Introduction to Computer Science

PTI CS 103 Lecture Notes

The Prison Teaching Initiative

This version: July 15, 2019

Authors

TK.

Acknowledgements

TK.

Contents

1

Introduction

1.1 Overview of course

Knowing just a little bit of computer science can get you started right away in actual applications. One of the goals of this course is to learn about the fascinating subject of computer science. Another is to develop algorithmic thinking skills that will help with day-to-day critical problem-solving skills. But perhaps the most important goal of the course is to develop coding skills, which will not only open up new job opportunities but also make you more effective in most areas of business.

In the first semester, we will spend the first two classes of each week on computer science theory and special topics. The final day of each week will be a lab day, where we actually start practicing coding skills.

In the second semester, we will start focusing more on practical coding, with a single day a week for theory and 2 lab periods per week for coding.

Broadly, we will cover the following topics:

- How modern computers work
  - Hardware
  - Software
  - Computer networks and information systems
- Algorithms for quickly solving complex problems
  - Searching
  - Sorting
- Data structures
  - Arrays
  - ArrayLists
- Applications of Computer science
  - Basic coding in Java
  - How to use productivity software

1.2 Brief history of computer science

Timeline (credit: https://www.worldsciencefestival.com/infographics/a_history_of_computer_science/):

- Invention of the abacus (2700-2300 BC, Sumerians)
- Design of first modern-style computer (Charles Babbage, 1837)

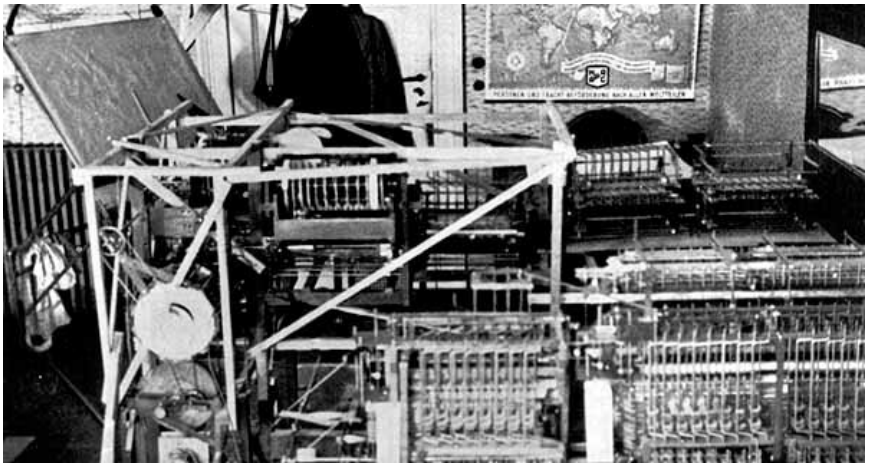


Figure 1.1: Construction of Konrad Zuse's Z1, the first modern computer, in his parents' apartment. Credit: https://history-computer.com/ModernComputer/Relays/Zuse.html

- Design of first computer algorithm (Ada Lovelace, 1843)
- Invention of first electronic digital computer (Konrad Zuse, 1941)
- Invention of the transistor (Bell labs, 1947)
- Invention of the first computer network (early Internet) (DARPA, 1968)
- Invention of the World Wide Web (Sir Tim Berners-Lee, 1990)

1.3 Components of a computer

A computer is an electronic device used to process data. Its basic role is to convert data into information that is useful to people.

There are 4 primary components of a computer:

- Hardware
- Software
- Data
- User

1.3.1 Hardware

Computer hardware consists of physical, electronic devices. These are the parts you actually can see and touch. Some examples include

- Central processing unit (CPU)
- Monitor
- CD drive
- Keyboard
- Computer data storage
- Graphic card
- Sound card
- Speakers
- Motherboard

We will discuss these components in more detail in lecture 3.

Figure 1.2: Examples of hardware components of a personal computer. Credit: https://www5.cob.ilstu.edu/dsmath1/tag/computer-hardware/

1.3.2 Software

Software (otherwise known as "programs" or "applications") are organized sets of instructions for controlling the computer.

There are two main classes of software:

- Applications software: programs allowing the human to interact directly with the computer
- Systems software: programs the computer uses to control itself

Some more familiar applications software include

- Microsoft Word: allows the user to edit text files
- Internet Explorer: connects the user to the world wide web
- iTunes: organizes and plays music files

While applications software allows the user to interact with the computer, systems software keeps the computer running. The operating system (OS) is the most common example of systems software, and it schedules tasks and manages storage of data.

We will dive deeper into the details of both applications and systems software in lecture 4.

1.3.3 Data

Data is fundamentally information of any kind. One key benefit of computers is their ability to reliably store massive quantities of data for a long time. Another is the speed with which they can do calculations on data once they receive instructions from a human user.

While humans can understand data with a wide variety of perceptions (taste, smell, hearing, touch, sight), computers read and write everything internally as "bits", or 0s and 1s.

Computers have software and hardware which allow them to convert their internal 0s and 1s into text, numerals, and images displayed on the monitor; and sounds which can be played through the speaker.

Similarly, humans have hardware and software used for converting human signals into computer-readable signals: a microphone converts sound, a camera converts pictures, and a text editor converts character symbols.

1.3.4 Users

Of course, there would be no data and no meaningful calculations without the human user. Computers are ultimately tools for making humans more powerful.

As we will see in the next section, however, different types of computers have different roles for the user.

1.4 Types of computers

1.4.1 Supercomputers

These are the most powerful computers out there. The are used for problems that take along time to calculate. They are rare because of their size and expense, and therefore primarily used by big organizations like universities, corporations, and the government.

The user of a supercomputer typically gives the computer a list of instructions, and allows the supercomputer to run on its own over the course of hours or days to complete its task.

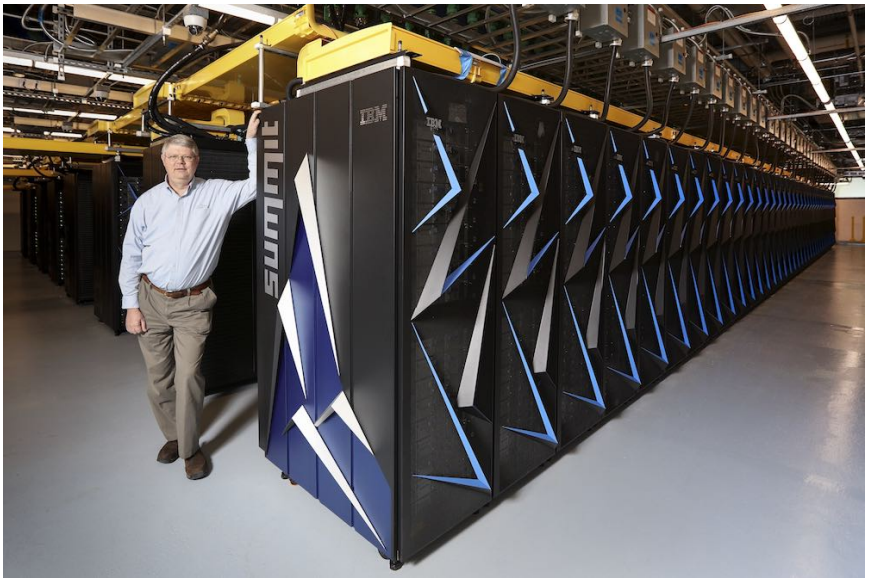


Figure 1.3: Summit, a world-class supercomputing cluster at Oak Ridge National Laboratory in Tennessee. Credit: https://insidehpc.com/2018/11/new-top500-list-lead-doe-supercomputers/

1.4.2 Mainframe computers

Although not as powerful as supercomputers, mainframe computers can handle more data and run much faster than a typical personal computer. Often, they are given instructions only periodically by computer programmers, and then run on their own for months at a time to store and process incoming data. For example, census number-crunching, consumer statistics, and transactions processing all use mainframe computers

1.4.3 Personal computers

These are the familiar computers we use to interact with applications every day. Full-size desktop computers and laptop computers are examples

1.4.4 Embedded computers

In the modern "digital" age, nearly all devices we use have computers embedded within them. From cars to washing machines to watches to heating systems, most everyday appliances have a computer within them that allows them to function.

1.4.5 Mobile computers

In the past 2 decades, mobile devices have exploded onto the scene, and smartphones have essentially become as capable as standalone personal computers for many tasks.

1.5 Why computers are useful

Computers help us in most tasks in the modern age. We can use them, for example, to

- write a letter
- do our taxes
- play video games
- watch videos
- surf the internet

- keep in touch with friends
- date
- order food
- control robots and self-driving cars

Example 1.5.1: What are some other tasks a computer can accomplish?

This is why the job market for computer scientists continues to expand, and why computer skills are more and more necessary even in non-computational jobs.

According to a Stackoverflow survey from 2018 (https://insights.stackoverflow.com/survey/2018/), 9% of professional coders on the online developer community have only been coding for 0-2 years. This demonstrates two things:

1. The job market for people with coding skills is continually expanding
2. It doesn't take much to become a coder

Some examples of careers in computer science include

- IT management / consulting
- Game developer
- Web developer
- UI/UX designer
- Data analyst
- Database manager

References

Computer Science: An Interdisciplinary Approach, Robert Sedgewick and Kevin Wayne.

University of Wisconsin-Madison CS 202 Lectures, Andrea Arpaci-Dusseau. (http://pages.cs.wisc.edu/ dussea F11/)

2

Hardware

Today's lecture will focus on computer systems, a complex group of devices working together to perform a common task (or tasks). A user will interact with a computer through a variety of input and output devices (e.g. keyboards, mice, speakers, microphone, and monitors). A user's input will be processed, some computations will be performed, and then the resulting output will be displayed to the user. When most of us thinking about computers, we often think of a desktop or laptop computer, that come equipped with a keyboard, mouse, and monitor as seen in Figure 2.1; however, many things we interact with daily are computerized, including cell phones, cars, traffic lights, smart watches, televisions, and manufacturing lines. Today each of these items have sensors to perceive the real world, use an embedded computing device to understand the sensory input, and use a combination of display and mechanical devices to interact with the real world.

Example 2.0.1: For intersections across busy roadways, some traffic lights are computerized to optimize road traffic. These lights will stay green along the busier of the two roads, and use cameras or pressure sensors to detect the presence of cars along the less busy of the two roads, thus switching to allowing the cars on the less busy road to cross when it arrives. Overall, providing a less congested intersection by relying on an embedded computer.

Today we will introduce three fundamental parts of computer systems: input and output devices, memory, and the central processing unit (CPU). These components work together to perform the basic building blocks of input processing, storage, control, and output. Understanding how the three parts work together will allow us to create powerful information processing tools. We will introduce each of these parts in turn. In figure 2.2, we see how these parts come together to form a computer system (similar to the ones you'll use to program in this course).

Figure 2.1: A variety of computer systems: desktops, laptops, tablet, and smart phone.

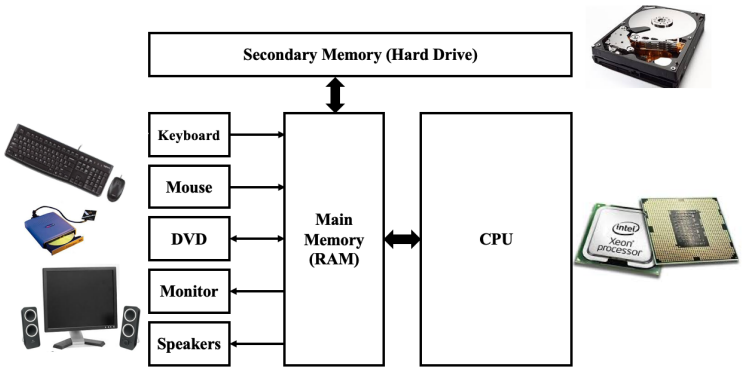


Figure 2.2: Interconnected parts of a computer system (keyboard, mouse, monitor, DVD player / burner, speaker, hard drive, CPU).

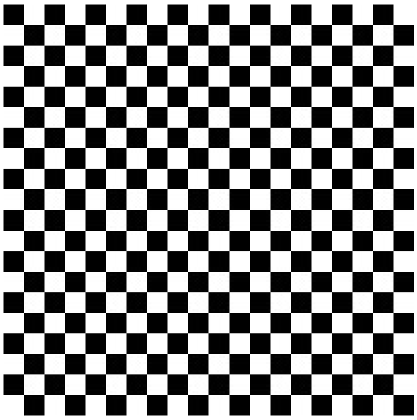
2.1 Input and Output Devices

We will begin by discussing input and output devices. These devices allow the computer to interact with users and the world directly. Without these devices, a computer system would be very boring, always performing the same computation each time it's used. Even if it did compute a different value we would be unable to examine the value. A computer needs to be able to accept input and allow output. The first computers would occupy a large room in an office building and connect to a terminal (a keyboard and a text screen) in another room for users to interact with. Thanks to Moore's Law, computers many orders of magnituted more powerful can fit in the palm of your hand. Likewise, the variety of input and output devices has multipled. We still have the keyboard and monitor, we've added the mouse for interacting with graphical displays. Today's phones are more computer than phone, coming equipped with: speakers, microphones, touch screens, cameras, fingerprint scanners, radio transmitters, and much more. Computers even come embedded in other devices like cars, traffic lights, X-ray machines, and thermostats to both control and monitor the devices. As shown in Figure 2.2, these devices connect to the rest of the computer through the computer's memory. This kind of input and output is called memory mapped I/O (input and output). Creaters of input (or output) devices are assigned a section of the computer's memory to write (read resp.) data. The computer will then read (write resp.) data to those locations to communicate with the given device. The creaters of these devices, agree upon a known format to read and write data.

