# JAVA:

## An Open Approach with No Objects

**Finger Lakes Community College**

Will McLaughlin
Aaron Sullivan
Dave Ghidiu

Version 1.1
January, 2018

This is the first full edition of the Introduction to Java textbook for Finger Lakes Community College. As such, there may be typos, grammatical mistakes, formatting issues, and erroneous content. If you find any of these mistakes, please email them to dave.ghidiu@flcc.edu - you may get your name credited in the official version that follows this first edition! And it's good karma. So be diligent in reporting your errors.

---

The Computing Sciences department at FLCC is a collaborative team. The primary authors for this book are Will McLaughlin, Aaron Sullivan, and Dave Ghidiu. But this text is a byproduct of excellence in pedagogy as demonstrated by all members in the department.

---

This book is also licensed under the Creative Commons - Attribution 4.0 International (CC-BY-4.0).

This is a human-readable summary of (and not a substitute for) the license. Disclaimer.

You are free to:

- Share — copy and redistribute the material in any medium or format
- Adapt — remix, transform, and build upon the material for any purpose, even commercially.
- The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

---

Errata Help

- The following people have contributed in editing: Shannon Norton, Travis Summerville, and many students in CSC-115 at Finger Lakes Community College.

Content Help

- Malcolm Kotok and Sophie De Arment played a big role in reviewing the flow of the book. A big thanks to John Ghidiu for lending his stellar programming knowledge.

Production Help

- A huge thank you to the FLCC staff who helped this project - namely Katie Nottke whose help and patience in producing this OER was instrumental, and Rachel Fairman whose expertise and speed led to a successful experience!

We dedicate our efforts to April, whose leadership and expertise has had profound impact on our students at FLCC. And she never says “No” when we have a bad idea.

# TABLE OF CONTENTS

# INTRODUCTION

*"If you aren't, at any given time, scandalized by code you wrote five or even three years ago, you're not learning anywhere near enough."*

**- Nick Black -**

## §1.1 Programming

*Computers are ripe to take over the world and you — YOU — may help humanity get swept away by artificially intelligent overlords by learning to code. Or maybe, by learning how to program sequences of commands for computers to follow, you can BE THE HERO that stops the inevitable onset of malevolent robots.*

It's likely your plans are more modest, but we really should recognize just how powerful a force programmable computers are when their potential is unleashed by the people who design their activities, computer programmers.

Applications and programs can exploit a computing device's ability to crunch numbers and follow instructions at breakneck speed, but most people who take advantage of them daily don't understand how much deep thinking goes into making those programs work.

You are about to take a step away from that group of "most people" and towards the much smaller group of people who build, advance, and perfect the programs that others use.

### Computer Programs

So, what is a computer program, really? A **computer program** is a sequence of instructions that defines how a computer can complete a task.

**What kinds of tasks can a computer perform?**

A computer may alert you to an upcoming appointment, tell you the weather, show you important messages from friends or family, and entertain you with videos, games, podcasts, and music - the possibilities seem endless. There are many hundreds of tiny unseen tasks performed to make each of those happen, too!

Are you reading this via a web browser? Anytime you click somewhere, type, roll a mouse wheel, or drag your finger across a touch screen, dozens of little tasks are reacting to those inputs. Numbers as simple as coordinates for the individual dots (pixels) that make up an image on a screen get changed and updated by constantly running tasks.

All of these tasks are managed by programs that a person, a programmer, had a hand in writing. If you've used a smartphone or other multi-purpose computing device like a tablet, laptop, video game console, or desktop computer, you are running programs and their tasks all the time. You may not realize that, in addition to the programs you directly interact with, you likely use hundreds of other complete programs indirectly, because of the complicated underlying **operating system** that supports them.

It's not uncommon for computer programs to leverage millions of lines of code to get their jobs done. For an electronic device with a central processing unit (CPU), chewing through that code isn't usually a problem. After all, its speed has been measured in MIPS; that's millions of instructions per second! However, someone has to write those lines of code, ensure the programs make logical sense and, more critically, it somehow must conform to how the computer reads commands, not necessarily how a human communicates.

## Machine Code

A CPU, or processor, is built to react to electrical inputs representing commands and data in the form of binary numbers. In case you're not familiar, binary numbers use the base 2 number system and consist only of the digits one and zero — not as limiting as it might sound because any number can be represented this way.

One fundamental part of learning to code is learning to become a good problem solver. So, let's go on a short journey to see how modern computers and the Java programming language came to be through decades of solving problems.

*Charles Babbage by Antoine Claudet (1850) (crop)*

In 1837, mathematician Charles Babbage invented a mechanical computer called the analytical engine that was meant to automatically solve mathematical equations. That was decades before electric light bulbs were invented! He was never able to construct it in his lifetime, however.

*Portrait of Ada by British painter Margaret Sarah Carpenter (1836) (crop)*

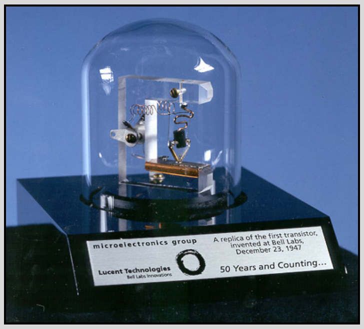
His friend and coworker, Ada Lovelace, envisioned that the analytical engine could potentially do more than mathematical calculations. Among extensive notes she wrote about it, she included what might be the first algorithm for a machine to follow. She may be the very first computer programmer and she accomplished it without an actual working device — it was all theoretical.

It would take the harnessing of electricity for the potential of computers to be fully realized.

**The Transistor**

Modern electronic computers rely on an invention called the transistor which became practical in 1947. For our purposes, just imagine it as a *closed* gate that stops electric current from running through it that can be *opened* with a separate weak electric current.

Those two states, open or closed, can be put to use as opposing states such as on/off, true/false, or one/zero as needed. As transistors were refined into smaller and smaller forms, their compact capability spawned a surge of smaller electronic devices, such as portable radios and calculators.



*A replica of the first working transistor*

The success of electronic devices relying on transistors made using the invention the cheapest and easiest path towards the future of computers. They were used to manage the **storage** and **processing** of binary numbers, where every digit is in the state of either a one or a zero, in nearly all future computing devices.

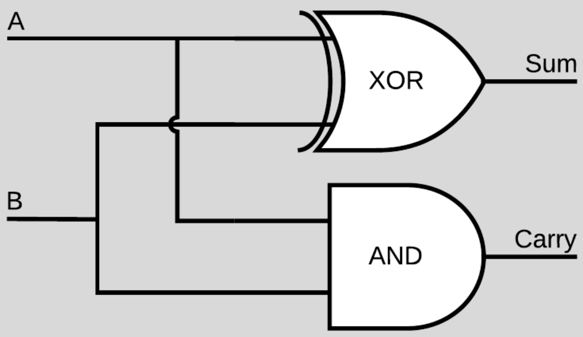
How could simple transistors process numbers? In order to enable computers to perform commands such as adding and comparing numbers, engineers used logical mathematics to arrange transistors. You have probably studied Boolean algebra and seen truth tables with true or false outcomes from operators such as logical AND and logical OR. Combinations of transistors can create gates such as AND, OR, and XOR (eXclusive OR) and those were combined to create pathways that could, for instance, add numbers together.

**Breaking It Down**

An important step in problem solving is to break complex problems into smaller, more manageable problems when possible. When trying to make a computer that is limited to using combinations of opened and closed gates, it might seem complex to add numbers. Our common decimal number system (base 10) is what makes it harder to imagine. How does someone add 9 and 6 with such a limited set of tools?

9 in binary is 1001, and 6 in binary is 110. Does that help?

It turns out that, by using the binary number system (base 2), engineers could boil down the logic to individual digits with values of only a one or a zero. The problem became simpler, as they only needed to produce logic gates that could arrive at a few possibilities: 0 + 0 = 0, 1 + 0 = 1, 0 + 1 = 1, and 1 + 1 = 0 *with a carry*. Then, the set of logic gates could receive each digit from right to left. The result is 1111, which, converted back to decimal, is 15.



The diagram above shows a *half adder*. In order to line these up and carry from one digit to the next, a *full adder* is made using some duplication along with a place for a carry to come *in*.

Users could then feed **input** in the form of coded commands and numbers. The input could be **stored** in temporary registers and memory. The commands could decide which **process** to use, that is, which channels of logically arranged gates the numbers pass through. Lastly, the results could be **output** to the user in some way, such as lighting up bulbs representing binary digits, punching cards, or printing to paper. In short, a computer now had these four fundamental functions: input, store, process, and output.

Though many refinements have developed over time, the foundation was laid. The previous paragraph is an accurate description of how modern computers function today, as well.

Programmers could now code and execute a program! Machine code, however, even now, looks something like this:

Machine code for adding 9 and 6

| Memory Address (hexidecimal) | Opcode (binary number of command) | Operand (value or address) |
|---|---|---|
| 0x10 | 0000 0001 | 0001 0100 |
| 0x11 | 0000 0110 | 0001 0101 |
| 0x12 | 0000 0010 | 0001 0110 |
| 0x13 | 0000 1111 | 0000 0000 |
| 0x14 | 0000 0000 | 0000 1001 |
| 0x15 | 0000 0000 | 0000 0110 |
| 0x16 | 0000 0000 | 0000 0000 |

If that looks daunting to you, as a learner, you'll be happy to know that you don't need to learn machine code to learn Java. However, that *is* what your computing device wants passing through its electric veins: cold ones and zeros. It adds a 9 and 6 that are stored in memory and inserts the result in a third memory location. The stored 9 and 6 values are 1001 and 0110 in binary. Can you find them in the code above?

Let's see how bits of metaphorical warmth were added to code over time.

## Assembly Language

Writing and reading binary digits is perfect for computers but miserable business for most people. We could just slap some human-readable shortcuts over those numbers to represent the operation codes. Maybe then, we could also write a translating program to convert the words back into the equivalent binary machine code that the computer wants. The program could be called an assembler and the language could be called assembly language. Someone beat us to it! That is exactly what the next major step was.

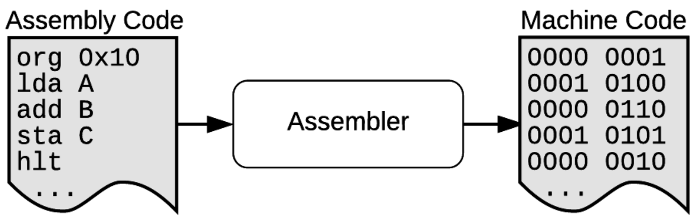
Assembly code | *comments on this side are ignored by the assembler, but can help us!*

```
org 0x10  ;Origin of code will be at memory address 10(hexadecimal)
lda A     ;Load register a with value in memory location labeled A
add B     ;Add value in location labeled B to value in register a
sta C     ;Store resulting value from register a in location C
```

```
      hlt       ;Halt program (end it)
A,    dec 9     ;Label A references the memory with a decimal number 9
B,    dec 6     ;Label B references the memory with a decimal number 6
C,    dec 0     ;Memory at label C will hold sum (0 for now)
      end
```

The descriptions to the right of each line, above, are just comments that the computer ignores, but, if you want to dig in a bit, you could probably get a much better idea of how these instructions work. If you're really ambitious, go ahead and compare this to the machine code, too, as they directly correlate with each other.

This assembly language uses short words, abbreviations and decimal numbers rather than just binary numbers to make it easier for programmers to write and read. However, computer processors do *not* understand assembly language. Before it can be executed, we need to run this code through another program, called an assembler, to translate it into machine code.



Assembly is considered a low-level programming language because it is so "close to the metal"; it is organized exactly how the machine organizes its operations. As a result, programmers who use assembly need to have a precise understanding of how a computer functions to the point of knowing its exact **instruction set**.

Besides the potential monotony of managing details in larger complex programs, code written in assembly can only work on the computer processor architecture it is targeting. When new advances in processors were introduced, they had different instruction sets. Programs coded in assembly would not work with them unless a programmer painstakingly rewrote them.

A great deal of software was successfully created using assembly languages and it is still used, but, over time, a need for something more flexible emerged.

## High-Level Languages

In 1949, when Grace Hopper began working in a team on the first commercial computer in America, the UNIVAC I, she saw the need for a programming language that used english words. She wasn't taken seriously at the time, but, by 1952, she had created the first **compiler**, a program to convert an easier-to-understand language into assembly and machine code. She soon started work on early high-level languages that led to COBOL, a language still used by some companies today!

