree **full** if every node has either two children or none. We call a binary tree **complete** if every level is filled with nodes, except the last level where nodes do not have to be filled in completely, but must be packed to the left. Finally, we call a binary tree **perfect** if all interior nodes have two children and all leaves have the same depth or same level. We can visualize the different types of trees below:

Binary Tree Traversal

There are three standard ways of traversing a Binary Tree: **preorder**, **inorder**, and **postorder**. I'll outline the pseudocode for **preorder**:

Listing 1: preorder binary tree traversal

```
1:   preorder(node):
2:     print(node.data)     # Perform computation
3:     preorder(node.left)
4:     preorder(node.right)
```

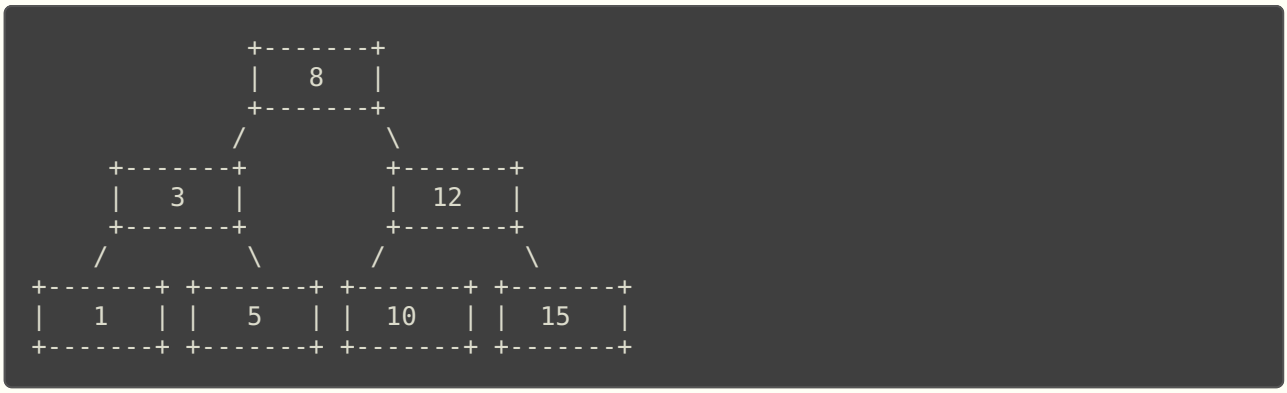
Here, we have a recursive function which looks at a node in a binary tree, performs a computation, and then moves to the left node, recursively, and then the right node. We can see it visually below:

```
                    +-------+
                    |   1   |  ← visit first (root)
                    +-------+
                   /         \
            +-------+       +-------+
            |   2   |       |   5   |  ← visit 2 before 5
            +-------+       +-------+
           /         \     /         \
        +-------+ +-------+ +-------+ +-------+
        |   3   | |   4   | |   6   | |   7   |
        +-------+ +-------+ +-------+ +-------+

Order visited: 1 → 2 → 3 → 4 → 5 → 6 → 7

Rule: visit the current node BEFORE visiting children
      always go left before right

Steps:
  visit 1 (root)
  go left  → visit 2
  go left  → visit 3 (leaf, go back up)
  go right → visit 4 (leaf, go back up)
  go right → visit 5
  go left  → visit 6 (leaf, go back up)
  go right → visit 7 (leaf)
```

I'll leave it as an exerise to the reader to try out inorder traversal (where the computation is between moving left and right) and postorder traversal (where the computation is after moving left and right).

Binary Search Tree

Binary trees in and of themselves are often not useful in the real world. They become more interesting when further constraints are added to them. One such binary tree variant is the **binary search tree**. Binary search trees are binary trees with one additional wrinkle: they maintain a strict order of nodes. For any given node, all values in its left subtree must be less than that node's value, and all values in its right subtree must be greater. Here is an example of a binary search tree:



It can be seen that for every node, all of the values to the right of it are greater than its value, and all of the values to the left are smaller.