Example 2.1.1: You can think of this communication between devices and computer similar to leaving messages for a friend in a locker. Only you and your friend have access to this locker, which only holds space for one message. The format you agree upon is which language you'll use to speak (e.g. English) and any special keywords or phrases. You might agree with your friend, that if either of you write a message saying "The morning is upon us" that the other will wait until "The night has come" before leaving any new messages.

The format that devices and computers communicate in are generally very simple and structured to permit fast and easily understandable communication for computers and devices.

Example 2.1.2: A monitor is a graphical display for computers. Let's consider a monitor connected to a computer that only displays in black and white images that are 20 x 20 pixels large. The monitor and keyboard agree upon using the following



```
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
```

Figure 2.3: An example checkered image and its encoding — newlines and spaces added for readability.

format to communicate. The format is black and white images that are 20 x 20 pixels large. Each pixel's value is represented at 0 for black and 1 for white. Then an image is represented as a 400 = (20 x 20) long sequence of pixel values. The sequence is ordered left to right, top to bottom. Now that both the monitor and computer agree upon the communication format, the computer can write images to the section of memory dedicated to the monitor and the monitor will read the image and display the image on it's screen. Figure 2.3 displays an example image, a 20 x 20 chekerboard with its encoding.

Note: while this is a simplified example, this is similar to how modern graphical displays communicate with computers.

2.2 Memory

Another fundamental part of a computer, is the memory. By memory, we mean the ability to store and recall data. This is very similar to physical storage of items. Figure 2.4 shows three storage locations — a storage closet, a garage, and a warehouse. Each of the three locations have tradeoff between convenience of location and storage capacity. The closet can contain a few things and is the same room you need it. The garage can fit even more things and is only a walk outside (or through) your home, and the warehouse can fit practically anything you would want to store but you have to drive to the warehouse to pick-up or store your items. Similarly, a computer's memory makes the same trade-offs.

There are two major types of memory, Main Memory (RAM), and Secondary Memory (e.g. hard disks, solid-state drves, tape drives, etc.) Main memory is volatile, meaning that the contents of the memory is not preserved when a computer is turned off and back on. On the other hand Secondary Memory, is meant to be persistent (the opposite of volatile). Main Memory can be thought of as the "scratch paper" the computer uses for computations. Computers will also use Main Memory as a conduit for communicating between the CPU and all other parts of the computer. Staying with the analogy from Figure 2.4 main memory is closer to a garage (where you can lose items when you turn off the lights) — there is enough room to fit most items you use regularly and is close enough to not worry about the time it takes to get to the garage.

In most modern computers, programs are treated as data. That is the individual instructions that combine to form a program are stored in memory just as data is. It is the job of the computer to properly understand if a segment of memory is data or a program. The computer is able to fetch data from Secondary Memory to Main Memory or persist data in Main Memory to Secondary Memory when needed; however, this process of transferring data between Secondary and Main Memory can cost a lot of time relative to keeping data in Main Memory only.

Figure 2.4: Storage closet, garage, and warehouse trading off between capacity and locality.

2.3 Central Processing Unit

The final part of a computer we will introduce today is the central processing unit (CPU) — processor, main processor, etc. The CPU is the physical circuitry of a computer that performs instructions. The CPU has several key components: the control logic, arithmetic and logic unit (ALU), registers, program counter (PC), and clock. These components work together to fetch, decode, and execute all instructions — the building blocks of all programs. Instructions vary between different brands of CPUs, but, in general, they will include arithmetic, control, read (from memory), and write (to memory) functionalities. Example 2.3.1 shows several instructions that together would perform `x = x + y`, given `x` is stored in memory location 16 and `y` at memory location 20. These instructions are quite low level, and harder for humans to read than the programs we will write in this course. However, the programs we write will be translated into these instructions to be easily understood and executed by the CPU.

Example 2.3.1:

```
load R1 16    -- Load value at memory location 16 into register 1
load R2 20    -- load value at memory location 20 into register 2
add  R1 R2 R1 -- add the value in register 1 to the value in register 2
                 and store in register 1
store R1 16   -- store the value in register 1 to memory location 16
```

A key component of a CPU is its clock. The clock allows the CPU to progress in time, triggering the time to progress from time `t` to time `t + 1`. A single time segment is referred to a clock cycle. You might have heard about a computer's CPU speed (e.g. my computer runs at 2.4 GHz = 2.4 billion clock cycles a second). This is determined by how fast the clock transitions from one time to the next. This clock drives the progress of the CPU. The time an instruction takes is measured in instruction cycles. The rate of the CPU's clock is determined by the slowest operation of the CPU (e.g. fetch, decode, execute stage).

In addition to clocks, the CPU contains a group of memory locations called registers. A single register is capable of holding a single word of information (the smallest unit of data in a computer). The key benefit of registers is the ability for the CPU to immediately read and write the contents of the CPU. The value of registers can be updated on each clock trigger (i.e. on the change from time `t` to `t + 1`). Most modern CPU's will have between 16 and 64 registers that programs may use. For comparison, accessing Main memory can take 10's or even 100's of instruction cycles to access while registers are immediately available to the CPU.

The control unit and program counter (PC) will fetch, decode, and output the controls for the execution of each instruction. The program counter holds the location of the next instruction to be executed. The next instruction is then fetched to the CPU. The CPU's control unit then decodes the fetched instruction and outputs control signals (commands) to main memory and the ALU. The CPU may read data from memory (e.g. store) and then the ALU will then execute the action specified by the control unit (e.g. add, subtract, multiply, compare to 0, etc.), and then possibly write the output to memory.

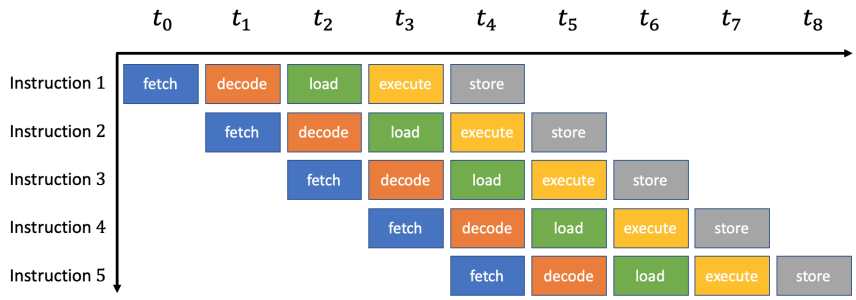


Figure 2.5: Instruction Pipelining

In most computers, the CPU and its constiuent parts are responsible for all computing needs of the computer. In some select systems, there will be additional hardware to perform specialized operations (e.g. graphics processing units for processing / producing images). It is the CPU's responsbility to control the computer and coordinate with devices to execute programs. As such, the CPU has seen a quick evolution to increase it's processing power. Electrical engineers, originally focused on making the CPU smaller and smaller and thus quicker, following Moore's Law : every two years the size of a CPU shrinks in half. Additionally, CPU's were designed with a pipelining architecture (i.e. multiple instructions are executed in quick succession). This is done by noting that each stage of the five stages — fetch, load, decode, execute, and store — can be performed independently. Thus, while one instruction is being executed, the next instruction can be decoded. Figure 2.5 shows how pipelining is peformed by executing the different parts of the pipeline in parallel for five consecutive instructions.

Due toe the decline of Moore's Law in recent years, many CPU designers focus on increasing processing power by improving parallelism (i.e. being able to execute multiple instructions at a time). This allows instructions that do not depend on each other to be executed at a time. These CPU are refered to as multi-core, as they have multiple cores that each have thier own set of registers, control unit, ALU, and PC but share main memory and a single clock. In this class we, will not teach how to effectively harness parallel programming, but note that this is an important progress in how hardware has evolved.

2.4 Conclusion

In this chapter, we covered the three fundamental parts of a computer system: input and output devices, Main and Secondary Memory, and the CPU. We discussed their roles, relationships, and basic capabilities. I hope that this will help you better understand how hardware works at a high level to better improve how to write programs that will eventually run on these computer system. In the next chapter we will begin discussing the concept of Software and how its similarities and differences to hardware and where the boundary between the two lies.

2.5 Learning Objectives

After covering this chapter, you should be able to answer the following questions:

1. What three parts comprise a computer system?
2. What are examples of common input and output devices?
3. Name an uncommon example of a computer system and explain how it may work.
4. How does memory mapped input and output work?

5. Name four kinds of memory devices and explain the difference between Main and Secondary Memory.
6. What parts comprise the central processing unit (CPU)?
7. Describe three possible methods to increase the computation power of a CPU.

3

Programming Languages and Linux commands

Today's lecture will focus on programming languages and how to give a computer instructions through a command line. Programming languages allow programmers to write instructions that can be interpreted by a computer to perform operations. Computers can not understand programming languages directly, so they have to be translated to a format that the computer can understand. In Java that transformation is referred to as compilation. When Java code is compiled, a program called a compiler takes the Java code as input and as output produces code that's easier for a computer to execute directly.

For instance, suppose that you have a text file with a large number of itemized expenses and you'd like to calculate the total of the expenses. Writing instructions for this task that the computer's hardware could understand directly could take more time than summing the entire list manually. Using a programming language to specify the instructions allows you to efficiently tell the computer to perform this task.

3.1 Most popular programming languages

What programming languages are most used depends on the application that the programming language is best used for. Java is one of the most popular programming languages. It's used extensively by large companies; it powers many of the services of companies like Oracle and Amazon. It's also one of the primarily languages used to develop applications for the Android mobile operating system.

JavaScript is another very popular language. Despite its similarity to Java in name, it is a very different language from Java. It's the primary language used for writing code to interact with browsers in websites. It powers features of websites like generating dynamic content, sending requests to a server, making sure that the information entered in forms is correct, and many more things. It's also increasingly used to write applications. It's a popular choice among newer and smaller companies.

The number and diversity of programming languages can be intimidating, but learning new programming languages languages becomes easier once you have a solid understanding of fundamental concepts. In this course we use one programming language, but the ideas and concepts you learn can help you learn any programming language you need to. We recommend focusing on one programming language initially to learn fundamental concepts rather than spending time trying to learn multiple languages. We've decided to focus on Java because it's easier to learn, widely used, and illustrates many important concepts in programming. 40% of developers who responded to a large survey

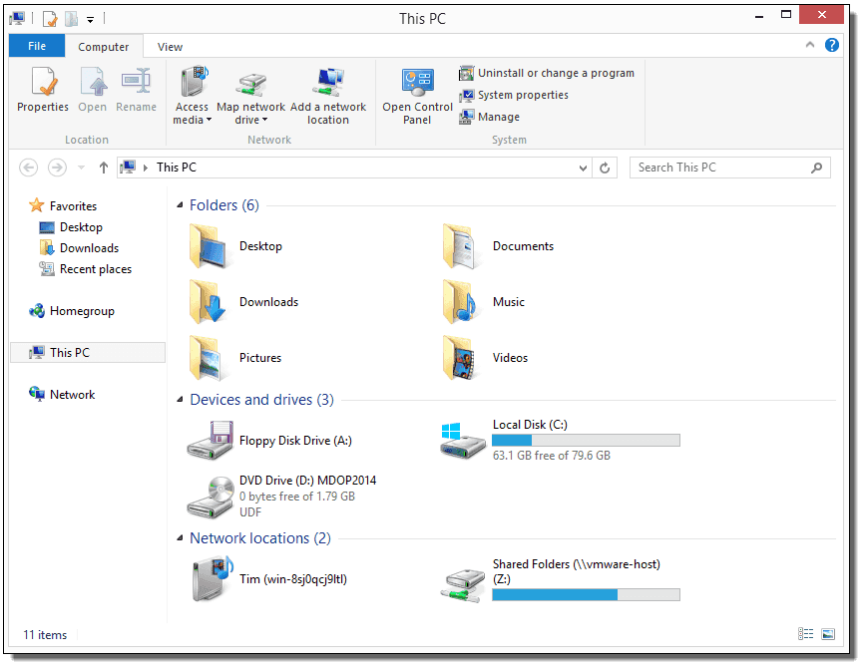


Figure 3.1: Windows file explorer

of developers stated that they know Java!

3.2 Overview of a GUI

Using a graphical user interface (GUI) or command line are the two primary ways to interact with your computer and complete task like edit files, create folders, and run programs. A graphical user interface like a file explorer (see figure 7.1) provides an easy way for users to complete tasks on their computer. A graphical user interface is the primary way most users of a computer interact with the computer. It's popular because users can use their mouse and intuitive commands to complete basic tasks.

The graphical user interface abstracts the details of the commands being issued to the computer away from the user of the program. This abstraction provides simplicity, but it also limits the flexibility of what a user can do with the computer. Furthermore, it means that users can only interact with a system that has a graphical user interface installed. This doesn't include servers, computers without standard operating system, or computers that a user is connected to remotely.