*Commodore Grace M. Hopper, USN (covered). (1984)*

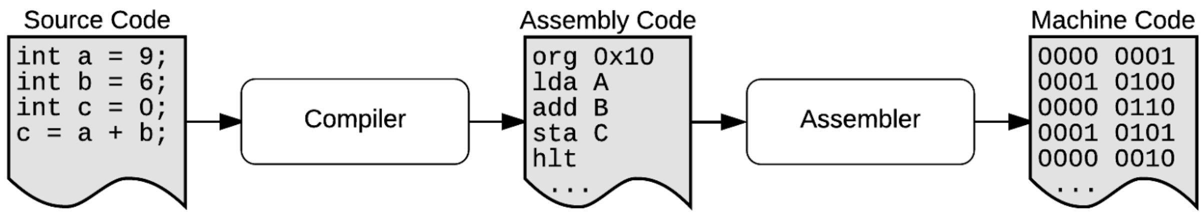
Other high-level languages like C, and eventually Java, added an extra layer of human-readability that brought relief to coders using assembly. Consider the following code that is functionally equivalent to the assembly and machine code listed above:

```
int a = 9;
int b = 6;
int c = 0;
c = a + b;
```

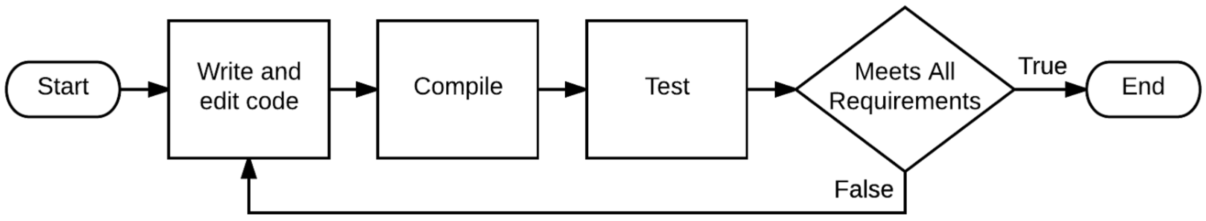
It's short and sweet compared to the others, right? Maybe even partly comprehensible for beginning programmers. Here is the same code with some **comments** added to help explain it.

```
//High-level language like C or Java

int a = 9; //Name an integer memory location "a", give it the value 9
int b = 6;
int c = 0;
c = a + b; //assign to c the result of adding value in a to value in b
```

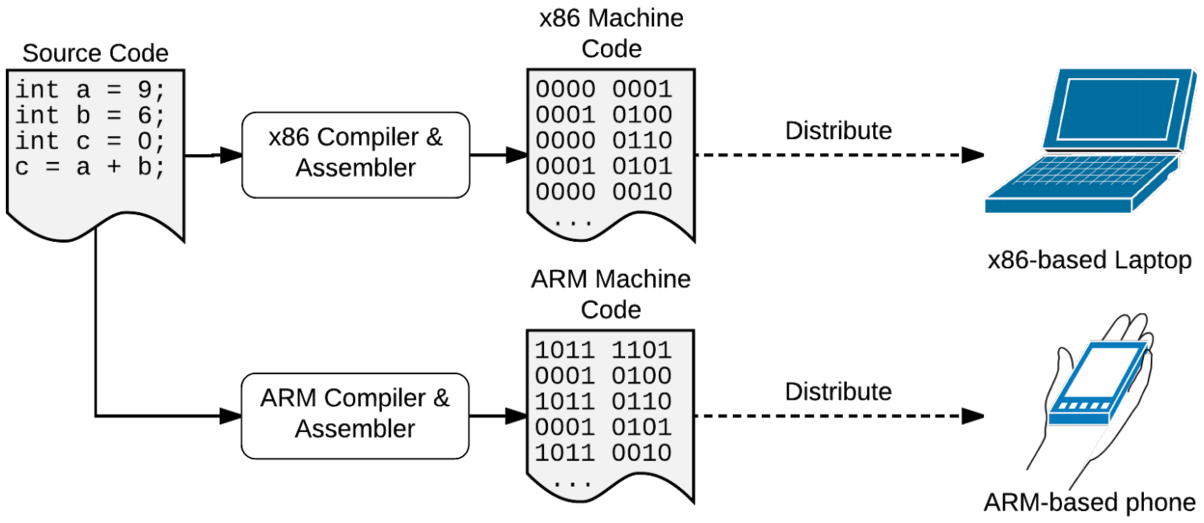
Compare this code example to the assembly and the machine code earlier in this chapter. Which would you rather write? Which would you rather read? There is an extra step needed when using a high-level language like C, because it is no longer so closely related to the hardware architecture. Single line commands in high-level languages are often equivalent to many lines of assembly or machine code. A program, called a compiler, is given high-level code and a target machine so it can create assembly code or machine code for it.



Even though a compiler does complicated work, it is trivial for the programmer to use it. In fact, it can be used over and over in an iterative process to write code, compile it, and test it in a cycle until a program is finished to satisfaction.



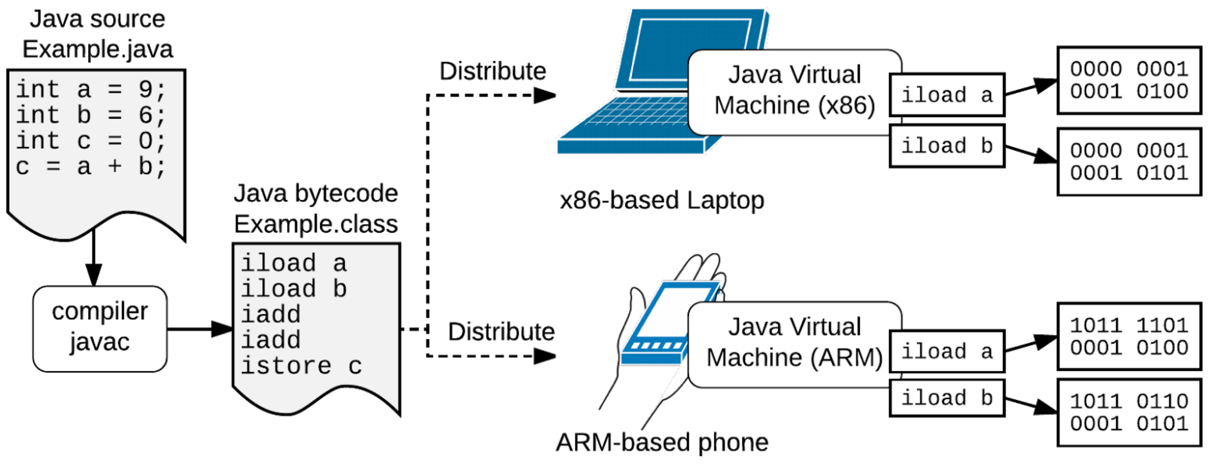
Because a compiler can target different computer architectures, it became possible to write **portable code** using high-level languages. In other words, if a new processor architecture came along, a compiler could be made for it. This way, programmers wouldn't have to rewrite their code. Instead, they could compile the *original source code* to a machine code that matches the new architecture.



What if the need to compile for each new target computer architecture could be taken away as well?

In 1991, James Gosling led a team of developers at Sun Microsystems to begin creating a language for the digital consumer devices market — a market that wasn't ready for it. By 1995, however, this new programming language, Java, found a place to thrive in the quickly growing Internet and World Wide Web. Since then, Java has expanded in popularity as a general-purpose language, but a key factor in its success was an important difference in its compiling process and distribution.

Instead of making programmers compile their programs for each target platform, the developers wanted a Java programmer to "write once, run anywhere". That is, compile it only once and allow end users to run it on all types of computers. The idea was to compile towards a *virtual* machine. The **byte code** created could be **interpreted** by a **Java virtual machine (JVM)** installed on an end user's computer.



**Interpreting and Compiling Just-in-time**

Interpreting is different from compiling in that it converts a small section of code into machine code, then immediately executes the code on the processor before converting the next section. It's similar to the difference between someone sitting down to write a translation from one language to another (compiling), and someone interpreting what someone is saying for a live audience (interpreting). With Java, code is both compiled into bytecode, and, can then be interpreted at runtime.

The problem is that interpreting is slow. Imagine subtitles that were already translated below a video of someone speaking as compared to waiting for a live interpreter to listen, translate, and then communicate it before the original speaker can continue.

To speed up execution of programs, Java also has a Just-In-Time (JIT) compiler that runs along with the JVM to compile sections of a program to native machine code ahead of time, just-in-time. Those sections can then be executed more quickly while the JVM runs them.

The good news for Java programmers is that they rarely need to be concerned with the intricacies of how this process works.

A potential issue for distribution is that an up-to-date JVM must be maintained for any person's type of computer in order for it to run a Java program. A benefit is that code written today could run on any new type of computer that pops up in the future with no burden on the programmer as long as someone writes a JVM for that computer.

**GUI Complications**

The advent of operating systems with graphical user interfaces (GUI) posed another challenge for portable code. This is because the GUI, along with other unique features, is completely different between operating systems. Windows, MacOS, iOS, Android, and the various flavors of Linux all have different techniques to generate the GUI for their programs. Java attempts to help with this by providing GUI code for programmers to use that can run in a similar way on any **platform** that has a JVM.

When it comes to understanding computers and programming, this was a quick dive into the deep end of the pool. You touched the bottom of the pool, the bare metal of computers, the low-level language it understands. It's cold down there, but getting this far means you are back to the surface, your head is out in the warm air. Take a deep breath. Next, we intend to swim on the surface using the high-level language of Java — with floaties if you need them.

## Check Yourself

1. An algorithm is a set of rules to be followed in order to calculate or solve a problem. Answer whether the following statement is **true or false**:

   *A computer program could be described as an algorithm built specifically for a computer to use.*

2. Why don’t most modern programmers create programs using machine code? In other words, what are the downsides of writing machine code?

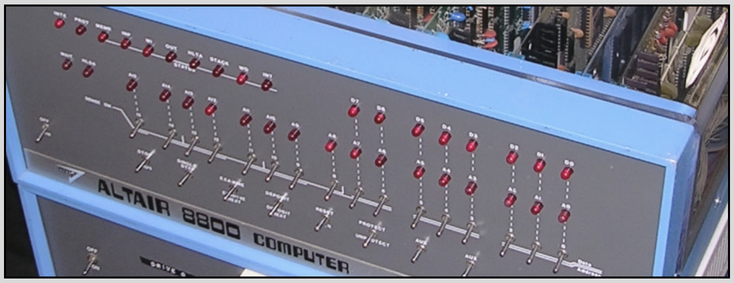
3. How does Java’s middle step of using `javac` to compile to bytecode change the way programmers distribute their programs compared to previous high-level languages?

## §1.2 IDE: Integrated Development Environment

A combination of a compiler and a text editor and other supporting tools make up an **integrated development environment (IDE)**. Modern programmers benefit from decades of helpful improvements their workflow. Today, as you rest your fingers on a keyboard to type code, chances are that you will have a program giving active feedback when you begin typing. That's right, a program working with you to help you write a program!

### Before the IDE

Early computers relied on mechanical interfaces for entering programs. Imagine that someone flips physical metal switches to turn LED lights on and off to represent binary data and a memory location. Then, that person flips another physical switch to set that one code entry into memory: machine code entered by manually flipped switches. Faster and reusable methods such as punch cards were developed but were still labor intensive.

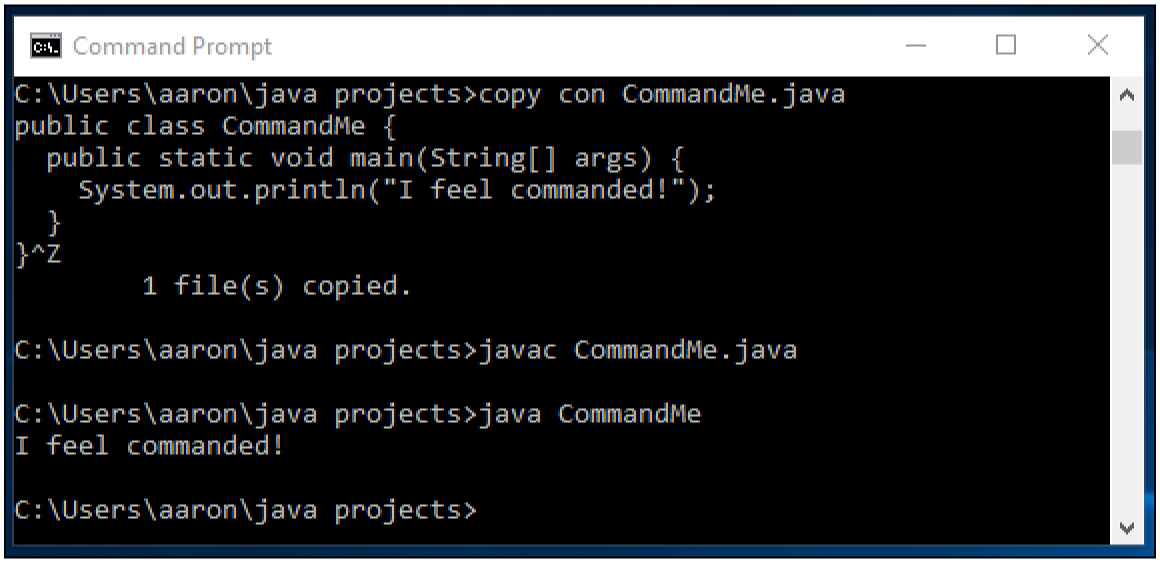


*User interface for the first personal computer, the Altair 8800*

The primary method for computer programming is with a keyboard for typing code and a screen for viewing code. Before point-and-click and touch interfaces were commonly used, text-only command line interfaces were still a huge step above mechanical input. Using simple text editors and separate compilers, programmers had all the fundamentals to support coding with even high-level languages. Using a command line interface is still preferred by some computer users, today.

You can experience this using a modern computer by running the Command Prompt program in Microsoft Windows or opening Terminal in MacOS, or a Linux-based operating system.