Binary Search Algorithm

If you've ever tried looking through the index of a book to find a specific word, you've likely already employed the basic logic of *binary search*. Let's say you're in a C textbook and you're trying to find the word *function*. First, you open up the index to the A's and grab a hunk of pages and flip right to left. Next, you find yourself in the T's, which you know comes after F, so you grab a hunk of pages and flip left to right. Now, you're in the D's which you know comes before F, so you grab a hunk of pages and flip right to left again. Finally, we've made it to the F's and function should be smack dab in the middle of the page.

This more or less describes the algorithm for performing a binary search on a binary search tree. The following pseudocode implements it more formally:

```
1:  search(node, elem):
2:    if node.data = elem:
3:      return true
4:    else:
5:      if elem > node.data:
6:        return search(node.right, elem)
7:      else:
8:        return search(node.left, elem)
```

We can visualize the binary search below:


50
25
75
12
37
62
87
6
18
81
93
Searching for 81:
step 1: visit 50 candidates: whole tree (15 nodes)
81 > 50 → go right, left subtree eliminated
step 2: visit 75 candidates: 7 nodes
81 > 75 → go right, left subtree (62) eliminated
step 3: visit 87 candidates: 3 nodes
81 < 87 → go left, right subtree (93) eliminated

```
step 4: visit 81 ✓        candidates: 1 node
        found!
```

What you'll notice is that every single time a decision is made to move left or right half of the binary tree is ignored further down the search. The order of the binary search is what allows for this to occur. Intuitively, the smaller the portion of the tree we have to search, the faster the search will be. It turns out that binary search in a "well-balanced" binary search tree should take only about $\log_2(n)$ steps where n is the number of nodes in the tree. If we take n to be 15, corresponding to the number of nodes in the above tree, we'll find that $\log_2(15) \approx 4$ which is equivalent to the number of steps taken to get the search results!

Not all binary search trees are guaranteed to be "well-balanced", i.e. an even distribution of nodes across all levels.

Binary Search Tree API

We can define a BST API as follows:

- `init`
  - Initializes a BST
- `insert`
  - Inserts an element into the BST
- `search`
  - Searches for an element in the BST
- `remove`
  - Removes the desired element in the BST
- `is_empty`
  - Indicates whether the BST is empty or not
- `free`
  - Frees the BST

Inserting Elements into a Binary Search Tree

Binary search trees only work if their constraints are maintained. We can guarantee that elements are slotted into their proper place by utilizing the same traversal algorithm as the binary search (we'll assume here that elements must be unique):

```
1:  insert(node, elem):
2:    if node.data = elem:
3:      ERROR
4:    else:
5:      if elem > node.data:
6:          if node.right = null:
7:            node.right ← elem
8:          else:
9:            insert(node.right, elem)
10:     else:
11:       if node.left = null:
12:         node.left ← elem
13:       else:
14:         insert(node.left, elem)
```

We can visualize this algorithm by adding `70` to the previous BST:

Listing 2: Inserting an element

Deleting Elements from a Binary Search Tree

Removing elements from a BST presents a greater challenge. For example, consider the tree in Listing 2. If we want to delete `81`, then the process is trivial. But what if we want to delete `50`? How could that be accomplished? If we naively replace it with `75` then something must be done about `62` and `87`. This is why BST's will have a well-defined method for finding a successor node. A common solution is to find the smallest node that is greater than the node being removed. For example, if `50` is marked for deletion, then `62` will be chosen:

```
successor(n):
  if n.right not = null:
    return minimum(n.right)
  s ← n.parent
  while s not = null and n = s.right:
    n ← s
    s ← s.parent
  return s
```