The command line is a program you can access through the GUI of a linux machine. It's also installed on servers without a GUI and can be accessed directly through monitors (this was the only way to interact with computers before the invention of the GUI).

A command line allows the user to issue a broader range of commands and interact with computers without a graphical user interface. It gives users the ability to interface directly with the operating system and issue commands that it can understand. This increases the range of the commands that the user can issue, giving them more flexibility to perform complex and custom tasks.

We will discuss the command line used in the Linux operating system because it is one of the most command and intuitive command lines.

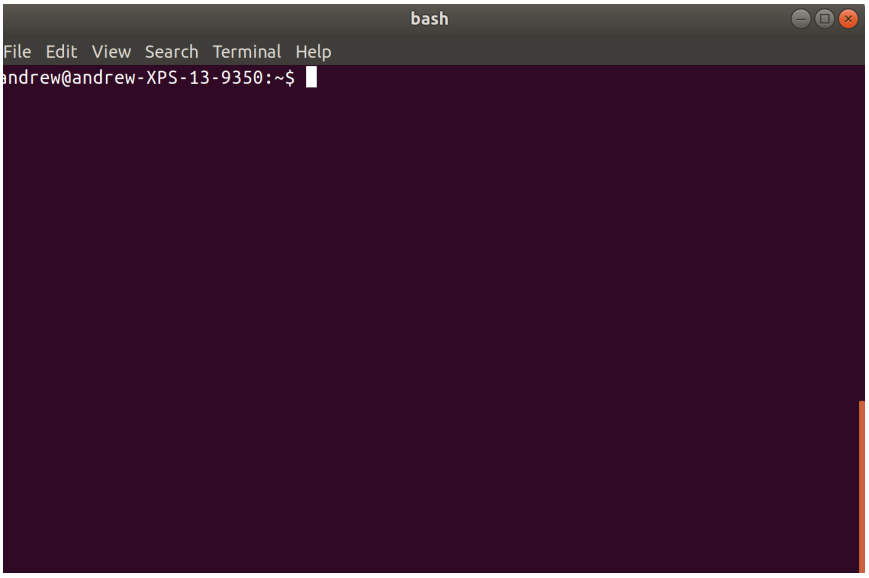


Figure 3.2: Linux command line

Command	Description
ls	List all the files in the current directory
touch	Create a new file
cd [directory]	Changes the current directory to the directory given
mkdir	Create a new directory within the current directory
man [command]	Display the manual for a certain command
cat [file]	Display the contents of file in the terminal
mv [file] [location]	Moves files to a new location. By default it moves file to a different name within the same directory.
cp [file] [location]	Creates a new file with identical contents to the file specified in a new location. If a path for the file isn't given, it will copy the file to the same directory.

A folder in Linux is referred to as a directory. When you open a command line, you're in a certain directory within the file system. All commands that you issue that relate to interacting with the file system like creating files, editing files, or renaming files will be issued in the context of this directory. This is like navigating to a particular folder within the GUI file explorer and completing all your operations relative to that folder.

Figure 7.2 shows the command line interface for a Linux computer. The prompt shown is where the user can type commands. By default it prepends the user's name, the name of the computer, and a dollar sign. The white rectangle represents the location of my cursor prompting the user to enter commands.

To make a new directory, the user can use the 'mkdir' command. This is like making a new folder within an existing folder in the window's file explorer.

To navigate to that directory the user can use the 'cd' command, short for change directory. Figure 7.3 displays an example of creating a new directory and changing to use that directory. In this example the sign prepending the command represents the user's home directory. The home directory is the outer most folder for that user's account on the server or machine.

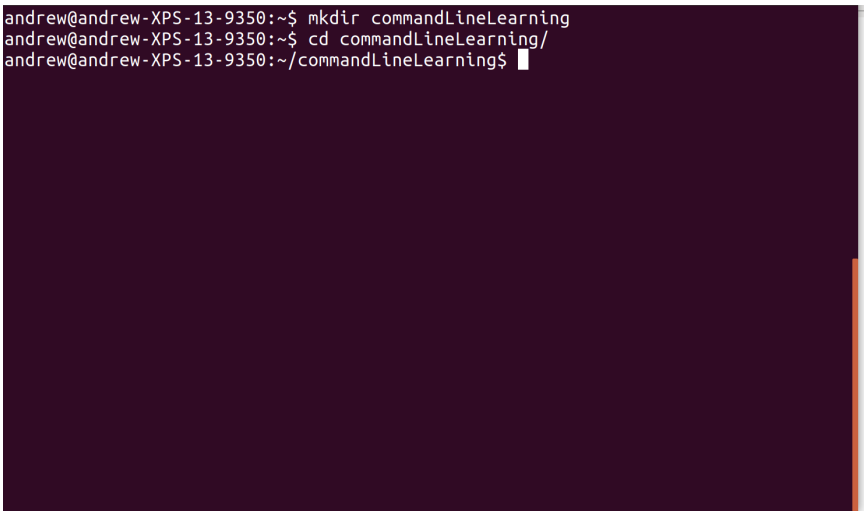


Figure 3.3: Creating a directory

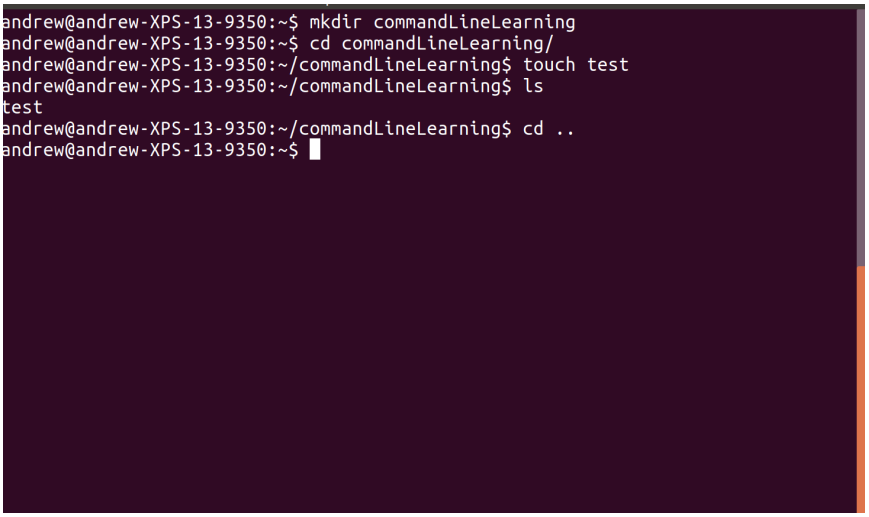


Figure 3.4: Listing files and navigating

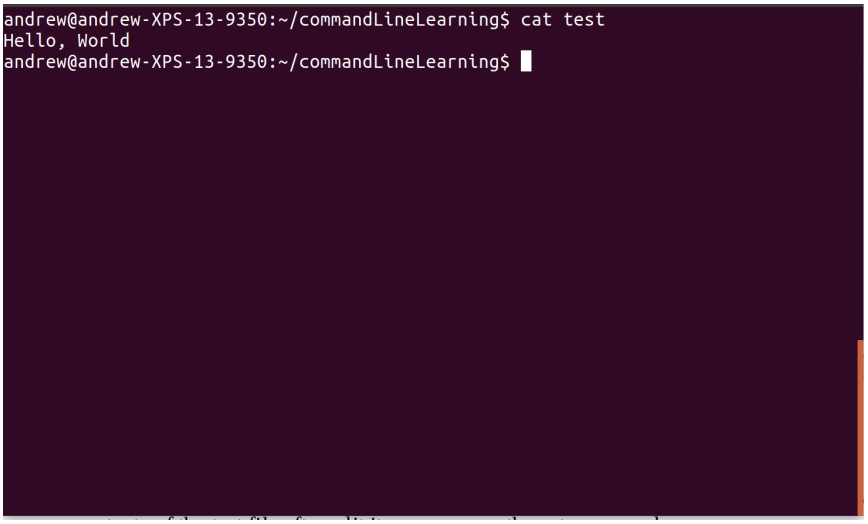


Figure 3.5: Cat command example

In figure 7.3 I create a commandLineLearning directory and change my current directory to that directory. If I want to create a file in that directory, I can use the 'touch' command to create a new file. If I want to see the files in the directory, I can use the 'ls' command to list all of the files in the current directory. If I want to move back to the directory I was in before, I can use the 'cd' command with the parameters '..' to move to the directory my current directory is contained within. In Linux, '..' refers to the directory that contains your current directory, or the parent directory. These commands are summarized in figure 7.4.

Try these commands out. You can edit the contents of test using any file editing software. To view the contents of the test file after edit it, you can use the cat command.

References

Stack Overflow Developer Survey 2019. (n.d.).

4

Networks

In this chapter, we'll learn about how the Internet and different kinds of networks work. The Internet is a giant network composed of many different smaller networks. Each of these networks have to link together various devices to allow devices within a network to communicate with each other and allow devices within one network to communicate with another network. There are complex protocols that dictate how different kinds of devices should communicate with each other, but you don't have to understand the details of these protocols to understand how devices can communicate with each other.

In addition to being interesting and important to understanding technology, understanding networks is very practical. Many companies look for Internet Technology or IT specialist that understand how networks work. Taking a test like the Network+ certification or A+ certification allows access to these jobs. These tests are knowledge based, so you don't need practical experience to do well on them. They qualify you for entry level positions that pay around $50,000 a year. If you're interested in preparing for these exams, there are many books that can help you prepare. For instance, "CompTIA A+ Certification All-in-One Exam Guide, Ninth Edition (Exams 220-901 & 220-902)."

In this section we'll start by examining how a device could communicate with another device in a small network and build up to understanding how a device can communicate to a device across the world.

4.1 Local Area Networks

A local area network (LAN) is a relatively small network that is often used in homes, schools, individual office buildings, or another small building. In a local area network each computer connected to the network will be assigned an Internet Protocol Address (IP address) for use within the network. If computer A wants to send a message to computer B within the same address, computer A would refer to computer B by its IP address. IP addresses are 32 bit numbers that are written with dots between each byte. For instance, the address 192.168.0.1 usually refers to the machine that enters the address, a location known as "local host."

If computer B wants to receive messages from computer A, it could want to receive messages for multiple reasons. To allow this, messages sent to another computer have to include what "port" they want to use. A port is a number used to identify a particular reason that another machine could want to receive a message. Many ports, especially ports with smaller numbers, have conventional services that they correspond to. For instance, port 80 corresponds to a web server, port 443 corresponds to a web server using encryption, and port 5432 corresponds to a particular kind of database.

The simplest way for multiple computers to communicate is to be connected together by a cable. If two machines want to talk with each other and the computers are connected to each other by a cable, a machine would broadcast the message to all other machines along with the address the machine was intended for. The machine the message was intended for would pick up the message and all others would hopefully discard it, although they could still see the contents if they wanted to.