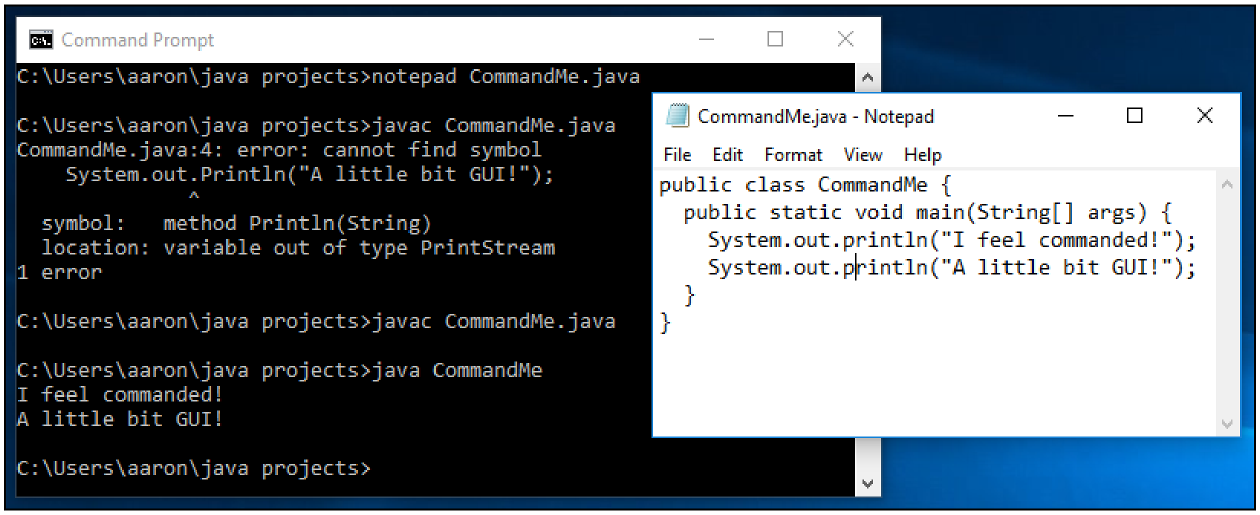
After you install the Java Development Kit (JDK) on a computer, you can invoke the **`javac`** program to compile a text file with Java code in it. You can then run the bytecode file that is made in the process by using the program named **`java`** to run it with the JRE.



```
C:\Users\aaron\java projects>copy con CommandMe.java
public class CommandMe {
  public static void main(String[] args) {
    System.out.println("I feel commanded!");
  }
}^Z
        1 file(s) copied.

C:\Users\aaron\java projects>javac CommandMe.java

C:\Users\aaron\java projects>java CommandMe
I feel commanded!

C:\Users\aaron\java projects>
```

*Command Prompt*

It’s possible to code using basic GUI text editors like Notepad or Notepad++ on Windows in combination with the standard java command line programs javac and java.



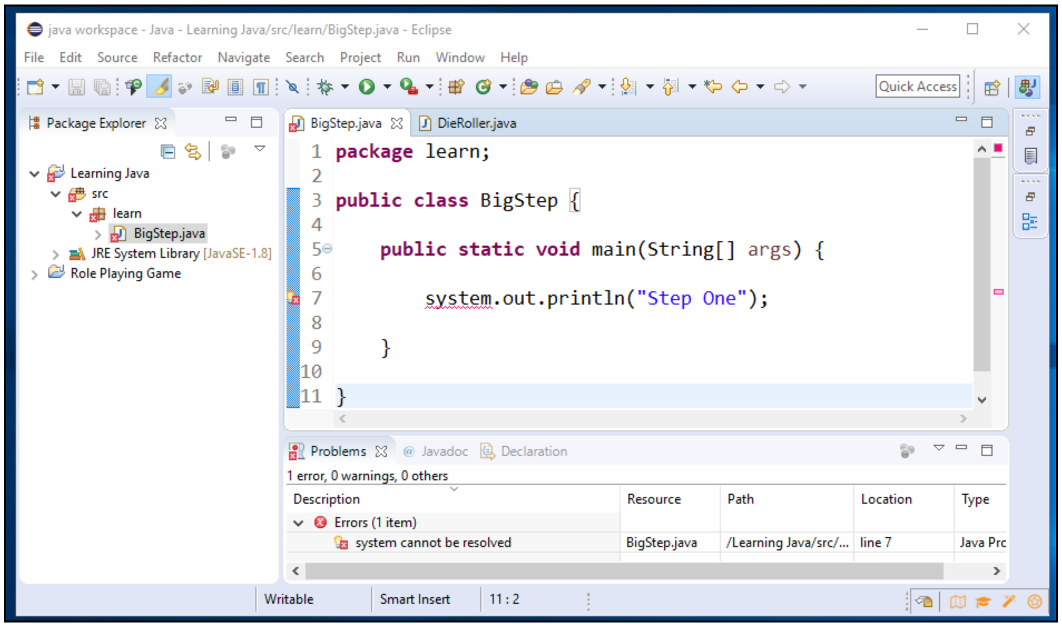
```
C:\Users\aaron\java projects>notepad CommandMe.java

C:\Users\aaron\java projects>javac CommandMe.java
CommandMe.java:4: error: cannot find symbol
    System.out.Println("A little bit GUI!");
              ^
  symbol:   method Println(String)
  location: variable out of type PrintStream
1 error

C:\Users\aaron\java projects>javac CommandMe.java

C:\Users\aaron\java projects>java CommandMe
I feel commanded!
A little bit GUI!

C:\Users\aaron\java projects>
```

```
public class CommandMe {
  public static void main(String[] args) {
    System.out.println("I feel commanded!");
    System.out.println("A little bit GUI!");
  }
}
```

*Command Prompt and Notepad (Can you follow how an error was fixed?)*

## The Modern IDE

With a modern IDE like Eclipse, Netbeans, or JGrasp, most programming tasks can be completed from within that one program. This integration streamlines programming, giving immediate feedback about errors as you type, or even suggesting shortcuts to help type parts of code for you!

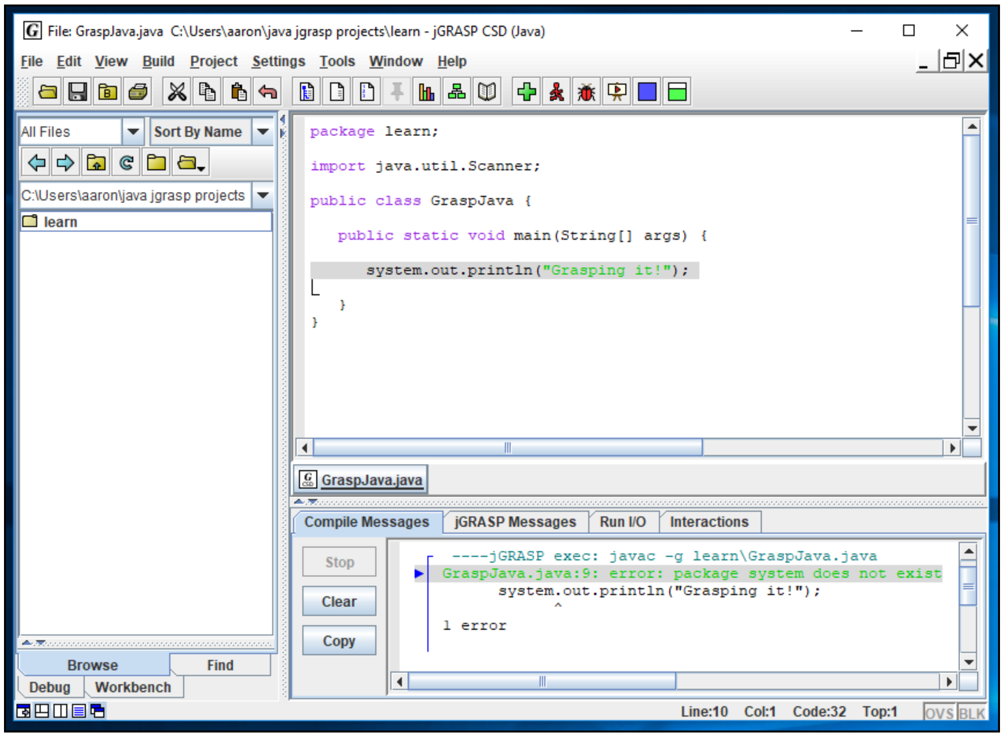

java workspace - Java - Learning Java/src/learn/BigStep.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Package Explorer
Learning Java
src
learn
BigStep.java
JRE System Library [JavaSE-1.8]
Role Playing Game
BigStep.java
DieRoller.java
Problems
Javadoc
Declaration
Writable
Smart Insert
11 : 2

```java
package learn;

public class BigStep {

    public static void main(String[] args) {

        system.out.println("Step One");

    }

}
```

1 error, 0 warnings, 0 others

| Description | Resource | Path | Location | Type |
|---|---|---|---|---|
| Errors (1 item) | | | | |
| system cannot be resolved | BigStep.java | /Learning Java/src/... | line 7 | Java Prc |

*Eclipse IDE underlining a syntax error with a red, jagged line*

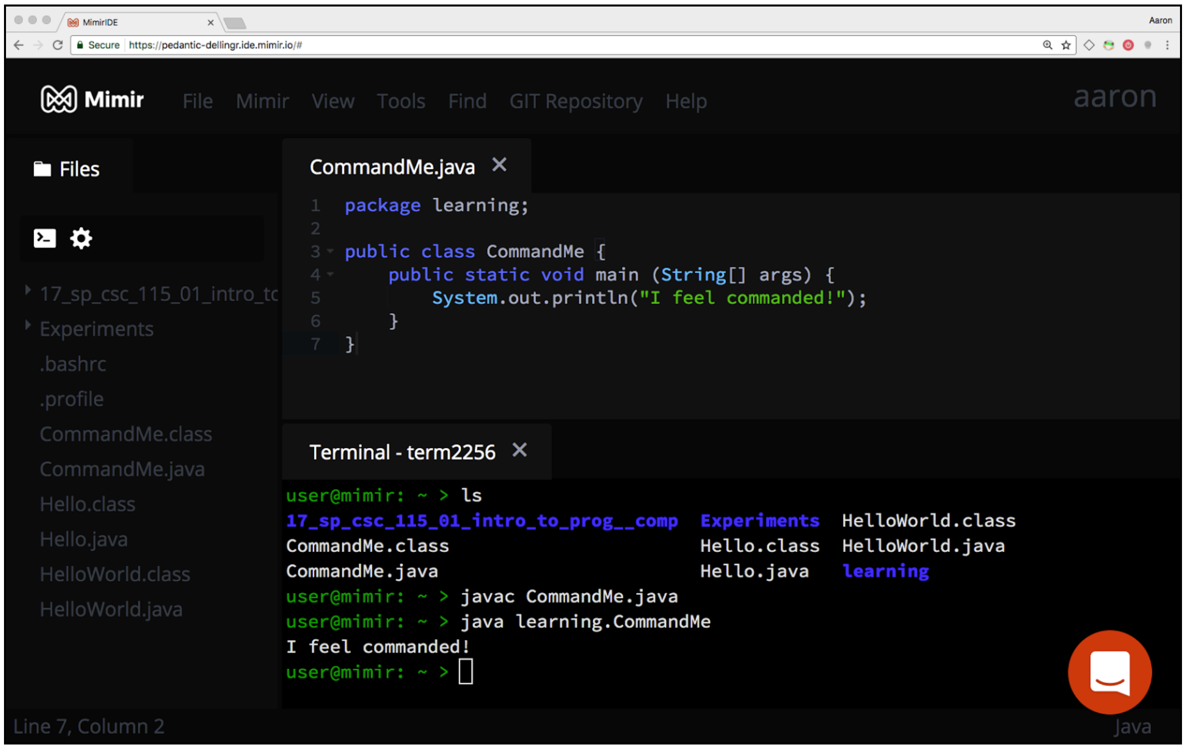

File: GraspJava.java C:\Users\aaron\java jgrasp projects\learn - jGRASP CSD (Java)
File Edit View Build Project Settings Tools Window Help
All Files
Sort By Name
C:\Users\aaron\java jgrasp projects
learn
Browse
Find
Debug
Workbench
GraspJava.java
Compile Messages
jGRASP Messages
Run I/O
Interactions
Stop
Clear
Copy
Line:10 Col:1 Code:32 Top:1
OVS BLK

```java
package learn;

import java.util.Scanner;

public class GraspJava {

   public static void main(String[] args) {

      system.out.println("Grasping it!");

   }
}
```

```
 ----jGRASP exec: javac -g learn\GraspJava.java
GraspJava.java:9: error: package system does not exist
      system.out.println("Grasping it!");
            ^
1 error
```

*The JGrasp IDE explaining an error in the Compile Messages pane*

Online services can provide IDEs that are Web accessible using a web browser such as Google's Chrome. Mimir offers just such an IDE, called MimirIDE, to complement its other educational tools. At the Mimir web site, a learner can read an assignment, code a solution, fix errors, submit the solution, and even receive feedback.



```
package learning;

public class CommandMe {
    public static void main (String[] args) {
        System.out.println("I feel commanded!");
    }
}
```

```
user@mimir: ~ > ls
17_sp_csc_115_01_intro_to_prog__comp  Experiments  HelloWorld.class
CommandMe.class                       Hello.class  HelloWorld.java
CommandMe.java                        Hello.java   learning
user@mimir: ~ > javac CommandMe.java
user@mimir: ~ > java learning.CommandMe
I feel commanded!
user@mimir: ~ >
```

*The Mimir IDE running in a web browser*

### Common Features

As mentioned above, modern IDEs attempt to help the programmer type code. They do this by constantly studying your code and generating an internal model. From that model it can:

- colorize parts of your code to differentiate them.
- add **linting** to your code, which means to mark sections that don't follow the syntax rules of the language or to warn about potential issues.
- automatically divide your code into sections that you can hide and show, a feature called **code collapsing**.

Have you ever noticed how typing in search terms in some web pages or applications shows a list of possible choices to finish the phrase? In a similar way, some IDEs make smart suggestions about what you might want to type next. Variations of this capability have feature names such as **code completion**, **code assist**, or **autocomplete**.

**Taking Advantage**

Did you ever see an expert use a tool in ways you never imagined? As you begin to invest time in coding, you'll want to be an expert at using your primary tool, an IDE. Learn how to activate its features in order to type code faster and, better yet, accelerate the understanding, editing, and fixing of code.

Different IDEs have different methods but you'll find similarities between them. Colorizing is automatic, but most IDEs offer options to change what colors are used. For code completion, you may tap the down arrow key to select the code you want from a list and press enter or tab to have it typed for you. Code collapsing can often be activated with a small plus or triangle icon next to a section of code.