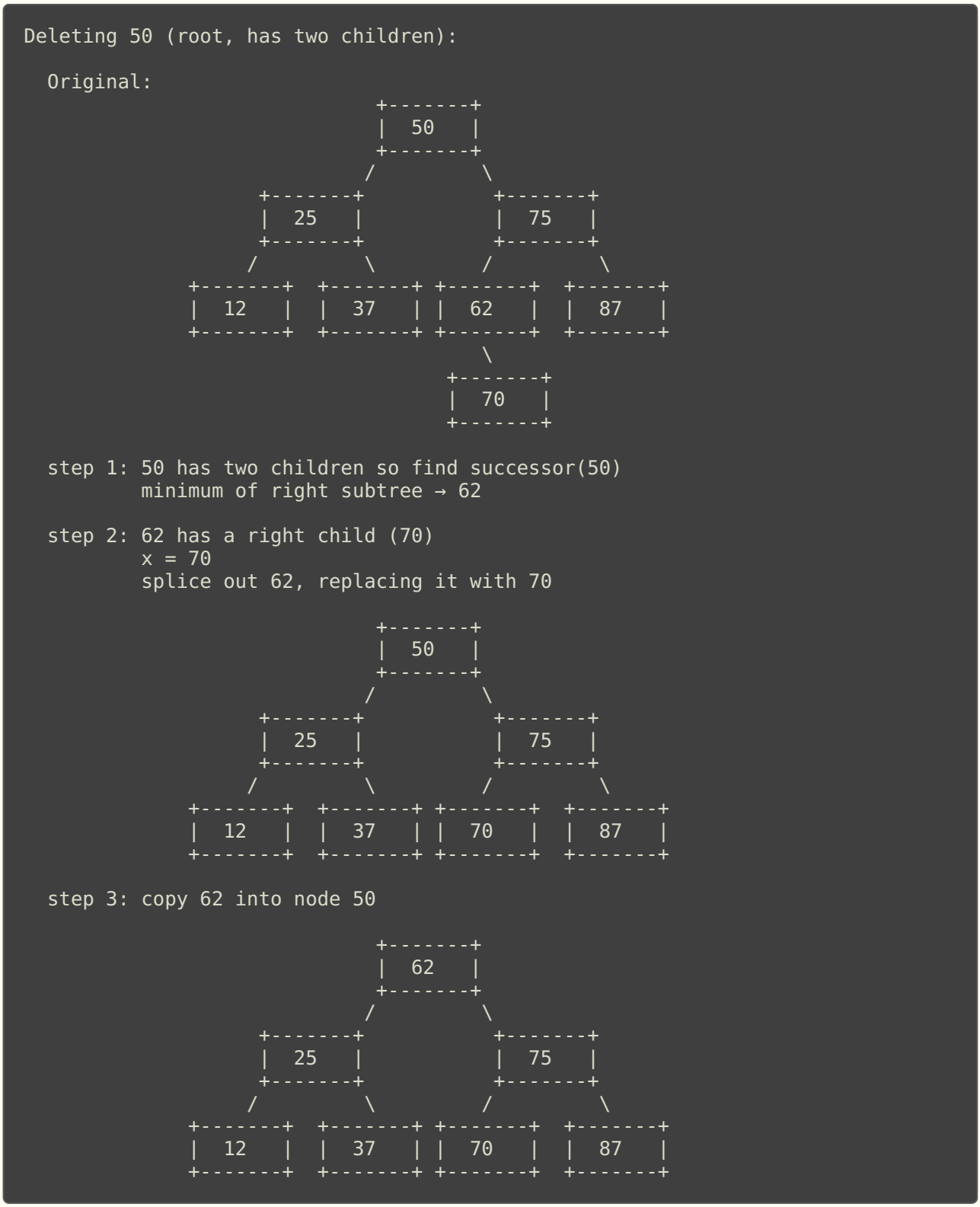
where `minimum` spans the leftmost children. This will always grab either the smallest element greater than `n` from its right subtree, or the lowest ancestor for which `n` lies in the left subtree—both of which are the next largest value in the tree relative to `n`.