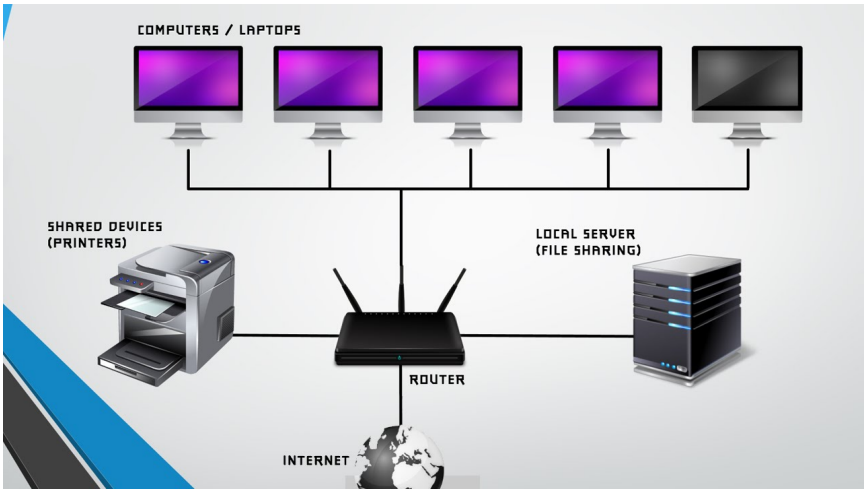


Figure 4.1: Local Area Network Example

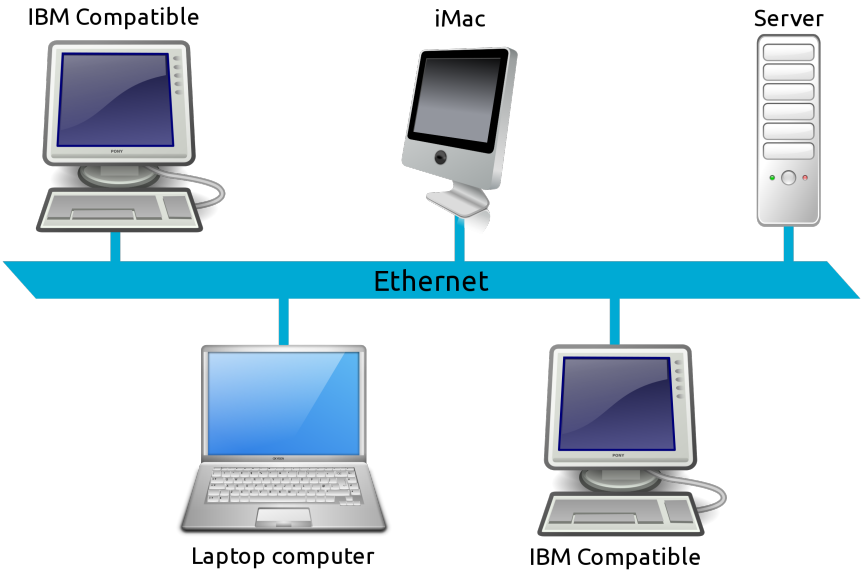


Figure 4.2: Example of communicating over Ethernet

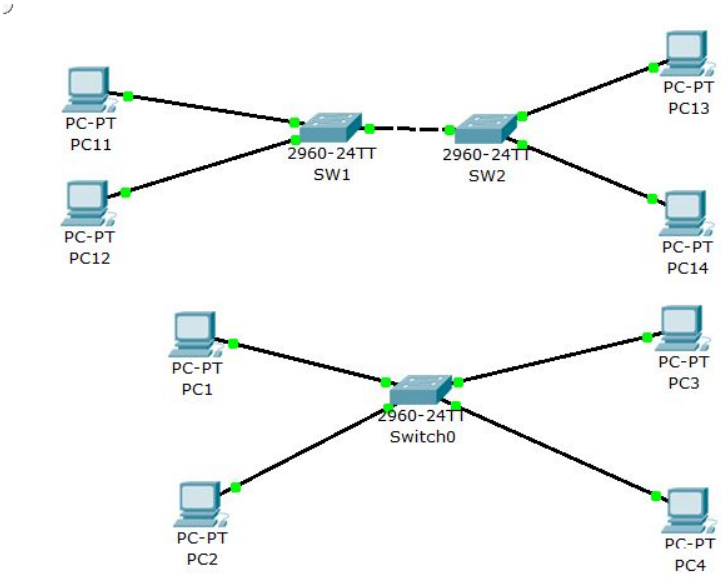


Figure 4.3: Example of communicating over Ethernet

A switch is a more sophisticated way to route information between computers on the same network than an Ethernet cable. A switch knows how to allow machines on the same network to communicate. In larger local area networks many different switches could be used to carry messages between different sets of computers. In the case of wireless devices, the wireless's devices signals are collected by an access point and then translated to signals that can be sent across a physical wire to switches.

If devices within the local area network want to communicate with devices outside the local area network, they'll need to go through a router. A router is the machine that assigns IP addresses to other machines in the same local area network. While these machines can use the IP addresses to communicate with other machine within the same local area network, they can't use the IP address to communicate directly with machines outside the local area network. The router has an IP address that it uses to communicate with other devices outside of the LAN. If a device within the LAN wants to communicate with a device outside of the LAN, the router makes the request for the device inside the LAN and changes the IP address to the IP address of the router. It maintains state about the device that sent the request to the router so that when the target of the request replies, the router can forward the reply to the right device in the network.

Consider the example given in figure 7.4. If the machine with the IP address 192.168.100.3 wants to send a request to a machine with the IP address 209.131.36.158 using port 80, it will forward that request to the router. The router will then send change the requesting IP address to its IP address of 145.12.131.7 and when it receives the reply from 209.131.36.158, it will forward that reply to 192.168.100.3.

One of the primary reasons this process occurs is because there are only 2^{32} possible IP addresses in the most used version of the IP protocol (the number of numbers that can be created using 32 bits). This may seem like a lot, but 2^{32} is about 4.3 billion, and there are many more devices that can be connected to the Internet than this! This scheme allows many of the device addresses to be reused.

4.2 Wide area networks

Wide area networks refer to large networks that usually combine multiple local area network. The definition of what is considered a wide area network varies, but it usually involves connecting local area networks that are miles apart. In this section we'll look at an outline of how large networks are connected to give an intuition for how devices in the modern world communicate.

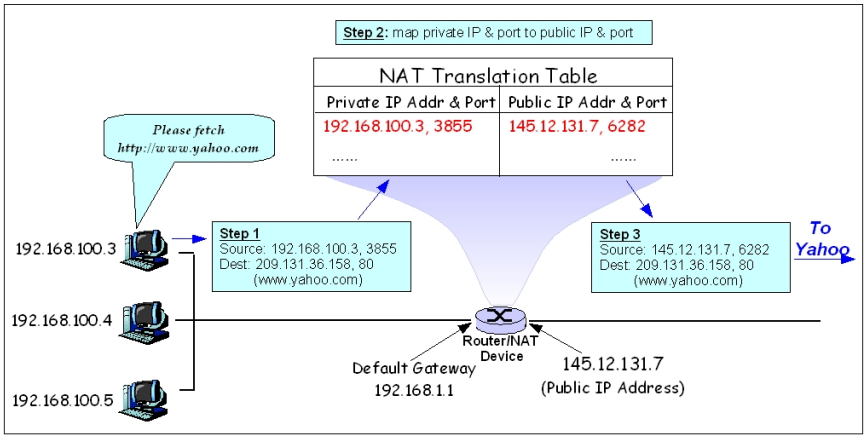


Figure 4.4: Network routing

Almost all information between computers will travel through wires for a significant portion of the time, with the major exception being information that is beamed up to a satellite and then beamed down to another device. For two local area networks to communicate over large distances, they usually enlist the service of an Internet Service Provider (ISP). An ISP like Time Warner Cable or Comcast is responsible for ensuring that there is a path between two separate local area networks that want to contribute. When connecting to an ISP, the LAN would have a wire that connects the LAN to the network of wires the ISP uses to route information between different LANs.

The ISP would make agreements with different autonomous systems (ASs) to be able to route traffic through the autonomous system so that it doesn't have to own the cable used to route data between different places. These ASs advertise what other ASs they can communicate with using the Boarder Gateway Protocol. The protocol can allow ASs to figure out if there's a path to another AS through an AS they're directly connected to.

To send information across the world, there are cables running through the ocean operated by Internet Service Providers. There are cables acros mountains, deserts, and other challenging terrains to enable the flow of information.

References

Muhammadlilg. (2014, October 21). Retrieved June 16, 2019.

Otome, S. O. (n.d.). What are the reasons for not putting multiple subnets on the same VLAN?

5

Control Structures

So far, we have seen programs in which each statement is executed in order, one by one. Today we will learn about *conditionals*, which allow us to execute statements depending on certain conditions. This is our first exposure to the idea of *control flow*, which refers to the order (or sequence) in which statements of a program are executed.

In this chapter, we will first learn about `if` statements, which allow us to write programs based on conditions. Then we will learn about `else` statements. We will combine these `if` and `else` statements into more complex nested structures, and then finally learn about `else if` statements. Lastly, this chapter ends with a list a common mistakes to avoid when writing conditionals.

5.1 The `if` statement

Definition 5.1.1. An *if statement* is a *conditional statement* that consists of the reserved keyword `if`, followed by a boolean expression enclosed in parentheses, followed by a statement typically enclosed in curly braces. If the boolean expression (or *condition*) evaluates to true, the statement is executed. Otherwise, it is skipped.

An `if` statement allows us to write programs that decide whether to execute a particular statement, based on a boolean expression which we call the *condition*. Below is an example of a simple `if` statement:

```
1 if (count > 20) {
2     System.out.println("Count exceeded");
3 }
```

The condition in this example is `count > 20`. It is a boolean expression that evaluates to either true or false. That is, `count` is either greater than `20` or not. If it is, "Count exceeded" is printed. Otherwise, the `println` statement is skipped.

Example 5.1.2: What does the following piece of code print if `x` is 101? What about if `x` is 200? What about if `x` is 8?

```
1 if (x > 100) {
2     System.out.println("Big number!");
3 }
4 System.out.println("Hi there");
5 if (x % 2 == 0) { // checks if a number is even*
6     System.out.println("Even number!");
7 }
```

* Recall that the modulo operator (%) computes the remainder of some number. By

checking if the remainder of some number when divided by 2 is 0, we are checking if that number is even.

Answer: If `x` is 101, the first boolean expression will evaluate to true, so the program will print "Big number!" Then the program will print "Hi there" regardless of any condition. Finally, since 101 is not even, the program will not print "Even number!"

Using the same reasoning to trace the execution with `x` as 200, the following statements will be printed: "Big number! Hi there! Even number!"

When `x` is 8, the output is "Hi there! Even number!"

Example 5.1.3: How would you fill in the boolean expression below to take the absolute value of an integer `x`? (Hint: to take an absolute value of a negative number, you must negate it).

```
if (/* Insert boolean expression here */) {
    x = -x;
}
```

Answer: Insert the boolean expression $x < 0$.

5.2 The `else` statement

Sometimes we want to do one thing if a condition is true and another thing if that condition is false. We can add an `else` clause to an `if` statement to handle this kind of situation. Below is an example of a simple `if-else` statement:

```
if (price > 20) {
    System.out.println("Too expensive");
} else {
    System.out.println("Affordable");
}
```

This example prints either "Too expensive" or "Affordable", depending on whether `price` is greater than 20. Only one or the other is ever executed, never both. This is because boolean conditions only evaluate to either true or false.

Note that it is not possible to have an `else` statement without having a corresponding `if` statement. Any `else` statement must be attached to a preceding `if` statement.

Example 5.2.1: What does the following piece of code do?

```
if (balance <= 0) {
    System.out.println("Unable to withdraw");
} else {
    System.out.println("Withdraw successful");
}
```

Answer: It prints "Unable to withdraw" if balance is 0 or lower. Otherwise, it prints "Withdraw successful".

Example 5.2.2: Rewrite the code below using an `else` statement.

```
if (rating >= 4) {
```

```
2       System.out.println("Would recommend");
3   } if (rating < 4) {
4       System.out.println("Would not recommend")
5   }
```

Answer: Since the second boolean expression is the negation of the first boolean expression, we can replace the second `if` statement with an `else`.

```
1   if (rating >= 4) {
2       System.out.println("Would recommend");
3   } else {
4       System.out.println("Would not recommend")
5   }
```

Example 5.2.3: Does the following code compile?

```
1   else {
2       System.out.println("Hello");
3   }
```

Answer: No, since an `else` statement must have a corresponding `if` statement.

Example 5.2.4: Does the following code compile?

```
1   if (duration > 60) {
2       System.out.println("Sorry, that's too long");
3   } else {
4       System.out.println("I can fit that in my schedule");
5   } else {
6       System.out.println("Let me know when you are free")
7   }
```

Answer: No, since an `if` statement may only have at most 1 corresponding `else` statement.

5.3 Nested conditionals

It is possible to combine `if` and `else` statements in more interesting ways, by nesting them. Consider the following example:

```
1   if (x < y) {
2       System.out.println("x is less than y");
3   } else {
4       if (x > y) {
5           System.out.println("x is greater than y");
6       } else {
7           System.out.println("x is equal to y");
8       }
9   }
```

This example correctly prints out the relationship between two integer variables `x` and `y` using nested conditionals. No matter what the values of `x` and `y` are, only a single statement will be printed.

Example 5.3.1: Fill in each blank below with `if` or `else` to describe a person's height.

```
/*__________*/ (height < 60) {
    System.out.println("Relatively short")
} /*__________*/ {
    /*__________*/ (height > 72) {
        System.out.println("Relatively tall");
    } /*__________*/ {
        System.out.println("Pretty average");
    }
}
```

Answer: The four blanks should contain `if`, `else`, `if`, and `else`, in that order.

Example 5.3.2: Given integer variables `a` and `b` and the following piece of code, what value gets stored in variable `c`?

```
int c;
if (a > b) {
    c = a;
} else {
    if (b > a) {
        c = b;
    } else {
        c = 0;
    }
}
```

Answer: The larger of the two variables, `a` and `b`, gets stored in `c`. If `a` and `b` are equal, 0 gets stored in `c`.

Example 5.3.3: Imagine that the following program represents the person's reaction to different types of food. Given a pizza (`green` = false, `bitter` = false, `warm` = true, `cheesy` = true), how will this person respond? What about when given a kiwi (`green` = true, `bitter` = false, `warm` = false, `cheesy` = false)? What about a strawberry (`green` = false, `bitter` = false, `warm` = false, `cheesy` = false)?

```
if (green || bitter) {
    System.out.println("No thank you");
} else {
    if (warm && cheesy) {
        System.out.println("Yum, yes please!");
    } else {
        System.out.println("Thank you, I'll take some");
    }
}
```

Answer: The person reponds "Yum, yes please!" to pizza, "No thank you" to kiwi, and "Thank you, I'll take some" to strawberry.