**Keystroke shortcuts**, also called *hot keys* or *keyboard shortcuts*, can help you access hundreds of commands in an IDE by simply pressing a combination of keys on your keyboard. For instance, holding the control key (ctrl) or command key and tapping the forward slash key (/) can turn a line of code into a comment for the compiler to ignore. You may see this listed as ctrl-/.

**File Management**

How does your workspace look? Code is stored in text files that a programmer writes, and complex programs require many files organized in a nested hierarchy of folders. Fortunately, most IDEs assist with organizing files.

For instance, you will soon see how programs are structured into packages which require storing files in folders with the package name. Without an IDE you may find yourself manually creating folders and moving files into them, but an IDE can do all of that for you.

Depending on the IDE, you may establish a folder where all of your projects are collected, a **workspace**. After that, creating a new project automatically creates new folders and files so that you stay organized. Most IDEs also include a file or project explorer section in the interface so that you can easily see changes that are taking place without concern for the rest of your computer's file system.

Even though an IDE helps, you will often need to know where on disk your workspace is stored and need to use your operating system's file management tools. After all, you may want to create a backup of your work to an external storage location like a flash drive, or to a folder with synced cloud storage. Depending on the IDE you use, you may need to copy the entire workspace folder. Copying only a single code file to a new location may separate it from other files it depends on and cause it to fail. So, be sure to take the time necessary to understand how your files are organized by your IDE.

TIP: If you use a basic text editor and command line compiler, rather than an IDE, you may make a change and forget to save the file before compiling. You may conclude that a fix didn't work when, in reality, the compiler is still working with the unchanged version! Be sure to save your files after you edit your code.

Larger programming projects require teams of people to work on code together and manage revisions and changes on a daily basis. There are tools called **version control systems** which allow a programmer to download a project from a shared online location, make improvements, test them locally, and then push the changes back to others working on the project. Popular version control systems include Git, Mercurial and Subversion. Some IDEs can integrate with these tools so that a programmer can contribute to distributed projects without using other applications.

An IDE offers many aids to you, as a programmer, if you learn to use it. The core work of computer programming happens in your mind, though, and you must learn a programming language like Java to express that work. If IDEs can be the floaties in our swimming metaphor, our next step is to start swimming.

## Check Yourself

1. You need an integrated development environment (IDE) to write, compile, and debug code. True or False?

2. Name at least three helpful features of a typical IDE.

3. When copying a program to continue work at a different location, say when working on it at home after starting work in a lab, why is copying just a .java file sometimes a problem?

## §1.3 A Program: Structured Code

### Hello, World!

It’s time to work on your first Java program, and, following tradition, you can start with a hello world program. The only requirement for this program is to display “Hello, World!” on the screen.

Create a file named First.java and type in the code below. It’s important to give the file that exact name with the capital F, even. Be very careful to make an exact copy when you type the code — and you should *type it*, don’t copy and paste! You only become familiar with a new skill by practicing, so my advice is to type it in line by line, and start training yourself.

```
public class First {
  public static void main(String[] args) {
    // display some output
    System.out.print("Hello, World!");
  }
}
```

When you compile and run this Java program it displays:

```
Hello, World!
```

Be sure to try it yourself. You may find that it won’t run or that portions of your code have colorful jagged underlines. If so, compare the code above to yours. Did you capitalize the same letters? Did you use the right surrounding symbols? There are square brackets, parentheses, and curly brackets (braces), and they are easy to misread or mistype. Keep at it until you see the output showing “Hello, World!”. If you get stuck, get help.

Now that it works, take a minute to analyze the code. (Go ahead, I’ll wait.)

If you haven’t used other programming languages before, there’s a good chance it doesn’t make much sense to you. All of the code is important, but only a small section is directly concerned with displaying “Hello, World!”. What does it all mean?

#### Comments

There is a nice plain-language explanation in the middle:

```
// display some output
```

That explanation is a **comment.** When you compile and run the program, a comment does... *nothing.* When the compiler reaches `//` it ignores everything after it up to the end of the line. Why include it, then? A comment is written to help a person understand other nearby code. That person could be another programmer studying your code, or it could be *future you*! Either person will appreciate the help. Go ahead and ask future you, later. There's more about comments in the next chapter, too.

## Tokens and Statements

The next line performs the action we intended:

```
System.out.print("Hello, World!");
```

That is a **statement**, which is a command that a programmer creates to help accomplish a task. You can think of that entire statement's purpose as "display a message on the screen". This type of statement is called a **print statement** or an **output statement**. You can learn about other types of statements like input statements and assignment statements in later chapters.

A statement is made of smaller parts, called **tokens**.

Think of tokens as the smallest unit in programming. They are broken into five types: keywords, identifiers, operators, separators, and literals.

Some example tokens:

- The semicolon (**;**) at the end is a separator token that signifies the end of the statement.
- `"Hello, World!"` is a literal token, called a **string literal**. Strings are covered in more detail, later.
- An **identifier** token, `print`, is the name of the method we want to call into action.

## Methods

It turns out that there's a whole series of unseen statements needed to get that message to appear on the screen. Again, `print` is the identifier for a particular method. A **method** is a named sequence of statements. Curiously, you don't need to know what the sequence of statements is, or how they work, you only need to call it into action (invoke it), and it does the work for you. The program does this by temporarily jumping to the statements in the print method which are defined somewhere else.

The example also has the code to define a method, complete with its sequence of statements:

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

The name of the method is **main,** and its header is defined very specifically so that, when the program runs, it is recognized as the first method to invoke. Any statements after the first curly bracket (brace) are executed one after the other in the body of the method. The program will stop running when it reaches the end of the main method. The end of the method definition is marked by a closing brace, the second to the last brace in the larger code example, to be exact.

## Classes

Many methods can be defined, and they are always defined inside of a **class**. Even though classes are capable of much more, for now, you can think of a class as a collection of methods.

```
public class First {
    ...
}
```

Notice how, like methods, the class definition has a header that includes an identifier, `First`, and also opening and closing braces ( **{** and **}** ) to surround what is inside the class, the body. One notable difference is that the identifier for a class is capitalized, but the identifier for a method is not (compare `First` to `print`).

Classes must be defined in files with the exact same name as the class. That’s why this example program had to be typed into a file named `First.java`. (The `.java` part is called a filename extension and helps programs understand what type of information is in it.)

## Packages

In a large project, you may decide to name your classes with the same identifier as classes written by another programmer. This could cause a conflict! For instance, the other programmer’s code could also have class named `First`. How would you or the compiler tell which `First` class is which? They can be grouped into a different **package**.

We could add a line like this to the top of the project example:

```
package edu.flcc.example;
```

Now, you can use the **fully qualified name** of your class when you need to differentiate:

```
edu.flcc.example.First
```

Depending on the tools you are using, you or an instructor may find it unnecessarily complicated to organize your many small learning projects using packages. This is partly because the file hierarchy must often be parallel to the package name. For instance, the `First.java` file may have to be located in an `example` folder, which is inside an `flcc` folder, which is inside an `edu` folder.

Either way, you *will* eventually want to take advantage of the many predefined classes in the **Java Class Library**. Yes, predefined means someone else did the work for you! They are organized in packages, so you can **import** them into your code using their fully qualified name in a statement like this:

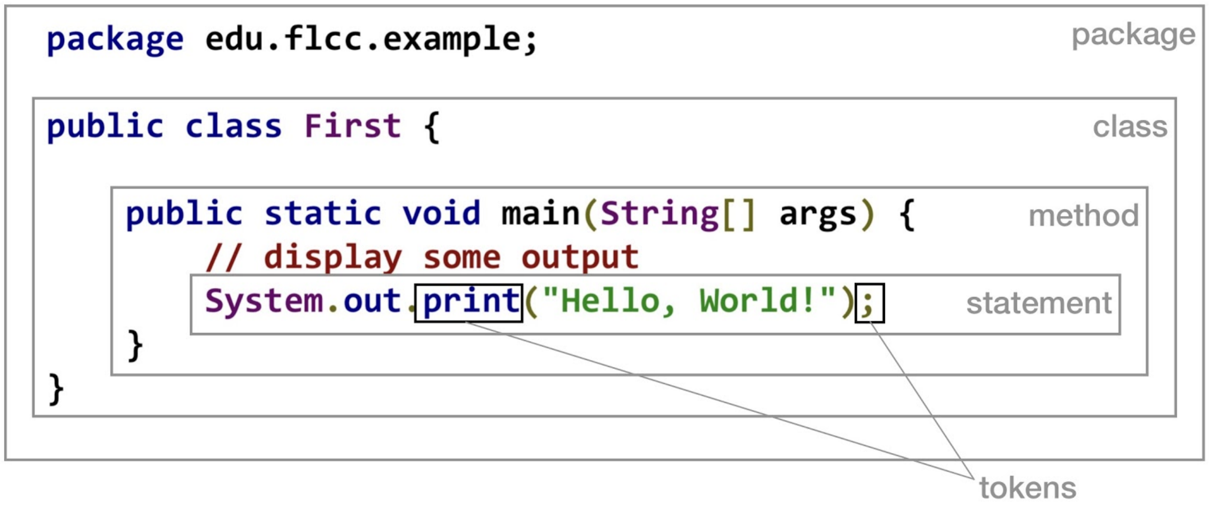
```
import java.util.Scanner;
```

The current steward of the Java language, Oracle, has extensive documentation for when you need further details, including this tutorial about packages: https://docs.oracle.com/javase/tutorial/java/package/index.html

For a program that performs a very limited action, that was an extensive analysis. All this just to display a single short message to the screen! The complexity is needed because the same structure and rules of the language must scale from something as simple as "Hello, World!" up to programs like LibreOffice or Minecraft that have thousands of lines of code.

## Program Structure

The previous section discussed the different sections of the "Hello World" example program, and it outlines almost the entire structure of any Java program:


```
package edu.flcc.example;

public class First {

    public static void main(String[] args) {
        // display some output
        System.out.print("Hello, World!");
    }
}
```

Packages are collections of classes.
- Classes contains method definitions.
  - Methods contain statements.
    - Statements are made of tokens and sometimes contain expressions.
      - Expressions are also made of tokens.
        - Tokens are the smallest elements of a program.

## Programmer Jargon

It can be a challenge to remember all the jargon associated with any field that is new to you, but it will benefit you greatly. As you might expect, learning the specific terms that programmers commonly use empowers you to learn at a faster rate. In addition, it enables your participation in the worldwide community of programmers that discuss their knowledge online.

In other words, it's better to align your vocabulary with how everyone else talks about programming, so you can help them, and they can help you!

Below is a list of some terms that may be new to you in the context of computer programming. Do you know what they mean?

| | | | |
|---|---|---|---|
| package | token | IDE | code |
| class | high-level | assembler | comment |
| method | low-level | compiler | platform |
| statement | program | interpreter | language |

## Check Yourself

1. Why is it counterproductive for someone learning to code to only copy and paste examples to test them. In other words, why should a learner hand-type code examples?

2. What is the name for messages typed into code that the compiler ignores, yet are helpful to programmers reading the code?

3. What is the name for the smallest unit of programming that the compiler processes?

4. What characters (symbols) do you type to create “braces”? What are they for?

5. Name all methods shown in the example code from this section.

## §1.4 Coding Conventions

It turns out that, even with all the specific syntax a programmer needs to learn for any given language, a tremendous amount of flexibility remains. This means your code can have a style and flair of its own!

Imagine a pile of papers with different solutions to the same problem in code. Instructors can often tell who wrote many of them just by comparing them. This is possible not only because there are many ways to logically find solutions to problems but also because of that flexibility in how code is written.

Look at this code, though:

```
                                        package
edu.flcc                               .example
       ;                                 public
   class
   First                                      {
  public static                             void
    main
(String[                                      ]
                                           args
     ){  System.out.print      ("Hello, World!"
     )
     ;                                       }}
```

Even early in your learning, that should look terrifying. You might be surprised that, to the compiler, the above code is functionally identical to the earlier “Hello World” example code. It will compile and run without issues! The Java language specification allows for formatting like this, but coding conventions keep us from writing code that is so hard to read.

Code that is hard to read but works may *seem* satisfactory, but it will let key people down. For instance, other programmers that must maintain and improve your code will have a harder time doing so. This is especially true as you work on large projects with many contributors. Another key person it may bother is future you. As you continue learning, you’ll want to return to code you’ve written to remind yourself how you solved problems and to build on your own knowledge. Make your code as clear and readable as possible, and future you might even thank you.

**Coding Standards**

The organization where you are learning, or where you work, may have very specifically outlined conventions, called **coding standards**. Coding standards have more weight behind them because you’ll be held directly accountable more immediately than when you are learning on your own or coding only for yourself.

This might seem harsh, but most organizations of programmers have already learned the hard lessons of maintaining confusing or poorly written code — time and/or money has been wasted — and standards have already proven themselves.

Hopefully, you've already decided that following coding conventions is inherently beneficial to yourself and others. That way, when coding in a larger organization, it will be natural to adjust to what will help the group.

Because you are just getting started, it wouldn't be helpful to outline every single coding convention. However, a list of some general coding conventions to deal with formatting and laying out your code is a good place to start.

### Keep Each Line Short

Standards between organizations vary, but each line of code should stay below 80 to 120 columns of **characters —** characters are symbols, like numbers and letters. Consider how the example below won't fit on this page. Even when using computers to view code, it's time-intensive to scroll horizontally back and forth or zoom, rather than fit code in the immediate view of the reader.

```
System.out.print("Hello, World! I'm happy to be here for as long as
```

You can break up long statements by carefully choosing where to use line breaks — press enter or return — and indenting the line in a clear way. The following example demonstrates how the long statement now fits on two lines and can still fit within an 80-column restriction.

```
System.out.print("Hello, World! I'm happy to be here for as long as "
      + "this program runs, so forgive my rambling. See, the longer "
      + "this string is, the l-");
```

### Only Break Up a Line If Necessary

It's possible to be too excited about keeping your lines short. Avoid trying so hard that you make it less readable as a result. Forcing the user to scroll a page of code vertically is also time-consuming.

```
//Poorly formatted code
System.out.print
("Hello,"
+"again!");

//Same result, but faster and easier to read
System.out.print("Hello, again!");
```

### Use Blank Lines to Divide Logically

Like writing paragraphs to help a reader know when the writer is transitioning to new thoughts, programmers should divide up sequences of code that are logically related. When there is no other division being used, such as braces, just add a blank line.