We can then define deletion as follows:

Listing 3: deletion for BSTs

```
delete(n):
  if n.left = null or n.right = null:  # Find a node without two children
    y ← n                              # and set it equal to y
  else:
    y ← successor(n)
  if y.left not = null:                # x is set to the left or right
child
    x ← y.left                         # or nill if neither exist
  else:
    x ← y.right
  if x not = null:                     # We then splice out y
    x.parent ← y.parent
  if y.parent = null:
    tree.root ← x
  else if y = y.parent.left:
    y.parent.left ← x
  else:
    y.parent.right ← x
  if y not = n:                        # If the succesor of n was
spliced out
    n.data = y.data                    # swap the data and return y for
removal
  return y

```

We can see the algorithm in action with the above BST:



We can see that after the deletion the assumptions about the binary search tree are maintained.

Binary Seach Tree: Pros & Cons

Pros:

+ Fastest possible search if balanced
+ Dynamic size
+ Naturally recursive structure

Cons:

- Loses search speed if unbalanced
- Memory overhead (storing pointers *and* data)
- No constant time operations

Exercises

1. Implement a BST node as an inner class in Java. It should contain an integer data field, a `left` reference, a `right` reference, and a `parent` reference. Then implement a `BST` class with a `root` field initialized to `null` and a constructor that sets up an empty tree.
2. Implement `insert` from the pseudocode in the chapter in Java. Test it by inserting the values `(50, 25, 75, 12, 37, 62, 87)` in that order and verifying the resulting tree matches the diagram shown in the chapter.
3. Implement `search` from the pseudocode in the chapter in Java. Test it on the tree built in exercise 2 by searching for both values that exist and values that do not. What is the maximum number of nodes visited when searching a tree of height 3?
4. The chapter leaves inorder and postorder traversal as an exercise. Write the pseudocode for both and then implement all three traversals (preorder, inorder, postorder) in Java. Run inorder traversal on the BST from exercise 2 and observe the output. What do you notice about the order of the values printed?
5. Write a recursive method `height` in Java that computes the height of a binary tree. Recall from the chapter that height is defined as the number of jumps from the root needed to reach the farthest leaf. Test it on the BST from exercise 2 and verify the result matches what you would expect from inspecting the tree diagram. Then test it on a tree with only a root node — what should the height be in that case?
6. The chapter notes that a BST "loses search speed if unbalanced." Starting from an empty BST, insert the values `(1, 2, 3, 4, 5, 6, 7)` in that order. Draw the resulting tree. What does it look like? Use your `height` method from exercise 5 to measure its height and compare it to a balanced BST containing the same values. What does this tell you about the importance of insertion order?
7. Implement `delete` from Listing 3 in Java. Handle all three cases: deleting a leaf, deleting a node with one child, and deleting a node with two children. Test each case on the tree from exercise 2 and verify the BST property is maintained after each deletion.

8. Implement `successor` from the pseudocode in the chapter in Java. Test it by finding the successor of every node in the tree from exercise 2 and verify each result by inspection of the tree diagram.
9. Java's garbage collector frees unreachable objects automatically so no explicit `free` is needed. Instead write a `clear` method that sets `root` to `null`, making the entire tree unreachable and eligible for garbage collection. Explain why setting `root` to `null` is sufficient to free the entire tree in Java when it would not be in C. Which traversal order would be required if you were freeing the tree manually in C and why?
10. A perfect binary tree of height *h* contains 2^(h+1) - 1 nodes. Verify this formula for heights 1, 2, and 3 by drawing the trees and counting nodes. Then write a method `isPerfect` that returns `true` if a binary tree is perfect and `false` otherwise, using your `height` method from exercise 5 to help.

Footnotes:

1 This, of course, implies a purely geneological/biological family tree. Those with step parents might break that assumption. Similarly, descendents of the Hapsburgs may also quickly run into serious trouble.

(Appendix) import and packages

Introduction

When programming in Java you'll often want more functionality than what's immediately available in a basic program. The way we obtain that additional functionality is through the `import` statement.