5.4 The `else if` statement

Nested conditionals are useful in many applications, but they can start feeling clunky as the nesting becomes deep. For example, consider the following piece of code:

```
if (grade > 90) {
    System.out.println("You earned an A");
} else {
    if (grade > 80) {
        System.out.println("You earned a B");
    } else {
        if (grade > 70) {
            System.out.println("You earned a C");
        } else {
            if (grade > 60) {
                System.out.println("You earned a D");
            } else {
                System.out.println("You earned an F");
            }
        }
    }
}
```

To avoid having such deeply nested conditionals, we introduce the `else if` clause. Using `else if` statements, we can rewrite the above program as:

```
if (grade > 90) {
    System.out.println("You earned an A");
} else if (grade > 80) {
    System.out.println("You earned a B");
} else if (grade > 70) {
    System.out.println("You earned a C");
} else if (grade > 60) {
    System.out.println("You earned a D");
} else {
    System.out.println("You earned an F");
}
```

You can imagine an `else if` statement as shorthand for an `if` nested inside an `else`. For example, the following snippets of code behave identically (for any boolean expressions A and B):

```
// Version 1 (using nested conditionals)
if (A) {
    System.out.println("A");
} else {
    if (B) {
        System.out.println("B");
    } else {
        System.out.println("C");
    }
}

// Version 2 (equivalent, using else if)
if (A) {
    System.out.println("A");
} else if (B) {
    System.out.println("B");
} else {
    System.out.println("C");
}
```

It is very common to see a conditional block of code that consists of 1 `if` statement, 0 or more `else if` statements, and 1 `else` statements. In these cases, recall that only one of these statements will ever execute.

Example 5.4.1: Imagine `A` and `B` are boolean expressions. If both of them evaluate to true, what does the following code print out?

```
if (A) {
    System.out.println("apple");
} else if (B) {
    System.out.println("banana");
} else {
    System.out.println("carrot");
}
System.out.println("dragonfruit");
```

Answer: The code will print "apple" and then "dragonfruit". Notice that "banana" does not get printed even though `B` is true since the entire `if`, `else if`, `else` chain will only ever print one of "apple", "banana", and "carrot".

Example 5.4.2: Imagine `X`, `Y`, and `Z` are boolean expressions. If `X` evaluates to true, but `Y` and `Z` evaluate to false, what does the following code print out?

```
if (X && Y) {
    System.out.println("xylophone");
} else if (!Z) {
    System.out.println("zoo");
}
```

Answer: This code will print "zoo". Notice that the boolean expressions used with `if` statements can be complex and contain boolean operators such as `&&` and `||` and `!`, as long as the expression evaluates to a true or false.

5.5 Curly braces

So far, we have been using curly braces to enclose each of our `if`, `else if`, and `else` statements. These curly braces can be omitted if there is only a single statement. For example, in the snippet of code below, the first set of curly braces can be omitted, while the second cannot.

```
if (guess == answer) {
    System.out.println("You guessed correctly!");
} else {
    System.out.println("Sorry, you guessed incorrectly.");
    System.out.println("The answer was " + answer);
}
```

If all the curly braces were left out (as in the code below), the program would first either print "You guessed correctly!" or "Sorry, you guessed incorrectly.", but then also print "The answer was ..." in all cases, regardless of the condition.

```
if (guess == answer)
    System.out.println("You guessed correctly!");
else
    System.out.println("Sorry, you guessed incorrectly.");
    System.out.println("The answer was " + answer);
    // ^^^ Careful! This is not part of the else clause!
```

Here it is important to note that whitespace and indentation are ignored by Java. Indentation has no effect on the behavior of a program. Proper indentation is extremely important for human readability. When used incorrectly, however, misleading indentation can result in unexpected behavior.

Example 5.5.1: Does the following code compile and print the larger of two integers `a` and `b` correctly?

```
if (a > b)      System.out.println("a");
else {
System.out.println("b");
}
```

Answer: Yes! Even though the whitespacing and indentation look funny, this piece of code compiles correctly.

5.6 Common mistakes

When writing conditional structures, beware of the following common mistakes:

5.6.1 Forgetting parentheses

In Java, the parentheses surrounding the boolean expression in conditional structures are required. For example, the following code will not compile.

```
if count > 10
    System.out.println("So many!");
```

5.6.2 Accidental semi-colons

Accidentally semi-colons immediately after the parentheses around the boolean expression in an `if` statement is one of the trickiest mistakes to detect. The following code compiles, but it behaves unexpectedly.

```
if (count > 10);
    System.out.println("So many!");
```

The code above is misleading because it is identical to the following:

```
if (count > 10) {
    ;
}
System.out.println("So many!");
```

The accidental semi-colon is treated as a single, "do-nothing" statement.

5.6.3 Missing curly braces

As mentioned before, indentation helps make code more readable to humans, but it can also make code more confusing if used incorrectly. It is a common mistake to accidentally forget curly braces.

```
if (leaves = 4) {
    System.out.println("You found a four-leaf clover, how lucky!");
} else
    System.out.println("Sorry, not a four-leaf clover.");
    System.out.println("Keep looking!");
```

This code above is misleading because it prints "Keep looking!" regardless of whether a four-leaf clover was found. The writer of this program probably meant to add curly braces around the `else` clause.

5.6.4 `else` without an `if`

Although the following example looks fine at first glance, it does not compile. This is because the `else` clause is nested *inside* the `if` statement instead of acting as an *alternative* option to the `if` statement.

```
if (chanceOfRain > 50) {
    System.out.println("Bring an umbrella!");
    else {
        System.out.println("Hm, I don't think it will rain today.")
    }
}
```

To fix the code above, move the `else` statement *outside* of the `if` as below:

```
if (chanceOfRain > 50) {
    System.out.println("Bring an umbrella!");
} else {
    System.out.println("Hm, I don't think it will rain today.")
}
```

5.6.5 Assignment v.s. equality operator

It is very easy to mix up the assignment operator (the single equals sign `=`) with the equality operator (the double equals sign `==`). The following code is incorrect and results in a compilation error:

```
if (temperature = 0) {
    System.out.println("It's freezing!");
}
```

The assignment operator cannot be used here because conditional structures require a boolean expression (i.e. something that evaluates to either true or false).

Exercises

5.1 What output is produced by the following code fragment?

```
int num = 87;
int max = 25;
if (num >= max*2)
    System.out.println("apple");
    System.out.println("orange");
System.out.println("pear");
```

TODO: modify this exercise, or make sure we can reference the Java Foundations textbook

5.2 What output is produced by the following code fragment?

```
int limit = 100;
int num1 = 15;
int num2 = 40;
if (limit <= limit) {
    if (num1 == num2)
        System.out.println ("lemon");
    System.out.println ("lime");
}
System.out.println ("grape");
```

TODO: modify this exercise, or make sure we can reference the Java Foundations textbook

5.3 Given a day of the week encoded as 0=Sun, 1=Mon, 2=Tue, ...6=Sat, and a boolean indicating if we are on vacation, we need to decide when to set our alarm clock to ring. We want to wake up at "7:00" on weekdays and "10:00" on weekends. However, if we are on vacation, it should be "10:00" on weekdays" and "off" on weekends. Imagine you are given an integer variable `day` and a boolean variable `vacation`. Write a program that prints out the appropriate alarm time.

TODO: modify this exercise, or make sure we can reference codingbat.com

5.4 Your cell phone rings. Normally you answer, except in the morning you only answer if it is your mom calling. In all cases, if you are asleep, you do not answer. Given three booleans, `isMorning`, `isMom`, and `isAsleep`, write a program that correctly prints out whether or not you will pick up your phone ("Answer" or "Do not answer").

For example, if `isMorning`, `isMom`, and `isAsleep` are all false, print "Answer".

TODO: modify this exercise, or make sure we can reference codingbat.com

5.5 You have a lottery ticket with a 3-digit number. If your ticket has the number 777, you win $100. If your ticket has a number with the same 3 digits (like 222, for example), then you win $10. Otherwise, you don't earn any money. Given three integers, `a`, `b`, and `c`, that represent the digits of your lottery ticket, write a program that prints out how much money you earn.

For example, if your ticket number is 123 (`a` = 1, `b` = 2, `c` = 3), print "$0".

TODO: modify this exercise, or make sure we can reference codingbat.com

---

References

Computer Science: An Interdisciplinary Approach, Robert Sedgewick and Kevin Wayne.

Lewis, John, Peter DePasquale, and Joseph Chase. Java Foundations: Introduction to Program Design and Data Structures. Addison-Wesley Publishing Company, 2010.

Programming exercises from codingbat.com

6

Business Software

6.1 Introduction

This lecture will go over three primary types of business productivity software most businesses use on a daily basis:

1. Word processing software: for typing and formatting text and images on a page (such as essays and reports)
2. Presentation software: for creating slides with information and graphics (such as for meetings and briefings)
3. Spreadsheet software: for storing data, doing computations, and making graphs

6.2 Word Processing

Word processing software is used for creating, editing, and formatting documents. Many tech companies have their own version of word processing software. A few more familiar examples include

- Microsoft Word
- Apple Pages
- Google Docs

Beyond providing an interface to type the words of your report or essay into, word processors allow a variety of styling and formatting options to customize the document to look the way you like it. For example, all of the word processors above support

- Automatic page numbering
- Adjustable margins
- Varieties of text size, text font, text color
- Addition of images and captions, along with image formatting
- Addition of tables and lists
- Spelling and grammar checking

6.3 Spreadsheets

Spreadhseet software is used for dealing with data. A few examples of spreadsheet software include

- Microsoft Excel
- Apple Numbers
- Google Sheets

Beyond providing a grid in which you can type numbers, spreadsheet software facilitate a variety of operations on the data. styling. A few key features supported by most spreadsheet applications include

- Statistical analysis
- Sorting
- Creating charts and graphs
- Formatting (e.g. adding color, borders, etc.)

6.4 Presentations

Presentation software is used for presenting data and graphics to an audience. Some common examples include

- Microsoft Powerpoint
- Apple Keynote
- Google Slides

All of these software give the user a series of "slides" that can be clicked through during a presentation. It allows the presenter to focus the audience's attention on a single point at a time. The slides are not meant to contain all the information that the presenter wants to get across. Instead, the slides only mention the key takeaways, and the presenter should verbally fill in the details. For that reason, slides often consist of just a few bullet points and an image. Of course, there are a variety of creative ways to present - this is just the most common format!

A few particularly important features offered by almost all presentation software include

- Automatic styling (color schemes, layouts, etc)
- Add images and videos
- Slide transitions
- Animations (e.g. showing one line of text on a slide at a time)

6.5 Project

7

Methods

Sometimes it is useful to abstract certain computations into methods. In this chapter, we will be looking at how and why we can use methods to achieve more concise and easily-maintainable code.

We will first look at some code that does not use methods and identify some problems with this code. We will then, in Section 7.1, introduce what a method is, including how to define and use them. In Sections 7.2 and 7.3, we will then talk about how we can use methods to improve the problems that we noticed with our code. In particular, in Section 7.2, we talk about how methods allow us to abstract some of our code to get more concise and maintainable code, and in Section 7.3, we talk about how methods allow us to divide our code up into chunks that we can consider in isolation, allowing us to manage and maintain our code more easily.

Let us consider the following problem:

We have a class of nine students, split up into three groups of three. For each group, we want to report the maximum score for each group. We will use the variable names `group1s1`, `group1s2`, and `group1s3` to refer to the three scores achieved by the students in Group 1, and we will use a similar naming convention for the variable names for the scores achieved by students in Group 2 and Group 3.

The following snippet of code computes the maximum score in Group 1, storing the result in `group1Max`.

```
int group1Max = group1s1;
if (group1s2 > group1Max) {
  group1Max = group1s2;
}
if (group1s3 > group1Max) {
  group1Max = group1s3;
}
```

If we now wish to extend this to compute the maximum score for each group, we can copy the code and rename the variables to achieve the following code:

```
int group1Max = group1s1;
if (group1s2 > group1Max) {
  group1Max = group1s2;
}
if (group1s3 > group1Max) {
  group1Max = group1s3;
}

int group2Max = group2s1;
if (group2s2 > group1Max) {
  group2Max = group2s2;
}
if (group2s3 > group2Max) {
```

```
14    group2Max = group2s3;
15  }
16
17  int group3Max = group3s1;
18  if (group3s2 > group3Max) {
19    group3Max = group3s2;
20  }
21  if (group3s3 > group3Max) {
22    group3Max = group3s3;
23  }
```

There are a couple of things about the process of getting this resulting code that aren't particularly ideal. The first is that it is very easy to make a mistake when copying the code and renaming the variables. For example, we might have missed renaming `group1Max` to `group3Max` in the computation of the maximum score for Group 3, leading us to compute an incorrect maximum value in some cases.

The second issue is that there is a lot of redundancy. If we look at the code, we can notice that there seem to be a lot of redundant parts across the code for the three different groups. If we had ended up with the wrong computation initially and needed to fix it later, then we would have to make sure we fixed it everywhere, leading to repeated work and the possibility that we didn't fix one of the copies. For example, if we initially started out with `group1s3 < group1Max` instead of `group1s3 > group1Max`, and then copied and renamed variables to compute the maximum scores for the other groups, we would end up with the code below:

```
1   int group1Max = group1s1;
2   if (group1s2 > group1Max) {
3     group1Max = group1s2;
4   }
5   if (group1s3 < group1Max) {
6     group1Max = group1s3;
7   }
8
9   int group2Max = group2s1;
10  if (group2s2 > group1Max) {
11    group2Max = group2s2;
12  }
13  if (group2s3 < group2Max) {
14    group2Max = group2s3;
15  }
16
17  int group3Max = group3s1;
18  if (group3s2 > group3Max) {
19    group3Max = group3s2;
20  }
21  if (group3s3 < group3Max) {
22    group3Max = group3s3;
23  }
```

To fix our code, we would not only have to replace `group1s3 < group1Max` with `group1s3 > group1Max`, but we would also need to replace `group2s3 < group2Max` with `group2s3 > group2Max` and `group3s3 < group3Max` with `group3s3 < group3Max`.