The code below is all print statements, but notice that the blank line marks a transition between welcoming the user and asking for input.

```
System.out.println("Hello, World!");
System.out.println("It's good to be back.");

System.out.print("Could you please tell me your social security number?");
```

Comments added above sections of related code can also help make the code more readable. When learning, extensive commenting can help you review what you've done before, but among some experts, commenting everything gives way to writing code that is so clear, it may not need a comment to be understood!

For now, however, adding comments like below will also help you decide where to divide up your code with blank lines.

```
//Output a welcome message to help the user feel comfortable
System.out.println("Hello, World!");
System.out.println("It's good to be back.");

//Prompt the user to input sensitive information
System.out.print("Could you please tell me your social security number?");
```

**Indent Code that is "Inside"**

When code is inside another code structure, indent it. For instance, from the hello world example, the idea that the print statement is inside the method, and the method is inside the class, is made clearer by the indentations.

```java
//Poorly formatted code
public class First {
public static void main(String[] args) {
//display some output
System.out.print("Hello, World!");
}
}

//formatted for clarity according to coding conventions
public class First {
  public static void main(String[] args) {
    //display some output
      System.out.print("Hello, World!");
  }
}
```

In later chapters, some coding conventions may be highlighted directly, while others will be implied, when new concepts are introduced. Either way, be sure to pay attention to coding conventions, so you can help make a complex field more clear to programmers around you.

## Check Yourself

1. Because of the syntax rules of a language, all solutions to the same problem look and function the same way, true or false?

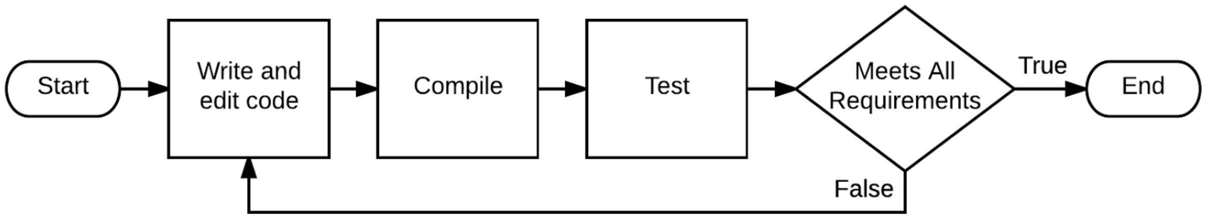
2. Coding conventions say that you should include as many comments as possible, true or false?

3. Why is code indented?

## §1.5 Experiment and Iterate

### Iterate

As you work with code, either to program a solution, fix problems, or learn by experimenting, you can use a repeating process like this flowchart shows:



Programmers often call working through a process like this iterating. To **iterate** just means to perform a task repeatedly. In the chart above, a sequence of “write and edit code”, “compile”, and “test” is one **iteration**. There is a decision at the end that checks if all requirements have been met. If it’s true that they have been met, then you’re done. If it’s false, something isn’t right, and it’s time to iterate!

**If At First You Don’t Succeed...**

Be prepared to do many, many iterations of this kind of process as you learn to program. It’s only natural to make many attempts while learning something so complex, but it’s also part of the job! Professionals use this repetitive process constantly.

You may find that failing to find a solution and getting stuck can be very emotionally frustrating, at first. Programmers learn to take it in stride, eventually, though. Many have found that experience can help someone overcome that frustration. Like them, you can learn to anticipate the sweet gratification that comes from creating great code — code that works!

### Experiment

Now, it’s time to experiment just a bit with the hello world example program. It shouldn’t be too hard to add an additional output statement. Let’s go to a phrase that predates the traditional “Hello,  World!”. Let’s pretend the computer is excited by your newfound programming skills and make it exclaim, “MY HUMAN UNDERSTANDS ME”.

I’m going to show the code to you below, but I encourage you to try changing and testing your code without it. Turn to your screen, and try a few things before looking ahead for the solution. Go ahead and hide this page, now. I'll wait.

**Undo (ctrl-z or cmd-z)**

Are you hesitant to try without a reference? There is always undo — have I mentioned undo? The beauty of experimenting on computers is that most programs let you undo what you've done by holding ctrl and tapping z (ctrl-z or cmd-z). Many people know about it, but don't always take advantage. You'll learn better if you confidently experiment. Try it. If it doesn't work, undo, and try again.f

Maybe you had an unexpected result from a solution like this:

```
public class First {
  public static void main(String[] args) {
    //display some output
      System.out.print("Hello, World!");
      System.out.print("MY HUMAN UNDERSTANDS ME");
  }
}
```

OUTPUT:

```
Hello, World!MY HUMAN UNDERSTANDS ME
```

That looks jammed up. I don't want the second sentence pressed up against the first. Time to experiment. Try adding the space character inside the quotes next to the exclamation point (!) or the "M" in my. It's an easy symbol to forget about because, well, it's invisible. If you are still having trouble, my recommendation is to move the insertion point to the right side of the exclamation point and hit that spacebar with purpose. Run it again.

OUTPUT:

```
Hello, World! MY HUMAN UNDERSTANDS ME
```

This is better, but I'd prefer it on two separate lines. I'm betting you expected that, anyway. I mean, you wrote the two print statements on separate lines, right? The issue is that the method, `print()`, outputs every character exactly as you specify between those quotation marks and nothing more. You need that invisible newline character that is inserted when someone presses the enter or return key.

There is more than one way to add the newline, but there is a method like `print()` that does it for you. It uses an abbreviation for “line” at the end of “print” to form the identifier, “println”. Most read it as, “print line” and it means to print and then add a newline character at the end so further printing will happen below it.

Try to test out the new method quick before looking below!

```
public class First {
  public static void main(String[] args) {
    //display some output
    System.out.println("Hello, World!");
    System.out.println("MY HUMAN UNDERSTANDS ME");
  }
}
```

OUTPUT:

```

Hello, World!
MY HUMAN UNDERSTANDS ME
```

That’s what I’m talking about! So, println() added a newline after the exclamation for you.

Maybe you noticed that the example uses println() for the second print statement, as well. I did this because some IDEs will add messages to the end of your output, and using a println() on your last print statement keeps them from getting mashed together.

## Learn by Doing

I hope you followed along and tested the code above. Just reading about something or even watching someone do something doesn’t help as much as doing it yourself. Trust me, I’ve watched people do parkour and freerunning for years, and I still trip on the front step at home.

Many new concepts in this book will be introduced with code examples that are experimented with and iterated on. The hope is that you’ll participate, and it will begin to feel natural for you. And remember, even professionals fail often with code before they get a breakthrough.

## Check Yourself

1. What is an iteration?

2. What is the functional difference between `print()` and `println()`?

3. What are the three steps a programmer iterates through when coding?

# BASIC JAVA

*"Simple things should be simple, complex things should be possible."*

**- Alan Kay -**

*Now that you've studied a very basic Java program, let's get started on the key building blocks of Java. The goal of this chapter is to give you the basic understanding of the programming process and to keep your development and code organized. While this chapter scratches the surface of many Java and software development concepts, it establishes a foundation to prepare you for a deeper dive in later chapters.*

*Coding is just a small part of the process. Unfortunately, many focus on just the typing of code and inadvertently make programming more difficult. Elegant software solutions are produced when you do more planning and testing before you even begin to code - just like writing a paper, landscaping, or building anything physical.*

## §2.1 Comments

Comments are used by programmers in almost every programming language. *Comments* are not processed by the compiler, so code that is "commented out" is a good place to put things that you don't want the computer to run. This could include comments that help people who read the code understand what the code is supposed to do. Sometimes, it is handy to comment out code that works perfectly fine, but just gets in the way when we are testing other parts of the code. Or sometimes, we comment out code that is "in progress" and isn't fully developed yet.

Most IDEs will change the color of text that is commented out. The color of comments is not really important, although typically the color an IDE reserves for comments will not be used anywhere else.

### Types of Comments

There are three basic ways to comment:

#### The Single Line Comment   //

We use the double slash when we want to comment out only one line (or part of a line). As soon as the computer encounters a double slash, it does not read anything else on that line.

```
System.out.println("This code runs.");
// System.out.println("This is a test.");
System.out.println("This code executes, too.");
```

The code above displays:

OUTPUT:

```
This code runs.
This code executes, too.
```

Occasionally, we will see code with comments at the end of the line. In the case below, the developer has added a comment that describes what the line of code does (spoiler alert, the code will output `3.141592653589793`).

```
System.out.println(Math.PI); // This will display Pi
```

Using the two slashes as comments on the same line - but after the code - is a great way to leave remarks and comments about what that line of code is expected to do.

```
System.out.println(3 + 6); // 9
```

We'll also see it in textbooks and online as a place where the author of the code will "comment out" what the answer to a problem is.

**The Block Comment** `/* */`

When blocking out many lines of code at one time, it's probably easier to use the block comment, also called the multi-line comment. Think of it as a sandwich - the first part, `/*`, is the top piece of bread, and the second part, `*/`, is the bottom of the sandwich. Anything in between is commented out. Stylistically, people like to add an `*` to each line in the comments, but that's not necessary.

```
/* Written by Ferris Bueller
 *
 * This program will compute the interest on a bank loan
 * when given the loan amount, interest rate, and time
 *
 */
```

This code is equivalent to the following code - they both do the same thing - that is, comment out whatever is in between the slices of "codeBread."

```
/* Written by Ferris Bueller

 This program will compute the interest on a bank loan
 when given the loan amount, interest rate, and time

 */
```

Oftentimes, when you are working on code and want to comment out a big chunk of it, it's pretty quick if you jam a `/*` before the code you want to comment and a `*/` after the code you want to comment.

**The JavaDoc Comment** `/** */`

Also known as "Documentation Comments", here is a third way to comment. If we were to make a super sweet program (or class) that required a “user manual”, then we should get used to the JavaDoc comments, also known as Documentation Comments. When used properly, the user manual will be generated for you automatically. If you are sitting there wondering what a JavaDoc looks like, check out the JavaDoc for the `String` class.

You will revisit Documentation Comments when you start developing your own methods with parameters and return types - we don’t use them much in this class

## Reasons to Comment

Many programmers use comments at the top of their programs to put their name and a brief description of what the program does.

```
// Created by Bart Simpson
// This program will output some letters on the screen

public class HelloWorld {
  public static void main (String args[]) {
    System.out.println("Hello World!");
  }
}
```

Occasionally, when a programmer thinks a section of code is fairly confusing to someone who might be reading it, they will put comments before the code so that the reader can understand it. There are times where the author will be confused by their own code, and the comments will help them out when they have to update their code.

```
System.out.print("Enter a number: ");
int num = scanner.nextInt();

// Check to see if the number is odd by
// seeing if there is a remainder when
// divided by 2
if (num % 2 == 1) {
  System.out.println(num + " is odd.");
} else {
  System.out.println(num + " is even.");
}
```

It’s pretty common for programmers to put a “to do” list in a program (or method) so that they can come back later and add more features. Later on, when reading about methods and program stubs, this may make more sense.

```
private boolean computePrimeness() {
// TODO: Design a sweet, sweet algorithm
  System.out.println("computePrimeness has run");
  return false;
}
```

When debugging, comments are extremely helpful. Suppose we have twenty lines of code, and there is an error somewhere in those lines. We could “comment out” nineteen of the lines and run the program. If it runs successfully, we can uncomment another line (but keep the other eighteen lines commented out). Run it. If no error occurs, uncomment another line. By reducing the amount of code that runs, it is easier to find the error.