Using import

If we want to use a standard library object, like `ArrayList`, we first need to *import* it. We can do like so:

Listing 1: Importing ArrayList

```
1:  import java.util.ArrayList;
2:  public class UsingPackages{
3:      public static void main(String[] args){
4:          ArrayList<Integer> lst = new ArrayList<>();
5:      }
6:  }
```

Let's see what happens if we don't include the `import`:

Listing 2: Not importing ArrayList

```
1:  public class UsingPackages{
2:      public static void main(String[] args){
3:          ArrayList<Integer> lst = new ArrayList<>();
4:      }
5:  }
```

If we compile the above we get:

Listing 3: No import fails compilation

```
josephraskind@stargazer:/tmp/packages$ javac UsingPackages.java
UsingPackages.java:3: error: cannot find symbol
        ArrayList<Integer> lst = new ArrayList<>();
        ^
  symbol:   class ArrayList
  location: class UsingPackages
UsingPackages.java:3: error: cannot find symbol
        ArrayList<Integer> lst = new ArrayList<>();
                                     ^
  symbol:   class ArrayList
  location: class UsingPackages
2 errors
```

Why? `ArrayList` is defined as part of the `java.util` package. The only package that is loaded in by default during the Java runtime is `java.lang`. This is why we can use `String` by default---`String` is also defined in `java.lang`. Anything that isn't a part of `java.lang` *must be imported*.

Defining packages

We can define our own packages through the use of the `package` keyword. We can see it below:

Listing 4: Using package

```
1:  package mypackage;
2:  public class MyClass{
3:      public MyClass(){}
4:  }
```

This only works because `MyClass` has been defined in a directory named `mypackage`:

Listing 5: Looking at mypackage

```
josephraskind@stargazer:/tmp/packages$ ls
mypackage  UsingPackages.java
josephraskind@stargazer:/tmp/packages$ ls mypackage/
MyClass.java
```

And we can import it easily:

Listing 6: Importing the user-defined package

```
1:  import mypackage.MyClass;
2:  public class UsingPackages{
3:      public static void main(String[] args){
4:          MyClass mc = new MyClass();
5:      }
6:  }
```

We can also use the wildcard syntax:

Listing 7: Importing the user-defined package using wildcards

```
1:  import mypackage.*;
2:  public class UsingPackages{
3:      public static void main(String[] args){
4:          MyClass mc = new MyClass();
5:      }
6:  }
```

Note: Wildcards do not import subpackages but only packages at that given level.

There are a few rules associated with defining packages:

- Packages must be unique
    - There cannot be two packages with the same path
- Packages may have the same class name in two different paths

(Appendix) Command-line Arguments

Introduction

Oftentimes you'll want to write programs that take input directly from the user. One way to accomplish that is to take in arguments from the *command line*—i.e. directly from the terminal itself. It's actually rather simple to do so.

Taking Command-line Arguments

The *Java Programming Language Specification* does not precisely lay out how command line arguments are to be had. It does, however, suggest how they tend to be processed through the use of the `main` method's `String[]` parameter (§ 12.1).:

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

For example, if I want to write a program called `Hello.java` which will say "Hello" to every name given in the command line, I could write the following:

```
1:   public class Hello{
2:       public static void main(String[] args){ // args is a conventional
name, but is not required
3:           for(String name : args){
4:               System.out.printf("Hello, %s!\n", name);
5:           }
6:       }
7:   }
```

Then I can compile and run it with three arguments:

```
josephraskind@stargazer:/tmp/CLI$ javac Hello.java; java Hello Marx Engels
Lenin
Hello, Marx!
Hello, Engels!
Hello, Lenin!
```

Converting Command-line Arguments

Command line arguments are always Strings. This means you *must* convert them if you want to take in numbers to perform calculations. For example, if I want to write a program that calculates standard deviation:

```
 1:  public class Statistics{
 2:      public static void main(String[] args){
 3:          int size = args.length;
 4:          double[] elements = new double[size];
 5:          double sum = 0;
 6:
 7:          for(int i = 0; i < size; i++){
 8:              elements[i] = Double.parseDouble(args[i]);
 9:              sum += elements[i];
10:          }
11:
12:          double mean = sum / size;
13:          sum = 0;
14:
15:          for(int i = 0; i < size; i++){
16:              sum += Math.pow(elements[i] - mean, 2);
17:          }
18:
19:          double std = Math.sqrt(sum / size);
20:          System.out.printf("mean: %.2f%n std: %.2f%n", mean, std);
21:      }
22:  }
```

```
josephraskind@stargazer:/tmp/CLI$ javac Statistics.java; java Statistics 1 2 3
mean: 2.00
 std: 0.82
```