7.1 Methods

In Java, we can use abstractions in our code in the form of *methods*. Methods contain bits of code that may compute things using *arguments* that are passed into the method.

An example of the syntax for a *method definition* is given below:

```
1   public bool negate(bool arg) {
2     return !arg; // method body
3   }
```

We can identify the different components of the method definition:

- The *access modifier* of the method is `public`,
- the `bool` before `negate` gives the *return type* of the method
- `negate` is the *name of the method*,
- `arg` is the *name of the first argument/parameter* of the method,
- the `bool` before `arg` it is the *type* of the `arg` parameter,
- and the code between the curly braces is the method *body*.

Inside the method body, there is code that may use the arguments of the method. In the example above, the body only consists of the `return` statement. The `return` statement is followed by something of the return type of the method. This expression, after being evaluated, gives the *return value* of the method. As soon as a `return` statement is encountered, a method finishes its execution. *Access modifiers* will be described later on when we talk about Classes, but note that if an access modifier is omitted, it the method has the *default* modifier.

Other important terms to note are as follows: The *parameter list* of a method consists of the types of its parameters in the order they are given in the method definition. For the above method, it is simply `bool`. The *method signature* consists of the method name and parameter list. For the above method, it is `negate(bool)`. A method is identified by its method signature.

Example 7.1.1: Consider the following method:

```
int squareSum(int arg0, int arg1) {
  int res = arg0 + arg1;
  return res * res;
}
```

Give the following for the method:

- Access modifier
- Return type
- Arguments
- Method signature

Answer:

- Access modifier: default
- Return type: `int`
- Arguments: `arg0`, `arg1`
- Method signature: `squareSum(int, int)`

Example 7.1.2: Consider the following method:

```
private void mystery (int age, String name) {
  if (age >= 18) {
    System.out.println(name);
  }
}
```

Give the following for the method:

- Access modifier
- Return type
- Name
- Parameter list

Answer:

- Access modifier: `private`
- Return type: `void`
- Name: `mystery`
- Parameter list: `int, String`

We may *call* or *invoke* the above `negate` method from elsewhere in the code by, for example, using `negate(true)` if we want to call `negate` with argument `true`. We must specify the name of the method that we wish to call, followed by the arguments that we wish to call it on, separated by commas if there are more than one. The arguments provided in the call must correspond to the types of the arguments specified in the method definition: if the method definition has the type `bool` for its first argument, then the first argument supplied in the call should also be of type `bool`, and if the definition has the type `int` for its second argument, then the first argument in the call should also be of type `int`, and so on. A method invocation evaluates to the return value it computes, where this computation happens with the actual arguments given in the call substituted for the arguments specified in the definition, so in this case, the invocation `id(true)` evaluates to `false`. A method invocation has the same type as its return type, which in this case, is `bool`.

So, for example, we can do something like in following code snippet:

```
1    bool f = negate(true);
2    bool t = negate(negate(true));
```

After executing the above code, the variable `f` contains the value `false` and the variable `t` contains the value `true`.

Note that we can also have methods that do not return anything. Such methods have `void` return types. They may contain a `return` statement with no expression following `return`. Calls to these methods cannot be used inside of other expressions. Two examples of methods with `void` return types are below:

```
1  public void printSum(double a, double b) {
2    System.out.println(a + b);
3  }
```

```
1  public void printBigger(double a, double b) {
2    if (a > b) {
3      System.out.println(a);
4      return;
5    }
6    System.out.println(b);
7  }
```

Note that the latter example only ever prints the value of either `a` or `b` but never both because the `return` statement finishes the execution of the method.

Some methods also do not take any arguments at all. One such example is below:

```
public int constantOne() {
  return 1;
}
```

This method can be called using `constantOne()`.

Example 7.1.3: What value does `res` have after the following code snippet is executed?

```
int res = constantOne() + constantOne();
```

Answer: The value of `res` is 2.

.

Note that for every method that does not have a `void` return type, for every possible control-flow path through the method body, there must be a `return` statement that is eventually reached. For example, the following method does not meet this requirement because when `arg` is `false`, the `return` statement is not reached:

```
public int twoIfTrue(bool arg) {
  if (arg) {
    return 2;
  }
}
```

Example 7.1.4: Consider the following method:

```
public void mystery (bool arg0, int arg1) {
  if (arg0) {
    return -1 * arg1;
  }
  return arg0;
}
```

What do each of the following evaluate to?

- `mystery(true, -2)`
- `mystery(true, 0)`
- `mystery(true, 4)`
- `mystery(false, -2)`
- `mystery(false, 0)`
- `mystery(false, 4)`

Answer:

- `mystery(true, -2)` evaluates to 2
- `mystery(true, 0)` evaluates to 0
- `mystery(true, 4)` evaluates to -4
- `mystery(false, -2)` evaluates to -2
- `mystery(false, 0)` evalutes to 0

- `mystery(false, 4)` evalutates to 4

What does the method do?
Answer: It negates its second argument `arg1` whenever its first argument `arg0` is `true` and otherwise just returns its second argument `arg1`.

Example 7.1.5: How many methods can be called by the following method? What do you think each of their return types and parameter lists are?

```
public void sayHello(int times, String name, String day) {
  if (shouldStop(times))
    return;
  String hello = getHelloStr(name, day);
  System.out.print.ln(hello);
  sayHello(times - 1, name, day);
}
```

Answer: Three methods can be called: `shouldStop`, `getHelloStr` and `sayHello`.

The return type of `shouldStop` is `bool`, and it has parameter list `int`. The return type of `getHelloStr` is `String`, and it has parameter list `String, String`. The return type of `sayHello` is `void`, and it has parameter list `int, String, String`.

Example 7.1.6: Write a method with signature `printInOrder(int, int, int)` that takes three `int`s `a`, `b`, and `c` as arguments and prints them out, one per line, in increasing order. E.g. `printInOrder(5, 9, 7)` should output the following:

```
5
7
9
```

Answer: A possible solution is the following:

```
public void printInOrder(int a, int b, int c) {
  int smallest = a;
  int middle = b;
  int biggest = c;
  if (smallest > middle) {
    int tmp = smallest;
    smallest = middle;
    middle = tmp;
  }
  if (smallest > biggest) {
    int tmp = smallest;
    smallest = biggest;
    biggest = tmp;
  }
  if (middle > biggest) {
    int tmp = middle;
    middle = biggest;
    biggest = tmp;
  }
  System.out.println(smallest);
  System.out.println(middle);
  System.out.println(biggest);
}
```

7.1.1 Overloading

As mentioned above, methods are identified by their signature, not just their names. Because of this, we can actually have methods with the same name (in the same Class) as long as they have different parameter lists.

For example, we can have the following two methods because one has signature `printBigger(double, double)` and the other has signature `printBigger(int, int)`:

```
public void printBigger(double a, double b) {
  if (a > b) {
    System.out.println(a);
    return;
  }
  System.out.println(b);
}
public void printBigger(int a, int b) {
  if (a > b) {
    System.out.println(a);
    return;
  }
  System.out.println(b);
}
```

In this case, we say that the `printBigger` method is *overloaded*.

However, we cannot have the following two methods because they both have the same signature `twoIfTrue(bool)`:

```
public int twoIfTrue(bool arg) {
  if (arg) {
    return 2;
  }
}
public double twoIfTrue(bool arg) {
  if (arg) {
    return 2.0;
  }
}
```

Example 7.1.7: Consider the following two methods that we have seen in previous examples:

```
private void mystery (int age, String name) {
  if (age >= 18) {
    System.out.println(name);
  }
}
public void mystery (bool arg0, int arg1) {
  if (arg0) {
    return -1 * arg1;
  }
  return arg0;
}
```

Are we allowed to have both of these methods together (in the same Class)? Why or why not?

Answer: We are allowed to have both of these methods together because they have different signatures. One has signature `mystery(int, String)` and the other has signature `mystery(bool, int)`.

Example 7.1.8: Which of the following are valid overloadings?
A:

```
public int absValue(int x) {
  if (x < 0) {
    return x * -1;
  }
  return x;
}
public double absValue(int x) {
  if (x < 0) {
    return 0.0 - x;
  }
  return x - 0.0;
}
```

B:

```
public int absValue(int x) {
  if (x < 0) {
    return x * -1;
  }
  return x;
}
public double absValue(double x) {
  if (x < 0) {
    x = 0.0 - x;
  }
  return x;
}
```

C:

```
public int absValue(int x) {
  if (x < 0) {
    return x * -1;
  }
  return x;
}
public int absValueX(int x) {
  if (x < 0) {
    return x * -1;
  }
  return x;
}
```

Answer:

- A is not a valid overloading because both methods have the same signature.
- B is a valid overloading because both methods have different signatures.
- C is valid code, but this is not a valid overloading. It is not overloading at all because the methods have different names.

7.1.2 Methods Summary

In summary, every method definition has the following:

- An access modifier (if not explicitly stated, this will be the *default* modifier)
- A return type (possibly `void`)

- A name
- A (possibly empty) list of parameter types and names that can be passed to the method when it is called
- A body
- A return statement reached on every control path in the body, where each `return` is followed by an expression of the method's return type (unless the return type is `void`)

Each method is identified by its signature, so you may have methods with the same name but different parameter lists.

7.2 Abstraction

Let us now return to our problem of finding the maximum scores for each of the three groups. The code for each of the three groups, though similar, have different variable names; however, this is the only way in which they are different. The process of distilling the similarity among different pieces of code is the process of *abstraction*. In this case, we would like to *abstract* away the details of each copy to achieve a piece of code that describes all of their behavior.

After abstracting away the variable names, the code for each group is such that, if we use the appropriate variable names for the students in the group instead of `x`, `y`, and `z`, that the following code snippet would describe all of the three different operations, with `max` storing the desired result:

```
int max = x;
if (y > max) {
  max = y;
}
if (z > max) {
  max = z;
}
```

We can wrap this code snippet up in a method definition that returns the value we care about:

```
public int max3(int x, int y, int z) {
  int max = x;
  if (y > max) {
    max = y;
  }
  if (z > max) {
    max = z;
  }
  return max;
}
```

If we use this method in place of the redundant code, our original code that compute the maximum scores for each of the three groups then becomes as follows:

```
int group1Max = max3(group1s1, group1s2, group1s3);
int group2Max = max3(group2s1, group2s2, group2s3);
int group3Max = max3(group3s1, group3s2, group3s3);
```

This resulting code is much more concise and is an example of *code reuse*, where the same exact code (i.e. the code inside the body of `max3`) is being reused in several places. This code is also perhaps easier to understand if you know that `max3` simply calculates the maximum of its three arguments. In the original code, after going through the code

for one of the groups and realizing that it computes a maximum, you would have to go through the calculations of the other groups to make sure that they also compute a maximum. Here, it is easy to see that the same computation is happening for each group (but with different inputs).

We might further notice that both of the conditional statements in the `max3` method compute very similar things (i.e. the maximum of two numbers), and might perform further abstraction to achieve the following method:

```
int max2(int x, int y) {
  int max = x;
  if (y > max) {
    max = y;
  }
}
```

We can then adjust our `max3` method to call this one:

```
int max3(int x, int y, int z) {
  return max2(max2(x, y), z);
}
```

Note that we do not need to change our code for computing `group1Max`, `group2Max`, or `group3Max` in order to use the method `max2`. The changes to `max3` are sufficient.

7.3 Modularity

Let us again consider the case in which our calculation of the maximum score is incorrect because we used `<` instead of `>` in the last comparison. Let us assume that we are still using the version of `max3` that does not call `max2`, but we mistakenly have the comparison `z < max` instead of `z > max` in our method:

```
public int max3(int x, int y, int z) {
  int max = x;
  if (y > max) {
    max = y;
  }
  if (z < max) {
    max = z;
  }
  return max;
}
```

In order to fix, this we simply need to change `z < max` to `z > max` once. The code that calculates the values of `group1Max`, `group2Max`, and `group3Max` by calling `max3` is fixed by this one change because for all the groups, we call the same method. Here we have achieved a *separation of concerns* in the two parts of our code:

- We have one part of our code (the `max3` method) that is concerned with finding the maximum of three *arbitrary* numbers, but it is not concerned with *which* numbers specifically for which it is finding the maximum.
- We have another part that is concerned with computing the maximum scores for each group, *given that* we have some other code that can calculate the maximum of three numbers, but it is not concerned with *how* this maximum is found, as long as it is done correctly

This separation of concerns is referred to as *modularity*, and we can regard the code inside of different methods as being in different *modules*. We have already seen that modularity can allow us to do things like fix bugs in one part of our code without having to touch other parts of our code.

Modularity also lets us do other things, like naturally divide up labor. If you wanted to split up the work of writing a program that finds the maximum of each group's three scores with a friend, you might volunteer to write the code that deals with finding the maximum scores for each group provided that your friend writes code that finds the maximum of three numbers contained in a method with signature `max3(int, int int)` that returns an `int` that is the maximum of its three `int` arguments.