Let’s imagine that I want to write a program to create ten random numbers between 1 and 100. In the code below, there is a logic error. Furthermore, let’s imagine I have no idea where that error is. Don’t worry if this code is over your head - you don’t have to know what it does. All you need to understand is the process of adding in comments to help target the location of the buggy code.

```
import java.util.ArrayList;

public class RandomNumbers {
  public static void main(String []args){

    ArrayList<Integer> list = new ArrayList<Integer>();

    for (int i = 0; i < 10; i++) {
      list.add(new Integer((int)Math.random()*100));
    }
    System.out.println(list);
  }
}
```

OUTPUT:

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

*Well that’s weird… What are the chances that all ten random numbers are zero? It’s much more likely I screwed up somewhere. Maybe I declared the `ArrayList` improperly? Maybe I’m not using `Math.random()` right? I’m not sure. So I’m going to comment almost everything out and see if it runs. If it does, I’ll try adding some of the commented lines back into the program…*

```
import java.util.ArrayList;

public class RandomNumbers {
  public static void main(String[] args){
    ArrayList<Integer> list = new ArrayList<Integer>();

       // for (int i = 0; i < 10; i++) {
       // list.add(new Integer((int)Math.random()*100));
       // }

    System.out.println(list);
  }
}
```

OUTPUT:

```
[]
```

*Huh. So I notice that when I run it this time, there is no error. I wasn't expecting random numbers to be generated because I commented out the code to do that. So I think it's safe to say that since nothing else broke, the code that is commented out has the error in it. So maybe now I'll try uncommenting some of the lines of code that were omitted from execution in the last run:*

```
import java.util.ArrayList;

public class RandomNumbers {
  public static void main(String[] args){
    ArrayList<Integer> list = new ArrayList<Integer>();

    // for (int i = 0; i < 10; i++) {
    list.add(new Integer((int)Math.random()*100));
    // }

    System.out.println(list);
  }
}
```

OUTPUT:

```
[0]
```

*Hmmm…. I'm not sure if that zero was really a zero that was randomly generated, or if it's the bug. Lemme try running it again.*

OUTPUT:

```
[0]
```

*And one more time, just for good measure.*

OUTPUT:

```
[0]
```

*I'm calling it! It looks like the error has come back, and it's in the line of code where I generate a random number. I guess I'll really look into that one line of code and dissect it. See if I can't find the error.*

```
list.add(new Integer((int)Math.random()*100));
```

*The error, by the way, is that `Math.random()` generates a number between `0.0` and `0.99999999999`, so I should have multiplied it by 100 before I turn it into an integer. Easy mistake, I do it all the time. You will too!*

## Check Yourself

1. Name the three types of comments used in java and how you create each type of

2. What should be included in the heading comment of any program?

3. Which statement(s) are valid use of line comments:

   a.
   ```
   //variable to hold a person's lucky number
    int num = 0;
   ```

   b.
   ```
   String quote = " "; //hold a student's favorite quote
   ```

   c.
   ```
   //hold a person's age: int age = 0;
   ```

   d.
   ```
   //this variable is a long named variable
   // and will be used to hold a student's
   // favorite number when they were younger
   int thisVariableHoldsANumberFromEarlierTimes = 0;
   ```

4. Compare the following heading comments, which are all basically the same. Create a three column table and highlight the strengths and weaknesses of each. Label the first column "Letter", the second column "Strengths", and the third column "Weaknesses").

   a.
   ```
   //Name: Gordon Freeman
   //Date: 11/8/98
   //Purpose: This program defends the human race against hostile Aliens.
   ```

   b.
   ```
   /* Name: Gordon Freeman
    Date: 11/8/98
    Purpose: This program defends the human race against hostile Aliens.
   */
   ```

   c.
   ```
   /* Name: Gordon Freeman
    * Date: 11/8/98
    * Purpose: This program defends the human race against hostile Aliens.
    */
   ```

   d.
   ```
   /***********************************************************************
    * Name: Gordon Freeman
    * Date: 11/8/98
    * Purpose: This program defends the human race against hostile Aliens.
    *********************************************************************/
   ```

e.
```
/**
 * This program defends the human race against hostile Aliens.
 *
 * @author Gordon Freeman
 * @version 1.0
 * @since 1998-11-08
 *
 */
```

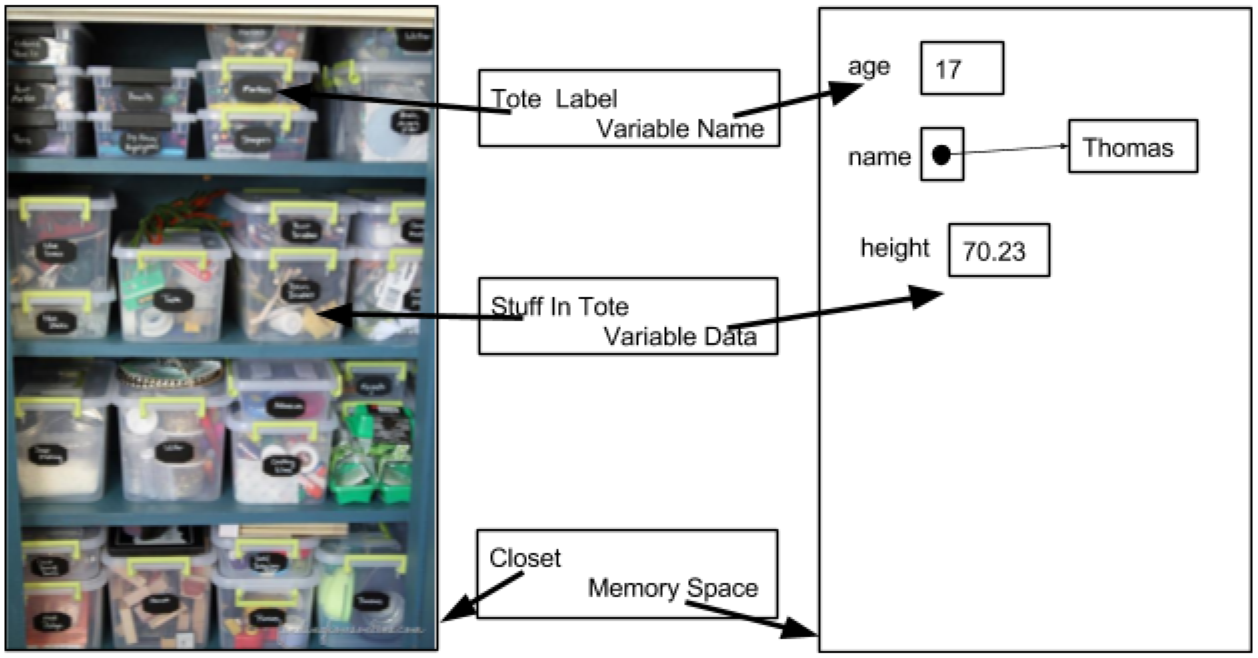
f.
```
/*
 Gordon Freeman
 11/8/98
 This program defends the human race against hostile Aliens.
 */
```

5. Rewrite the following program using only line comments. (You only need to modify the comments, you do not need to understand or modify the program itself.)

```
/*
 * Author: Will McLaughlin
 * Date: 6.29.17
 *
 * This program will print the alphabet to the standard console output.
 */

public class AlphabetPrinter {
/*This is a java application, so has a main method.*/
  public static void main(String[] args) {
    int anUnusedVariable = 0; /* an unused variable */
    /*variable to hold the initial letter to be printed to the console
     * and is initialized to the letter a, the first character to print
     */
    char letter = 'a';
    /*loop through all the letters of the alphabet and display to
     *console
     */

    for(char ch = letter; ch <= 'z'; ++ch){
      System.out.print(ch);
    }
  }
}
```

## §2.2 Basic Use of Variables

Programs need data to work with, and they need to hold onto the data to use later. The data is held, or stored, in memory. The software needs to know where it put the data in memory, so it needs to label the memory location to be able to recover the data later. This is analogous to how a family may store their holiday decorations. You only need them at certain times, so you store them in totes, boxes with labels on them. Variables are the totes with labels. Data is stored in totes (memory), and is labelled (variable names) for access later on.



www.pinterest.com/pin/130322982942249481

Variables will be discussed much heavier next chapter, but for now here are some simple concepts.

### Declaring a variable for use

Before you can declare a variable to hold data, you first need to know two things. (analogous to totes in a closet)

1. Data type (analogous to size of the tote to be used for storage)
2. Variable name (analogous to the label for the tote)

**Data type** is the type of data that the variable will be storing. The three main data types in Java are int, double, and String. These are whole numbers, real numbers, and strings (text). Note, in any Java program, String data is surrounded by double quotes. `"Hello World"` and `"Thomas"` are Strings, however `Hello World` and `Thomas` are not, as they are not between double quotes.

The variable name is the name you want to use for that memory location. For now, you should name your variables lowercase letters with no spaces. (You will study all the rules and naming conventions in the near future, as well as many other data types.)

To declare a variable in java, state the data type and then the name of the variable followed by a semicolon. For example:

```
//Declaring 3 variables, notice the only uppercase letter is the 'S'
//in "String"
int age; //declare a variable named age to hold an integer
double height; //declare a variable named height to hold a decimal
String name; //declare a variable to have a string
```

The above block of code now gives the program space to store data. To put values into the variables, we use the = assignment operator :

```
age = 17; //assign the age variable to the value of 17
name = "Thomas"; // assign name to Thomas
height = 70.23; //set height
```

Now the data is stored in memory and could be depicted as in the diagram 2.2a, located in the previous pages. You can now get their values later on in the program to use, or even change. For example, to calculate the age after a year, you get the current value and add one:

```
age + 1 /*expression to add 1 to the current age*/
```

To update the variable in memory, simply re-assign the variable:

```
age = age + 1; //statement to add one to age, and update the variable
```

Couple of concepts to point out from this example. On the right side of the assignment operator, you have an expression. The expression is evaluated before the assignment happens. To evaluate it, it needs the age, so will look it up in memory and use the current value of 17 in the expression. The value of 1 is then added to get 18. Lastly, the assignment operator replaces the value in the age variable with 18.

**Camel-case and Snake_case**

You have heard of uppercase and lowercase, but what is camel-case? Camel case is a style to write multiple words together, with no spaces, making it easier to read by capitalizing the first letter of each word.

ForExampleThisOneWordSentenceIsWrittenInCamelCase (camelCase)

Some_people_would_say_that_underscores_would_work_better. (snake_case)

Different software languages have different coding conventions, and some lend itself better to snake_case, while others to camelCase. Java uses the camelCase naming convention for variables.

## Check Yourself

1. Many first year Java programmers mostly use what three main data types?

2. What kind of data type should you use for each of the following?

    a. Age of a student
    b. Name of a pet
    c. Total amount on lunch bill
    d. Number of student in a college
    e. Average number of siblings for college students

3. Declare a variable to store the number of classes a student is taking.

4. Declare variables to store data for each of the following:

    a. Age of a student
    b. Name of a pet
    c. Total amount on lunch bill
    d. Number of student in a college
    e. Average number of siblings for college students

5. What is Pascal case and how does it compare to camel case?

## §2.3 Basic Output

A great program has an intuitive and smooth user interface. User interface design is entire branch of software development that will not be covered here. However, we want our programs to let the user know what is going on. The easiest way is to print a message and the value of variables.

### `println`

You have used the `println` method, `System.out.println()`, already in your HelloWorld.java application.

```
System.out.println("Hello World!");
```

This will take the argument, `"Hello World!"`, and print it to the console (output), on its own line. The `println()` method takes only one argument, everything between the parenthesis (....). The argument is typically a `String`, number, or a variable. It will print whatever is in the argument on one line, then move onto the next line for the next output to be made. It is like hitting the Enter key on the keyboard. For example, all the following are valid output calls and would print out three lines of output:

```
System.out.println( "The quick brown fox jumps over the lazy dog." );
System.out.println( 1234 );
System.out.println( name ); //assume the name variable has a value
```

OUTPUT:

```

The quick brown fox jumps over the lazy dog.
1234
Julio
```

The argument can be almost anything, even an entire expression! Also valid method calls:

```
System.out.println( 2 * 12 - 4 + 5 );
//25 (order of operations work here)

System.out.println( total + total * 0.08 );
//display total with 8% tax added to it

System.out.println( "Greetings " + name + ", " );
//example that add Strings together (known as "concatenation")
```

OUTPUT:

```
25
101.52
Greetings Julio,
```

You will study expressions more in the future, but for now, know that you can do mathematical expressions or add `String` data types together. String concatenation is the operation of joining characters together end-to-end. (i.e. adding words together.) For example:

> `"Ground" + "hog"` becomes `"Groundhog"`
> `"Name: " + "Tim"`  becomes `"Name: Tim"` (notice the space after the colon)
> `"Name" + ":" + " " + "Tim"` becomes `"Name: Tim"`

Adding `String` data types with numbers gets a little confusing and will be discussed later in the text.

## `print`

There are times you will want to print, but not move onto the next line of output. For example the output from the following code would not be very good:

```java
String user = "Bill Gates";

//print greeting to user
System.out.println("Hello" );
System.out.println(user);
System.out.println("!");
```

OUTPUT:

```
Hello
Bill Gates
!
```

The output would look much better if the data was printed on a single line. A solution is to print the data within the argument, but do not skip on the next line. The method `print` does just that. The following would be much better.

```java
String user = "Bill Gates";

//print greeting to user
```

```
System.out.print("Hello " );
System.out.print(user);
System.out.print("!");
```

OUTPUT:

```
Hello Bill Gates!
```

## Escape sequences

So `print` and `println` output to the standard console the value of the parameter, whether it is a `String`, variable or an expression. Many times the parameter is a Java `String` literal, made by using double quotes. So whatever you type between the double quotes become a `String` literal such as `"Hello World!"`. However, there are some characters that you type that will cause problems, or not be able to be registered as characters within a `String` literal. For example, what if you want to output the greeting, but have the word `World` quoted? You would try:

```
System.out.println("Hello "World"!"); //ERROR …
Exception in thread "main" java.lang.Error: Unresolved compilation
problems:
      The left-hand side of an assignment must be a variable
      Syntax error on token "World", invalid AssignmentOperator
```

The problem is the double quote before the `W` in `World` ends the `String` literal `Hello`, the compiler does not know what to do with this. There are a handful of these special characters that are difficult for the compiler to handle well. The solution is to use a special character to flag the compiler that the next character is special. The character chosen to flag the compiler is the backslash(`\`). This escape charter is used within the `String` literal (between the `" "`). The sequence of characters is known as an escape sequence. So to solve the issue, we would use `\"` to represent the double quote character:

```
System.out.println("Hello \"World\"!"); //notice the spacing
```

The backslash character tells the compiler that the next character is to be translated as a special character. Do not forget that the escape sequence is text, so needs to be used within a `String` using double quotes. Common escape sequences:

`\"` Insert a double quote in the text at this point.
`\t` Insert a tab in the text at this point.
`\n` Insert a newline in the text at this point.
`\'` Insert a single quote character in the text at this point.
`\\` Insert a backslash character in the text at this point.