Common functions for String conversions are listed below for reference:

- `Integer.parseInt(str)`
  - String to int
- `Long.parseLong(str)`
  - String to long
- `Double.parseDouble(str)`
  - String to double

(Appendix) Debugging with jdb

Introduction

A lot of beginner programmers do what's called "print debugging"—using print statements to debug their programs. Although it tends to be fine for tiny programs, it is often not enough to get at the heart of catastrophic errors in complicated programs. This is where the use of programs specifically made for debugging shines.

jdb

`jdb` is the OpenJDK debugger which was made to pair with `javac` in order to debug Java programs. It's important to compile java files with `-g` if you want to use the debugger.

Below is an example Java program which has a simple bug in it:

```
 1:   public class AverageCalculator {
 2:       public static double average(int[] arr){
 3:           int sum = 0;
 4:           for(int i = 0; i < arr.length; i++){
 5:               sum += arr[i];
 6:           }
 7:           return sum / arr.length;  // bug: integer division truncates
result
 8:       }
 9:
10:       public static void main(String[] args){
11:           int[] grades = {85, 92, 78, 91, 88};
12:           double avg = average(grades);
13:           System.out.printf("Average: %.2f%n", avg);  // prints 86.00,
should be 86.80
14:       }
15:   }
```

Compiling and running it I get:

```
josephraskind@stargazer:/tmp/jdb$ java AverageCalculator.java
Average: 86.00
```

Which is clearly not the `86.80` I would have expected. This tells me I should fire up the debugger. I can do so with a call to `jdb ./AverageCalculator`:

```
josephraskind@stargazer:/tmp/jdb$ jdb AverageCalculator
Initializing jdb ...
>
```

I've now entered `jdb`. There are a handful of core `jdb` instructions that are often always helpful:

- `run`
  - Execute the program
- `stop`
  - `stop at m` breaks at the beginning of the `m` method
- `next`
  - Advances one line at a time (bypasses methods)
- `step`
  - Advances one line at a time (enters methods)
- `locals`
  - Prints out local variables

`jdb` let's you freeze a program and look at it step by step:

```
> stop at AverageCalculator.average
> Deferring breakpoint AverageCalculator.average.
It will be set after the class is loaded.
> run
> run AverageCalculator
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: Set deferred breakpoint AverageCalculator.average

Breakpoint hit: "thread=main", AverageCalculator.average(), line=3 bci=0
3            int sum = 0;

main[1]
```

Here I set a breakpoint at the entrance to the method `average` and run the program. Once program execution hits `average` it is paused. I can then step one line (after the method arguments are initialized) and look at the local variables and arguments:

```
main[1] step
main[1] >
Step completed: "thread=main", AverageCalculator.average(), line=4 bci=2
4            for(int i = 0; i < arr.length; i++){

main[1] locals
main[1] Method arguments:
arr = instance of int[5] (id=451)
Local Variables:
sum = 0
```

We see that `jdb` is now paused at the `for` statement. I use `locals` to check the value of all local variables (here `sum` is initialized to 0).

I can then keep using `next` to go from one line to the next. Eventually, we'll be able to get to the average calculation:

```
main[1] next
main[1] >
Step completed: "thread=main", AverageCalculator.average(), line=7 bci=22
7            return sum / arr.length;  // bug: integer division truncates
result
main[1] print sum / arr.length
main[1]  sum / arr.length = 86
```

I can use the `print` function to evaluate expressions using the available symbols at that point in the runtime. We can see that the integer division is causing this error. Now it is a simple process of exiting `jdb` with `quit` and fixing the code!

Note: Using `jdb` can get extremely complicated. For this class, most major issues can be solved with setting a breakpoint and stepping through a function line by line while examining local variables/global data structures. It is a great resource and should not be ignored when you are in trouble.