Given that you know your friend will write such a `max3` method, you can, without seeing your friend's code, write the following code (that we have already seen):

```
int group1Max = max3(group1s1, group1s2, group1s3);
int group2Max = max3(group2s1, group2s2, group2s3);
int group3Max = max3(group3s1, group3s2, group3s3);
```

Once your friend finishes writing `max3`, then your code should work as expected.

An advantage of modularity is that it is not necessary to know the details of how the modules that you use are implemented nor used. In the above example, you do not need to know how exactly your friend implements `max3` (maybe it calls `max2` and maybe it does not) and your friend does not need to know how you use `max3` (maybe you also use it to find the maximum of the three maximum scores and maybe you do not).

Example 7.3.1: What would you add to the following code to calculate the maximum of `group1Max`, `group2Max`, and `group3Max` and store it in `allGroupMax`?

```
int group1Max = max3(group1s1, group1s2, group1s3);
int group2Max = max3(group2s1, group2s2, group2s3);
int group3Max = max3(group3s1, group3s2, group3s3);
```

Answer:

```
int group1Max = max3(group1s1, group1s2, group1s3);
int group2Max = max3(group2s1, group2s2, group2s3);
int group3Max = max3(group3s1, group3s2, group3s3);
// added code below:
int allGroupMax = max3(group1Max, group2Max, group3Max);
```

Example 7.3.2: Consider the case where not only do you want to compute the maximum scores for each group for Group 1, Group 2, and Group 3, but you also want to do the same for Group 4 and Group 5. Unfortunately, while Group 4 and Group 5 also have three scores per group (`group4s1`, `group4s2`, `group4s3`, `group5s1`, `group5s2`, and `group5s3`), these scores are all `double`s rather than `int`s.

You want write the following code:

```
int group1Max = max3(group1s1, group1s2, group1s3);
int group2Max = max3(group2s1, group2s2, group2s3);
int group3Max = max3(group3s1, group3s2, group3s3);
double group4Max = max3(group4s1, group4s2, group4s3);
double group5Max = max3(group5s1, group5s2, group5s3);
```

What code do you ask your friend to write so that your code works?

Answer: You can ask your friend to write an overloaded `max3`: one overloading (the one we have seen so far) should return an `int` and have method signature `max3(int, int, int)` and the other should return a `double` and have method signature `max3(double,`

`double, double)`. Both versions of `max3` should return the maximum value of their three arguments.

Example 7.3.3: Give the signatures of methods that you would need to write to make the following code work:

```
int computeAverages(int a, int b, int c, int d, int e, int f, int g, int
    h) {
  int avg0 = sum(a, b) / 2;
  int avg1 = sum(c, d, e) / 3;
  int avg2 = sum(f, g, h) / 3;
  int avgAvg = sum(avg0, avg1, avg2) / 3;
  return avgAvg;
}
```

Answer: `sum(int, int)` and `sum(int, int, int)`

Example 7.3.4: This is the same code as in the previous example, but someone made a mistake and divided by 4 instead of 3 when taking the average of three numbers:

```
int computeAverages(int a, int b, int c, int d, int e, int f, int g, int
    h) {
  int avg0 = sum(a, b) / 2;
  int avg1 = sum(c, d, e) / 4;
  int avg2 = sum(f, g, h) / 4;
  int avgAvg = sum(avg0, avg1, avg2) / 4;
  return avgAvg;
}
```

Improve the original code by using abstraction.
Answer:

```
int avg(int a, int b) {
  return sum(a, b) / 2;
}
int avg(int a, int b, int c) {
  return sum(a, b, c) / 3;
}
int computeAverages(int a, int b, int c, int d, int e, int f, int g, int
    h) {
  int avg0 = avg(a, b);
  int avg1 = avg(c, d, e);
  int avg2 = avg(f, g, h);
  int avgAvg = sum(avg0, avg1, avg2) / 4;
  return avgAvg;
}
```

8

Arrays

Consider this snippet of code:

```
if      (day ==  0) System.out.println("Monday");
else if (day ==  1) System.out.println("Tuesday");
else if (day ==  2) System.out.println("Wednesday");
else if (day ==  3) System.out.println("Thursday");
else if (day ==  4) System.out.println("Friday");
else if (day ==  5) System.out.println("Saturday");
else if (day ==  6) System.out.println("Sunday");
```

What does this code do? It prints the day of the week after conditioning on the value of an integer `day`. But this code is repetitive. It would be useful if we had some way of creating a list of days of the week, and then just specifying which of those days we wanted to print. Something like this:

```
System.out.println(DAYS\_OF\_WEEK[day]);
```

To achieve this in Java, we need arrays.

Definition 8.0.1. An *array* is an ordered and fixed-length list of values that are of the same type. We can access data in an array by *indexing*, which means referring to specific values in the array by number. If an array has `n` values, then we think of it as being numbered from `0` to `n-1`.

To *loop* or *iterate* over an array means that our program accesses every value in the array, typically in order. For example, if we looped over the array in the diagram, that would mean that we looked at the value at the 0th index, then the value at the 1st index, then the value at the 2nd index, and so on.

When we say that the array is "ordered" is that the relationship between an index and its stored value is unchanged (unless we explicitly modify it). If we loop over an unchanged array multiple times, we will always access the same values.

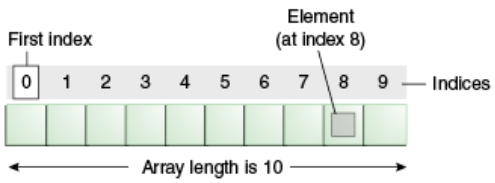


Figure 8.1: Diagram of an array (Credit: https://www.geeksforgeeks.org/arrays-in-java/)

Arrays are *fixed-length*, meaning that after we have created an array, we cannot change its length. We will see in the next chapter [TK: confirm] that `ArrayLists` are an array-like data structure that allows for changing lengths.

Finally, all the values in an array must be of the same type. For example, an array can hold all floating point numbers or all characters or all strings. But an array cannot hold values of different types.

8.1 Creating arrays

The syntax for creating an array in Java has three parts:

1. Array type
2. Array name
3. Either: array size or specific values

For example, this code creates an array of size `n = 10` and fills it with all `0.0`s

```
double[] arr;                       // Declare array
arr = new double[n];                // Initialize the array
for (int i = 0; i < n; i++) {       // Iterate over array
    arr[i] = 0.0;                   // Initialize elements to 0.0
}
```

The key steps are: we first declare and initialize the array. We then loop over the array to initialize specific values. We can also initialize the array at compile time, for example

```
String[] DAYS_OF_WEEK = {
//  Indices:
//  0      1      2      3      4      5      6
    "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"
};
```

Notice the difference in syntax. When creating an empty array, we must specify a size. When initialize an array at compile time with specific values, the size is implicit in the number of values provided.

Finally, in Java, it is acceptable to move the brackets to directly after the type declaration to directly after the name declaration. For example, these two declarations are equivalent:

```
int arr[];
int[] arr;
```

8.2 Indexing

Consider the array `DAYS_OF_WEEK` from the previous section. We can *index* the array using the following syntax:

```
System.out.println(DAYS_OF_WEEK[3]);  // Prints "Thu"
```

In Java, array's are said to use *zero-based indexing* because the first element in the array is accessed with the number `0` rather than `1`.

Example 8.2.1: What does `System.out.println(DAYS_OF_WEEK[1]);` print?

Example 8.2.2: What does this code do? What number does it print?

```
double sum = 0.0;
double[] arr = { 1, 2, 2, 3, 4, 7, 9 }
for (int i = 0; i < arr.length; i++) {
    sum += arr[i];
}
System.out.println(sum / arr.length);
```

8.3 Array length

As mentioned previously, arrays are *fixed-length*. After you have created an array, it's length is unchangeable. You can access the length of an array `arr[]` with the code `arr.length`.

Example 8.3.1: What does `System.out.println(DAYS_OF_WEEK.length);` print?

Example 8.3.2: Write a `for` loop to print the days of the week in order (Monday through Sunday) using an array rather than seven `System.out.println` function calls.

8.4 Default initialization

In Java, the default initial values for numeric primitive types is `0` and `false` for the `boolean` type.

Example 8.4.1: Consider this code from earlier:

```
double[] arr;
arr = new double[n];
for (int i = 0; i < n; i++) {
    arr[i] = 0.0;
}
```

Rewrite this code to be a single line.

8.5 Bounds checking

Consider this snippet of code.

Example 8.5.1: Where is the bug?

```
int[] arr = new int[100];
for (int i = 0; i <= 100; ++i) {
    System.out.println(arr[i]);
```

```
}
```

The issue is that the program attempts to access the value `arr[100]`, while the last element in the array is `arr[99]`.

This kind of bug is called an "off-by-one error" and is so common it has a name. In general, an off-by-one-error is one in which a loop iterates one time too many or too few.

Example 8.5.2: Where is the off-by-one-error?

```
int[] arr = new int[100];
for (int i = 0; i < array.length; i++) {
    arr[i] = i;
}
for (int i = 100; i > 0; --i) {
    System.out.println(arr[i]);
}
```

Example 8.5.3: Fill in the missing code in this `for` loop to print the numbers in reverse order, i.e. `5, 4, 3, 2, 1`:

```
int[] arr = { 1, 2, 3, 4, 5 };
for (???) {
    System.out.println(arr[i]);
}
```

8.6 Empty arrays

This code prints five values, one per line, but we never specified which values. What do you think it prints?

```
int[] arr = new int[5];
for (int i = 0; i < arr.length; i++) {
    System.out.println(arr[i]);
}
```

In Java, an unitialized or empty array is given a default value:

- For `int`, `short`, `byte`, or `long`, the default value is `0`.
- For `float` or `double`, the default value is `0.0`
- For `boolean` values, the default value is `false`.
- For `char`, the default value is the null character `'\0000'`.

Note that an array can be partially initialized.

Example 8.6.1: What does this code print?

```
char[] alphabet = new char[26];
alphabet[0] = 'a';
alphabet[1] = 'b';
```

```
for (int i = 0; i < alphabet.length; i++) {
    System.out.println(alphabet[i]);
}
```

8.7 Enhanced for loop

So far, we have seen how to iterate over arrays by indexing each element with a number:

```
char[] vowels = {'a', 'e', 'i', 'o', 'u'};
for (int i = 0; i < vowels.length; ++ i) {
    System.out.println(vowels[i]);
}
```

We can perform the same iteration without using indices using an "enhanced `for` loop" or `for-each` loops:

```
char[] vowels = {'a', 'e', 'i', 'o', 'u'};
for (char item: vowels) {
    System.out.println(item);
}
```

8.8 Exchanging and shuffling

Two common tasks when manipulating arrays are *exchanging two values* and *shuffling* values. (*Sorting* is more complicated and will be address later.)

To exchange to values, consider the following code:

```
double[] arr = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 };
int i = 1;
int j = 4;
double tmp = arr[i];
arr[i] = arr[j];
arr[j] = tmp;
```

Example 8.8.1: What are the six values in the array, in order?

To shuffle the array, consider the following code:

```
int n = arr.length;
for (int i = 0; i < n; i++) {
    int r = i + (int) (Math.random() * (n-i));
    String tmp = arr[r];
    arr[r] = arr[i];
    arr[i] = tmp;
}
```

Example 8.8.2: What does this code do:

```
for (int i = 0; i < n/2; i++) {
    double tmp = arr[i];
    arr[i] = arr[n-1-i];
```

```
4       arr[n-i-1] = tmp;
5   }
```

Exercises

8.1 Write a program that reverses the order of values in an array.

8.2 What is wrong with this code snippet?

```
1   int[] arr;
2   for (int i = 0; i < 10; i++) {
3       arr[i] = i;
4   }
```

8.3 Rewrite this snippet using an enhanced `for-each` loop (for now, it is okay to re-define the array):

```
1   char[] vowels = {'a', 'e', 'i', 'o', 'u'};
2   for (int i = array.length; i >= 0; i--) {
3       char letter = vowels[i];
4       System.out.println(letter);
5   }
```

8.4 Write a program that uses `for` loops to print the following pattern:

```
1   1********
2
3   12*******
4
5   123******
6
7   1234*****
8
9   12345****
10
11  123456***
12
13  1234567**
14
15  12345678*
16
17  123456789
```

8.5 Write a program `HowMany.java` that takes an arbitrary number of command line arguments and prints how many there are.

References

Computer Science: An Interdisciplinary Approach, Robert Sedgewick and Kevin Wayne.

9

ArrayLists

A *collection* is a group of objects. Today, we'll be looking at a very useful collection, the `ArrayList`. A *list* is an ordered collection, and an `ArrayList` is one type of list. We will see at the end of this chapter how an `ArrayList` is different from an `array`.

9.1 Creating an `ArrayList`

Let's create a class `NameTracker` and follow along in it. Before we can use an ArrayList, we have to import it:

```
1 import java.util.ArrayList;
2
3 public class NameTracker {
4
5     public static void main(String args[]) {
6     }
7 }
```

Next, we call the constructor; but we have to declare the type of object the `ArrayList` is going to hold. This is how you create a new `ArrayList` holding `String` objects.

```
1 ArrayList<String> names = new ArrayList<String>();
```

Notice the word "String" in angle brackets. This is the Java syntax for constructing an ArrayList of String objects.