## The `String` Class - Output with Escape Sequences

You have done same simple basic Java programs with outputting data, or at least read about them. This following application, a Java class with a `main` method, using the `print` and `println` methods to display various lines of output using escape sequences.

```java
/* Aaron Sullivan Experiment
 * with escape sequences!!
 * 9/21/2017
 */

public class Escape {
  public static void main(String[] args) {
    System.out.print("Hi.\n");
    System.out.print("Aga\nin.\n");
    System.out.println("Hello,\nagain.");
    System.out.println("Aaron says, \"Code well!\"");
    System.out.println("Look! \\ A backslash!");
    System.out.println("\tThis is indented");
  }
}
```

OUTPUT:

```

Hi.
Aga
in.
Hello,
again.
Aaron says, "Code well!"
Look! \ A backslash!
      This is indented
```

## The `String` Class - Output with Expressions

You should understand that a Java `String` is any text found between a set of double quotes. `Strings` are data that are manipulated by programs. When used in a Java program, these are known as `String` literals. Literally, they are a `String`, explicitly written in the program. The `print` and `println` methods take the data and display it in the console output. There are times, as shown earlier, that you can print expressions that involve both `String` data and numbers. Let’s look at different output examples using `String`, numbers, and expressions using addition.

Expressions follow order of operations. The following blocks of code uses only addition, so all computation are computed left to right, unless there are a set of parenthesis. The expression within the parenthesis are calculated before adding.  When adding `String` data together, it puts them end-to-end. When adding numbers together, it evaluates as you would expect.

```
1 System.out.println("3 + 4"); //String literal output:3 + 4
2 System.out.println("3 " + "+"  + " 4"); //String concatenation output:3 + 4
3 System.out.println(3 + 4); //output:7
```

Line 1 is a simple printing of a `String` literal with no expression to calculate

Line 2 is an expression with just `String` data. `Strings` are added together end-to-end. This is called `String` concatenation. It computes the expression and then outputs the single (`String`) value

Line 3 is an expression with just numbers, so it computes it before output the value.

When adding `String` and numbers together, it get complicated. Expressions with both numbers and `Strings` still follow the Java order of operations. When adding left to right, the evaluation depends on the type of data being added. When two `Strings` are added together, they are **concatenated**. When two numbers are added together, they are **evaluated** as a number. When adding a number and a `String`, or a `String` and a number, the number is treated as a `String` and then concatenated.

```
4 System.out.println("3 + 4 = " + 3 + 4); //output: 3 + 4 = 34
```

Line 4 is evaluated left to right. It is adding the `String` `"3 + 4 = "` , with the number `3`. It cannot add a `String` and a number together, so the number `3` is converted to a `String` “3”. Then it concatenates the values to get `"3 + 4 = 3"`.  The rest of the expression is then evaluated: `"3 + 4 = " + 4` . The same process works here  `String` + number, convert number `4`  to a `String` “4” and concatenate to get `"3 + 4 = 34"`.

OUTPUT:

```

3 + 4 = 34
```

Look at the following examples and try to understand the output. These are evaluated using order of operation, which is mostly left-to-right. If you are adding numbers, simply add the values. If you are adding `Strings`, concatenate. If you are adding a number with a `String`, convert the number to a `String` and concatenate.

```
System.out.println(3 + 4 + " = 3 + 4");
//output:7 = 3 + 4
```

```
System.out.println(3 + 4 + " = 3 + 4 = " + "3" + "4");
//output:7 = 3 + 4 = 34

System.out.println(3 + 4 + " = 3 + 4 = " + 3 + 4);
//output:7 = 3 + 4 = 34

System.out.println(3 + 4 + " = 3 + 4 = " + (3 + 4));
//output 7 = 3 + 4 = 7
```

## Check Yourself

1. What are the issues with each of the following lines of code.

   a. `System.out.prntln("Hello World!");`
   b. `System.out.println("Hello "  name );`
   c. `System.out.println("Hello \"World\");`

2. Write the output for each of the following EXACTLY:

   a. `System.out.println("Hello\n\t\"World\"\n!");`

   b.
   ```
   System.out.print("The");
   System.out.print("quick " + "brown " + "fox ");
   System.out.println("jumps over");
   System.out.print("the lazy\n");
   System.out.print("dog!");
   ```

   c. `System.out.print("Sum of 1,3, and 5 = " + (1+3+5) + "." );`

   d.
   ```
   String name = "Ted";
   System.out.print("Who would like to see " + name + " talk?" );
   ```

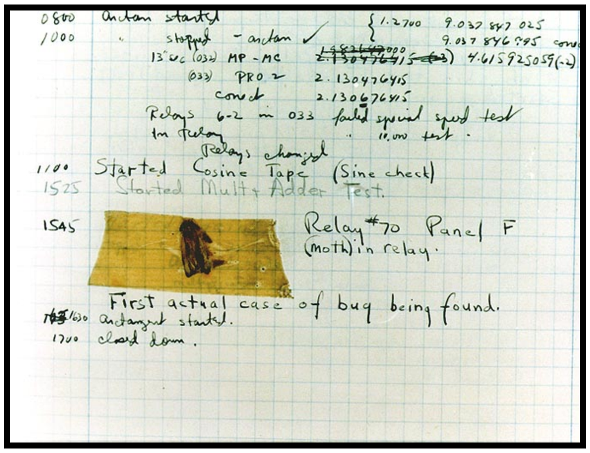
3. Rewrite the following in a single `println` method:

   a.
   ```
   System.out.print("The ");
   System.out.print("quick " + "brown " + "fox ");
   System.out.println("jumps over");
   System.out.print("the lazy");
   System.out.print("dog!\n");
   ```

   b.
   ```
   String ai= "J.A.R.V.I.S";
   System.out.print("\"");
   System.out.print(ai);
   System.out.print(", sometimes you gotta run before you can ");
   System.out.print("walk.\"-Tony Stark\n");
   ```

## §2.4 Errors

If anyone tells you that all their code is bug-free, they're lying. Coding is a fragile business, and we get errors all the time. All too often, the act of fixing one bug causes at least one more bug to surface. This means it is even more important to test your code incrementally - so that small bugs can be discovered and managed before the project gets very large.



0800 Anctan started
1000 " stopped - anctan ✓ {1.2700 9.037 847 025
9.037 846 995 conct
13"uc (032) MP - MC ~~2.130476415~~ (-3) 4.615925059(-2)
(033) PRO 2 2.130476415
conct 2.130676415
Relays 6-2 in 033 failed special speed test
in relay " 11.000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Antangent started.
1700 closed down.

*First documented computer bug*
*Grace Hopper - September 9th, 1947*

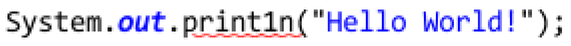
The story goes that Grace Hopper (remember her from the first chapter?), one of the lead programmers of the Mark II in the mid 1940's, found a moth trapped in one of the relays in the computer. This caused a short, which in turn caused erroneous data to be reported. Hence, the term, "bug".

### Types of Errors

The good news is that there are only three types of errors that programmers can make when programming in Java. The bad news is that errors happen often. All. The. Time.

#### Syntax Error

A **syntax error** is essentially a typo. When using an IDE, most editors will even underline the error in red (much like a spell checker in word processors). Syntax errors will prevent the software from running. Usually, it's an easy fix (a semicolon has been omitted), although every now and then the syntax errors can be harder to understand. The following code will cause a syntax error:



```
System.out.print1n("Hello World!");
```

The issue is that whoever wrote this code used a number “1” instead of a lowercase “L”. In the monospace font, it’s difficult to tell. Something like that may take a while to figure out because it is not really a Java mistake as much as it is a font mistake.

```
int num1 = 10;
int num2 = 12
```

The code above demonstrates the time-honored tradition of forgetting a semicolon. The program won’t compile, and you’ll get a nastygram on the screen from the IDE. Everyone forgets semicolons - so if you haven’t yet, don’t worry. You will.

Many IDEs will have a few hiccups when dealing with syntax errors. Because of the partial compilation required to make IDEs detect the syntax errors, it can be a little laggy - so if you fix an error and it is still underlined, you may have to save the document just to force the IDE to scan it again for errors. Also, IDEs sometimes reports errors on a line that is actually caused by a syntax error on a previous line. For example, the IDE may say that line 51 has an issue - let’s say a missing semicolon -, but really line 50 has the issue and the IDE is just confused. Usually, reading the error message by floating the cursor over the red squiggly underline will provide more context for the issue.

Bottom line - a syntax error is a typo and will prevent the software from even starting.

### Runtime Error

A **runtime error** happens when the program is running. Oftentimes, this means the software may run reliably for a number of trials, but will crash every once in awhile. This isn’t a problem with the software in the sense that the program is quitting for no reason, rather a runtime error is an indicator that the software isn’t handling some things properly. Runtime errors will stop a program dead in their tracks and you’ll get the “Red Text of Death”.

The quintessential example of a runtime error occurs in a primitive calculator. Here is the output and input from the first attempt. Note that the bold, underlined writing is what the user types in.

```
Please enter a number: 20
Please enter another number: 4
20 / 4 = 5
```

In the previous example, the first number was divided by the second number without incident.

```
Please enter a number: 18
Please enter another number: 6
18 / 6 = 3
```

In the example above, the first number was divided by the second number (again), and there were no problems (again).

```
Please enter a number: 17
Please enter another number: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
at adding.Testing.main(Testing.java:16)
```

Here we have a catastrophe - we think the program is fine because it worked twice, but it conked out the third time around. The problem is that computers don't much like it when you try to force them to divide by zero. The author of this program made the assumption that any two numbers can be divided. In retrospect, the author *probably* should have checked to see that the second number wasn't a zero (and if it was, the program should then go back and ask the user to enter a number that is not zero). By the way - the red writing is called a "**stack trace**". It's a breadcrumb trail that you can use to figure out where the program went wrong. In this case, the stack trace is only a few lines long. In some cases it can be much larger.

Another common mistake is when the program tries to store a value that the user entered into a variable that is incapable of storing that type of data. For instance, in this program, the computer asks the user for a number. In the first example, there is no issue. But the second time around, the user tries to enter a `String` instead of a `double`.

```
Enter a number: 21.3
Enter a number: Weird Al
Exception in thread "main" java.util.InputMismatchException
at java.util.Scanner.throwFor(Unknown Source)
at java.util.Scanner.throwFor(Unknown Source)
at java.util.Scanner.throwFor(Unknown Source)
at java.util.Scanner.throwFor(Unknown Source)
At adding.Testing.main(Testing.java:14)
```

Again, we can use this stack trace to find the problem. In this case, the stack trace starts off by saying there is an `InputMismatchException` (which shouldn't mean anything to you yet), but that's what caused the problem. The good news is that it looks like the problem originated from the code on line 14, so that's a good place to start. *If you haven't figured it out, an `InputMismatchException` is when the computer is expecting one type of input - like an `int` - but gets a different kind of input.*

Bottom line - runtime errors may not always happen, but when they do, it's never a graceful crash (and it can usually be prevented by good coding and solid **error handling**!).

### Logic Error

**Logic errors** are the most frustrating because we may never even know they exist. At least a runtime error and syntax error have the decency to make you aware of their presence.

These happen because the computer does exactly what we told it to do (it's just that we told it the wrong thing).

```
double num1 = 20;
double num2 = 2;
System.out.println(num1 + num2 / 5);
```

In this example, the programmer probably intended to add the two variables (`num1` and `num2`) and then divide by `5`. Instead, they disregarded precedence and the computer divided `num2` by `5` first and then added the answer to `num1`. This is an easy mistake to make, and if you haven't made it yet don't worry - your time will come.

```
System.out.print("Enter a number: ");
int number = scanner.nextInt();

if (num1 > 10) {
  System.out.println("Your number is big.");
} else if (num1 < 10) {
   System.out.println("Your number is small.");
}
```

There is a logic error in this code, too. Do you see it?

It’s an error by omission. The programmer accounted for the case when `num1` was greater than 10 and the case when `num1` was less than 10. But if `num1` equals 10, then what happens? What should happen? Most likely the programmer should have code in there to react appropriately when `num1` equals 10. Don’t get bent out of shape if you couldn’t decipher that code segment - we haven’t even talked about conditionals yet (but we will soon!).

One of the most popular errors is the Off By One Error (OBOE). This often happens when “walking through arrays” (don’t worry about what that means right now). Although an array might not have much contextual value to you yet, it’s safe to think of arrays as a bunch of lockers. It’s pretty common in Java to want to open every locker in a row and peek inside. So if you have 10 lockers, the code might look like this:

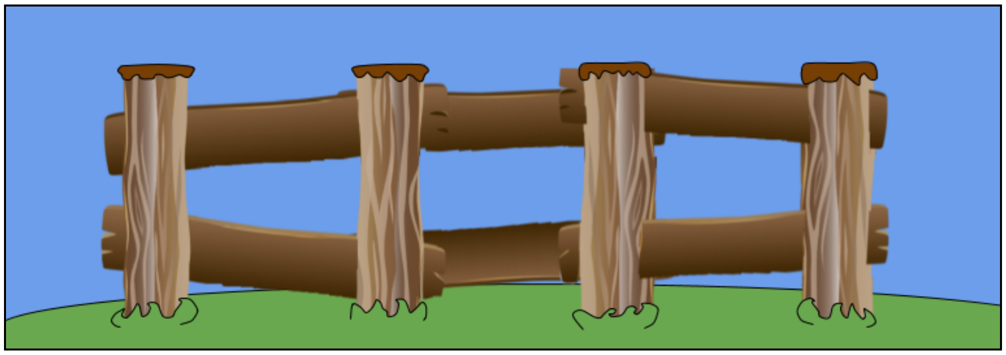
```
for (int i = 0; i <= 10; i++) {
 System.out.println(array[i]);
}
```

Recognizing that we haven’t covered the concepts of arrays and loops yet, we *might* be able to understand that this program won’t work because the computer is trying to look in a row of 11 lockers (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10). That’s bad news because there are only 10 lockers. The program should use `i < 10` instead of `i <= 10`. Because computer scientists start counting at zero instead of one, it’s easy to make this mistake.