9.2 `add()`

We can add a new String to names using the `add()` method.

```
1 names.add("Ana");
```

Example 9.2.1: Exercise: Write a program that asks the user for some names and then stores them in an ArrayList. Here is an example program:

```
1 Please give me some names:
2 Sam
3 Alecia
4 Trey
```

```
Enrique
Dave

Your name(s) are saved!
```

We can see how many objects are in our ArrayList using the size() method.

```
System.out.println(names.size()); // 5
```

Example 9.2.2: Modify your program to notify the user how many words they have added.

```
Please give me some names:
Mary
Judah

Your 2 name(s) are saved!
```

9.3 get()

Remember how the `String.charAt()` method returns the `char` at a particular index? We can do the same with names. Just call `get()`:

```
names.add("Noah");
names.add("Jeremiah");
names.add("Ezekiel");
System.out.println(names.get(2)); // ''Ezekiel''
```

Example 9.3.1: Update your program to repeat the names back to the user in reverse order. Your solution should use a for loop and the size() method. For example:

```
Please give me some names:
Ying
Jordan

Your 2 name(s) are saved! They are:
Jordan
Ying
```

9.4 contains()

Finally, we can ask our names `ArrayList` whether or not it has a particular string.

```
names.add("Veer");
System.out.println(names.contains("Veer")); // true
```

Example 9.4.1: Update your program to check if a name was input by the user. For example:

```
Please give me some names:
Ying
Jordan

Search for a name:
Ying

Yes!
```

9.5 ArrayLists with custom classes

An `ArrayList` can hold any type of object! For example, here is a constructor for an `ArrayList` holding an instance of a `Person` class:

```
ArrayList<Person> people = new ArrayList<Person>();
```

where `Person` is defined as

```
public class Person {

    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return this.name;
    }

    public int getAge() {
        return this.age;
    }
}
```

Example 9.5.1: Modify our program to save the user's input names as Person instances. Rather than storing String objects in the ArrayList, store Person objects by constructing them with the input name. You'll need to use the Person constructor to get a Person instance!

9.6 Arrays versus ArrayLists

The most important difference between an array and an `ArrayLists` is that an `ArrayLists` is dynamically resized. Notice that when we created an `ArrayLists`, we did not specify a size. But when we create an array, we must:

```
int[] numbers1 = new int[10];
ArrayList<Integer> numbers2 = new ArrayList<Integer>();
```

In the first line of code, we create an array of length 10, and we cannot add more than 10 elements to `numbers1`. In the second line of code, we create an `ArrayList`, but we do not need to specify a size.

The second distinction between arrays and `ArrayLists` is syntax. In Java, nearly everything is a class. An `ArrayList` is a class with useful methods such as `get()` and `contains()` and is therefore more "Java-like". Arrays are not quite primitives such as `ints` and note quite classes. Most likely, their syntax was borrowed from the C or C++ programming languages.

Exercises

9.1 Write a class BlueBook that tells the user the price of their car, depending on the make, model, and year. You should use Car.java and the stencil file provided, BlueBook.java. Your program depends on what cars your BlueBook supports, but here is an example program:

```
What is your car's make?
Toyota
What is your Toyota's model?
Corolla
What is your Toyota Corolla's year?
1999

Your 1999 Toyota Corolla is worth \$2000.
```

9.2 Notify the user if the car is not in your BlueBook.

9.3 Clean up main by putting your code for creating the ArrayList in a separate method. What type should the method return?

9.4 If the car is not in the BlueBook, ask the user to input the relevant data, construct a new Car instance, add it to your ArrayList.

References

1. `https://github.com/accesscode-2-1/unit-0/blob/master/lessons/week-3/2015-03-24_arraylists.md`

10

Systems Development

10.1 Introduction

Creating good software requires good coding skills, so you can get the computer to do what you want. Additionally, however, it requires good "systems development" skills so that (1) your code is organized / quickly updateable by yourself and others and (2) designing the product your users want, rather than what you *think* they want. Systems development is a useful concept not just for computer science projects, but for any type of product development.

For code organization, we use **Application Programming Interfaces** (APIs), which are a set of functions and procedures to allow different portions of code in a big project to communicate with each other. For user-guided product design, we use **iterative design**, which is the process of making many imperfect iterations of a project to prototype to users before trying to make the final product.

10.2 Code Organization

10.2.1 Modularity

Definition 10.2.1. **Modularity** means that code can be separated into many smaller components that are relatively independent.

Modularity has two primary benefits:

- **Teamwork**: It allows different people to work on different portions of the code without interfering with one another. This is how software companies like Google, Amazon, and Microsoft can have tens of thousands of teammembers all working on the same project.
- **Updateability**: It ensures that if one portion of the code needs to be updated or fixed, the rest of the code will not be impacted.

To illustrate this, we will consider modularity's usefulness in cars. Cars are incredibly complicated, but they can be understood by considering them as a collection of interacting components. Each component has its own functionality and relationship to the rest of the car.

A few components of a car include:

- Tires: a round object that can attach to an axle
- Engine: a device that can spin an axle
- Axle: a rod that can be connected in the center to an engine, and at the edges to tires

Figure 10.1: Tires, engine, and axle can be combined to make the core axle-tire-engine unit of a car.

Figure 10.2: A variety of tires are available, and each must simply (1) be built to roll and (2) be connectable to an axle

There are different ways to design each of these. They can each be made from a number of materials, and there are different sizes and varieties of each (e.g. 4 vs 6 cylinder engine, snow vs all-season tires, etc.). Furthermore, any *combination* of these different varieties can be put together: we can change the type of tires we have without thinking about what type of axle or motor is in the car.

We will now look at the benefits of this modularity in car design:

- **Teamwork**: Car companies can divide up labor in multiple teams, and those teams won't interfere with each other as they work. One team can spend months developing the axle: finding the right alloy, shape, and weight to handle the load from the car. Another team can spend months developing the engine: trying out different arrangements of the cylinders or different numbers of cylinders. What's more, car manufacturers almost always outsource the production of their tires to tire manufacturers: they leave the decisions on the type of rubber and treads to a team totally outside of their company. The modularity of the car allows these teams to work totally separately, with the only communication between them the above list: the tire must roll and attach to an axle; the engine must spin the axle; and the axle must connect to the engine and the tires. Once the teams agree on the way they will bolt the systems together, they no longer have to communicate.
- **Updateability**: Modularity allows car enthusiasts to upgrade and customize their vehicles without having to redesign the whole thing. If a vehicle owner wants to replace their V6 with a V8, they can do that as long as the connections between the motor and the rest of the vehicle are the same (i.e. the connection to the axle, to the fuel source, etc). If a vehicle owner wants to replace their all season tires for snow tires in the winter, they can do that as long as the connection with the rest of the vehicle is the same (i.e. the connection to the axle is the same).

10.2.2 APIs

Definition 10.2.2. An **Application Programming Interface**, or API, is a contract between components in a system, expressing what each component can expect from the others.

Building on the previous section's example of a vehicle, we can consider the API between the engine manufacturer and the vehicle manufacturer.

The engine manufacturer can expect the vehicle to

- have an axle
- have a fuel source
- have a cooling source
- have a control system

The vehicle manufacturer the engine to

- attach to an axle
- spin at a given revolution speed

As we saw in the previous section, this type of contract allows division of labor between different teams working on a project to ensure that they can work independently: as the engine manufacturer,

Another API exists between a driver and the manufacturer: the driver can expect the car to have a

- device to turn on/off the car
- device to steer the car
- device to accelerate the car
- device to decelerate the car

The car maker can expect the driver to have

- arms and hands that can turn and push
- feet and legs that can press

Example 10.2.3: Make the API between an ATM and a user.

From the perspective of a user, an ATM's purpose is to take in a credit car and a specified dollar amount from a user, and output the requested amount of money. The user can expect an ATM to

- read their credit card
- specify the card's PIN number
- specify a requested dollar amount
- output money and notify their bank of the transaction

An ATM can expect the user to

- have a credit card
- type with their fingers
- read and respond to prompts on the screen

Note that a problem with the API translates to a problem with the end-product. This API doesn't guarantee that a blind person will be able to use the ATM, for example (they won't be able to read).

Example 10.2.4: Make the API between an ATM and a bank.

From the perspective of a bank, an ATM's purpose is to send in credit card info and the requested dollar amount, and fulfill a user request for money only if the bank authenticates the request.

The bank can expect an ATM to

- accurately send the bank credit card info and a PIN
- accurately send the bank the requested dollar amount
- complete the user's transaction only if the bank sends back a confirmation

The ATM can expect the bank to

- receive credit card info and a PIN
- send back a confirmation or denial

10.3 User-oriented Design

10.3.1 Waterfall vs Iterative Design

Example 10.3.1: Using

- 10 sticks of dry spaghetti
- one foot of string
- one foot of tape

build the highest tower possible in 6 minutes.
Adapted from https://tinkerlab.com/spaghetti-tower-marshmallow-challenge/

This activity generally demonstrates that iteration trumps pre-planning. It's faster to just trying out imperfect designs than to try to wait for a perfect idea. With a 6 minute time limit, iteration tends to work out better than pre-planning.

Definition 10.3.2. Waterfall design is a development process in which each stage of development is finished before the next is started.

The components of waterfall design, adapted from https://airbrake.io/blog/sdlc/waterfall-model, are the following:

1. Requirements: define what the application should do (essentially, write the API between a user and your product)
2. Design: decide what the product will look like based on the requirements, and how it will be implemented
3. Implementation: build the product based on the design
4. Test: ensure the product works as expected
5. Deployment: release the product to the users

In waterfall design, mistakes early in the process can kill you. You might spend a lot of time and money going through the full waterfall and developing a final product and then realize that the requirements were wrong. Going back to a previous example, suppose you had designed an ATM thinking that users would be able to read and later found out that blind users must also be able to access the ATM. You would have to completely redesign the system, perhaps having to put speakers into the ATM so that the prompts can be read aloud to the user.

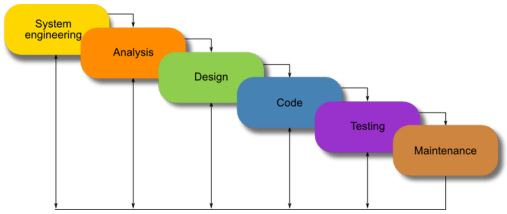


Figure 10.3: Waterfall design, courtesy https://airbrake.io/blog/sdlc/waterfall-model

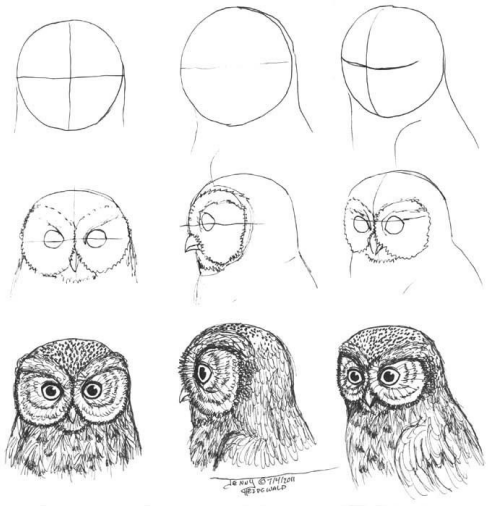


Figure 10.4: Iterative drawing of an owl. https://www.pinterest.com/pin/469289223655955022/

Definition 10.3.3. In **iterative design**, many iterations of a project are built and floated to users with the expectation that the requirements and design may need to be adjusted in further iterations.

Iterative design entails going through the same 4 steps of specifying requirements, drafting a design, implementing, then deploying. However, less time and money is invested into trying to make the first iteration perfect. The first iteration of a project might look horrible, but users will be able to tell you the fundamental flaws (such as missing requirements) before you start implementing a perfect product for the wrong problem.

Iterative design is also used in drawing. As shown in Figure 10.4, professional artists start with a rough sketch of an object before starting to fill in details. The first iteration (top) doesn't look very good, but you might find out that you're missing basic requirements: you might be missing certain body parts, or decide you'd like to add a background or another object. By the second iteration, things look a little better, but you still might find more fundamental errors along the way. By the time you're ready to fill in all of the details in the bottom step, you're confident you're drawing the figure you want to draw.

10.4 Testing

Code rarely works the way we would like it to the first time around.

Definition 10.4.1. A **software bug**, is an error in a computer program which causes an incorrect or unexpected result, or to behave in unintended way (Wikipedia).

Medium estimated that in 2016 approximately a trillion dollars was lost to the US economy due to software bugs. As one tangible example of the cost of bugs, in 1962

NASA's 18 million dollar Mariner 1 had to be self-destructed mid-flight because of a missing hyphen in the control code (https://medium.com/@ryancohane/financial-cost-of-software-bugs-51b4d193f107).

For this reason, developers are always expected to carefully test the code they write to ensure there are no bugs. In addition, even relatively small software firms often have an entire team whose sole job is to test code written by the developers. Their job is to use the product in a variety of potentially unexpected ways. In essence, they try their best to break the code. The product is not ready for the market until the testing team is unable to break it, and all of its behaviors are as expected by the API for the product.

Appendices

Possible appendices topics

- Comparing floating point numbers.
- Type conversion.
- Useful math functions, e.g. `min()`, `log`.