A similar logical mistake is the “fencepost error”. It’s a simple counting error, but it can cause wacky results. The name comes from the physical act of building a fence. Imagine we were going to build a split-rail fence that is three sections long. How many fence posts do we need (a fence post is one of those big pieces of wood that is vertical, goes in the ground, and supports the horizontal fence pieces).

Did you say three?

It’s easy to say that we need three fence posts if we are building a fence with three sections. But we really need four fence posts! The diagram below demonstrates this.



*A three panel fence needs four fence posts!*

## Famous Errors

The following are some of the most popular errors that programmers experience. Most of the examples contain code outside the scope of this chapter. However, it would be wise to revisit this section as you move forward through the text. There's a good chance that you'll encounter these errors:

**Missing Semicolon [SYNTAX]**

Description:

- This happens when you forget to punctuate a line of code with a semicolon.
- This error happens often when you first start to program, and will sporadically haunt you until your dying day.

Code Snippet:

```
public static void main (String args[]) {
  System.out.println("Hello World")
}
```

Example of Error Message:

- `error: ';' expected`

Way to fix/avoid

- You can't. It's inevitable. You cannot escape your destiny. #finalDestination

**Improper Capitalization [SYNTAX]**

Description:

- Forgetting a capital letter (for instance, the "S" in "System").
- Even though programmers won't get errors if they capitalize variable names (it's technically legal in Java, just against convention), you will get disapproving looks from your peers.

Code Snippet:

```
public static void main (String args[]) {
  system.out.println("Hello World");
}
```

Example of Error Message:

- `error: package system does not exist`

Way to fix/avoid

- Just be hyper-vigilant to Java rules!

**Division by Zero [RUNTIME]**

Description:

- This doesn't happen too often, but when a number is divided by zero, you'll know (the program crashes and a stack trace is produced).
- You may actually have this error present in your code, but you may never know because in your testing, you don't ever encounter a scenario where division by zero happens. Don't worry - the person you are writing code for will let you know you screwed up.

Code snippet

```
public static void main (String args[]) {
  int a = 12;
  int b = 0;
  System.out.println(a/b);
}
```

Example of error code

- `ArithmeticException: / by zero`

Way to fix/avoid

- You probably should just be diligent about pressing the "/" key when coding - make sure you consider times when the computer may encounter dividing by zero.
- Note that the same error will surface when modding ("%"), too.
- When testing code, use fringe cases (like zeros).

**Unnecessary Semicolon [LOGIC]**

Description:

- You'll most likely encounter this when you first start using if statements and loops, but once you get used to these concepts, you probably won't experience it too often.

Code snippet

```
if (numberOfStudents > 20); {
  System.out.println("Need Overload Slip");
}
```

Example of error code

- This is a logic error, so you won't get an error message.
- Most likely, you'll notice that the loop doesn't function right (the code in the loop will always execute, and it will only execute once).

Way to fix/avoid

- Don't put a semicolon after a conditional statement or loop control statement.

**Zero Index [LOGIC]**

Description:

- This error happens often when you first start to program and will take a bit to get used to. Once you get it, though, you'll probably never see it again.
- This error usually happens when dealing with arrays and Strings.
- It is often referred to as an “Off-By-One-Error” or a “Fencepost Error”, and means that the number of iterations a loop had was off by one - either one too many or one too few.

Code snippet

```
public static void main (String args[]) {
  String firstName = "Stringfellow";
  System.out.println(firstName.charAt(1)); // t
}
```

Example of error code

- Because this is a logic error, you will not generate an error.
- The output will not be what you expect - in this example, the programmer wanted to retrieve the first letter of the name, but they should have used "**`firstName.charAt(0);`**"

Way to fix/avoid

- Be aware that computer scientists start counting at zero!

**Order of Operations [LOGIC]**

Description:

- Get used to this one… It takes a while to shake this error.

Code snippet

```
public static void main (String args[]) {
  int a = 10, b = 20;
  System.out.println("Sum: " + a + b); // Sum: 1020
}
```

---

```
public static void main (String args[]) {
  int a = 40;
  int b = 50;
  int c = 60;

  System.out.println(a + b / c); // 40
}
```

---

```
// See error in §2.1 regarding RandomNumbers
```

Example of error code

- No error code will be displayed because this is a logic error.
- You'll probably see this more when trying to output the values of variables in a statement that also outputs a String.

Way to fix/avoid

- Use parentheses when doing mathematical computations.
- Remember that the "+" sign in a System.out.print statement will concatenate before it adds (order of operations).

**Assignment Versus Comparison [LOGIC]**

Description:

- This error happens often when you first start to program, but fades with time as you get used to the environment.
- It gets conflated with the zero-indexed logic error often, so be on the lookout.

Code snippet

```
public static void main (String args[]) {
  int answer = 33;
  if (answer = 42) {
    System.out.println("The answer to everything!");
  }
}
```

Example of error code

- Sometimes you'll see an error, sometimes you'll just see wonky results.
- error: incompatible types: int cannot be converted to boolean

Way to fix/avoid

- Don't forget that one equal sign, "=", means *assign* and two equal signs ("==") means *compare*.

**"<" instead of "<=" [LOGIC or RUNTIME] -**

Description:

- In the best-case scenario, this will just be a logic error and will yield odd results
- In the worst-case scenario, this will stop your program dead in its tracks.

Code snippet

```
public static void main (String args[]) {
  int sizeOfCrew = 9;
  if (sizeOfCrew < 9) {
    System.out.println("Firefly!");
  }
}
// This code should use sizeOfCrew <= 9, so unexpected
// results will occur.
```

---

```
public static void main (String args[]) {
  String name = "Nathan Fillion";
  for (int i = 0; i <= name.length(); i++) {
    System.out.println(name.charAt(i));
  }
}

// This will barf because the for loop should only
// iterate i < name.length() because if i is actually
// the length of the String, then charAt(i) will throw
// an error (recall that the first index in a String is 0,
// and the last is length-1).
```

Example of error code

- When this is a logic error, you'll only get weird results.
- `IndexOutOfBoundsException`

Way to fix/avoid

- Uh, just be careful.

**Code outside of brackets [LOGIC or SYNTAX]**

Description:

- This happens if you don't pay attention to the brackets in your code.
- If you don't use brackets for conditionals and loops, you will encounter this issue.
- If you put relevant information outside a set of relevant brackets (instead of inside), you'll get this issue (usually a logic error, but occasionally a syntax issue if you have scope - don't worry about what scope is yet).

Code snippet

```
public static void main (String args[]) {
  if (a == b)
  System.out.println(a + "Is equal to ");
  System.out.println(b);
}

// If a equals b, the first System.out.println() is called
// but no matter what, the second println() will always be
// called - it is not associated with the if statement
// because there are no brackets with the if statement, so
// only the first line of code is contingent on the
// conditional. The code should read:

public static void main (String args[]) {
  if (a == b) {
    System.out.println(a + "Is equal to ");
    System.out.println(b);
  }
}
```

Example of error code

- If this is a logic error, you'll just get odd results.
- `error: unreachable code`
- `error: cannot find symbol`

Way to fix/avoid

- *You need to have a solid appreciation for programming conventions! You need to know when to use brackets and the implications of the brackets!!!!*
- Most IDEs have an autoformatting function - be sure to invoke this often to help train you!

## Check Yourself

1. What are the three classifications of errors generated in Java?

2. What kind of error is generated by each line of code:

   A. `System.out.println("Hello World!")`

   B. `double average = (82+90+87) / 4;`

   C. `int age = 18.5;`

   D. `System.out.printn("Hello Again!");`

   E. `int area = length + width;`

   F. `System.out.println("Sum of 1 and 3 = " (1+3) );`

# VARIABLES AND MATH

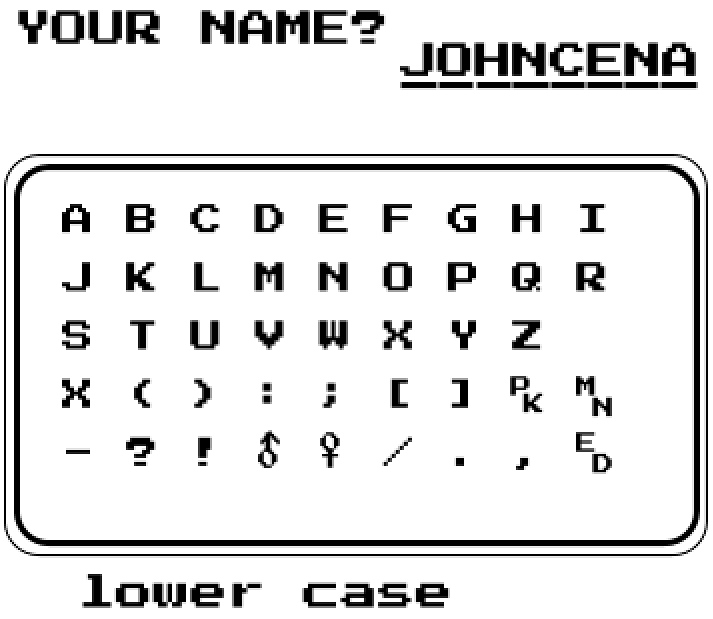
"Control all the Variables you can. . . . don't worry about the rest"

**- Forrest Griffin -**

*Great advice if you are a UFC fighter.*
*Bad advice if you are a programmer.*

## §3.1 Primitive Data Types

When we code, we need places to store information (only for the duration of the program). For instance, if you've ever played the original Pokemon game on GameBoy, when you first start a game, you are prompted for your name:



Throughout the game, you are referred to as whatever you typed your name in as. This is a **variable**. Typically, variables will have names that are self-explanatory. For instance, in the code for Pokemon, there is probably a variable called `name` or `firstName` that would store whatever the user enters. Data come in all shapes and sizes - integers, decimals, characters, Strings, and even funky things like arrays, `ArrayLists`, and `Scanners` (stay tuned for all that good stuff).

Programming can be thought of as nothing more than manipulating data. So to be a good programmer, you will want to understand how data is stored, used, and changed. The first thing to know about data, is how each piece of information is classified. We start with the building blocks of all data, the primitive data types.

### Primitive Data Types

There are eight types of **primitive data types**. A primitive data type is a fundamental building block in Java - anything that isn't a primitive data type is an **object**. Not all these data types will be used in this chapter, or even in the text. However, for completeness, all eight data types will are listed here.

| Primitive Data Type | Contains | Default value | Size (Bytes) | Size (Bits) | Range |
|---|---|---|---|---|---|
| `boolean` | true or false | false | NA | 1 | NA |
| `char` | Unicode character | \u0000 | 2 | 16 | /u0000 to /uFFFF |
| `byte` | Signed integer | 0 | 1 | 8 | -128 to 127 |
| `short` | Signed integer | 0 | 2 | 16 | -32,768 to 32,767 ($-2^{15}$ to $2^{15}$) |
| `int` | Signed integer | 0 | 4 | 32 | -2,147,483,648 to -2,147,483,647 ($-2^{31}$ to $2^{31}$) |
| `long` | Signed integer | 0 | 8 | 64 | -9,223,372,036,854,775,808 to -9,223,372,036,854,775,808 ($-2^{63}$ to $2^{63}$) |
| `float` | Floating point | 0.0 | 4 | 32 | $\underline{+}$1.4E-45 to $\underline{+}$3.4028235E+38 (around 7 significant digits) |
| `double` | Floating point | 0.0 | 8 | 64 | $\underline{+}$4.9E-324 to $\underline{+}$1.7976931348623157E+308 (around 16 significant digits) |

*Table describing the 8 primitive data types in Java (the building blocks of all data used in Java programs). Note that you don't have to memorize this. At all.*

**`int`**

An `int` is capable of holding integers (whole numbers) that are both positive and negative. `int`s can store values roughly between negative 2 billion and positive 2 billion. For instance, the following are all valid values of `int`. It is 4 bytes.

```
8, -5, 132
```

**`short`**

A `short` is similar to an `int` (it can hold whole numbers - both positive and negative), but it has a much smaller range. This is great when memory is an issue (like military radios), but often necessary for most programs. A `short` is only 2 bytes, and can hold numbers between -32,768 and 32,767.

```
8, -5, 120
```

**`byte`**

A byte is like a puny short. It is only 1 byte, so the numbers it can hold are between -128 and 127.

```
8, -5, 120
```

**`long`**

This is also like an `int`, but a `long` can hold numbers *much* larger than `int`. Actually, a `long` can hold numbers between negative 9 quintillion and positive 9 quintillion (give or take). Again, there are times when programmers will leverage a long, but for the scope of this course we probably won't need it too often. A `long` is 8 bytes (64-bits!).

```
145, 394, 965, 903, 560
```

**`double`**

A `double` is what's known as a *floating point* number. Your graphing calculators had a "float" mode that you may have seen. In the real world, we refer to these numbers as *decimals*. So, yeah, you're familiar with them. A double can hold decimals - positive and negative. A `double` is 8 bytes, and the range is *freaking huge*. Even though they take up more memory, we still default to `double` in this class. Don't worry - your computer won't really notice a difference.

```
3.2, 5.0, -8.4
```

**`float`**

Just like a double, but a bit smaller. A variable of type float is 4 bytes, and is used for precision decimals. In this course, we won't really use float all that much.

```
3.2, 5.0, -8.4
```

**`char`**

Think of a `char` as just one key press. It comes from the shortening of the word *character*. So, if you press 'u', that's a `char`. So is '='. Or '{'. Or ' ' (that's a space!). It's a bit more inclusive - there are roughly 65,000 different UNICODE symbols (because it's 2 bytes).

You can see a UNICODE table here: http://unicode-table.com/en/. This covers all the letters in the English language (both lowercase and uppercase, numbers, symbols, and characters from most other languages. With room to spare. There's even WingDing type characters in UNICODE. And believe it or not, UNICODE is responsible for emojis (check out this fascinating story at 99% Invisible). It's important to note that variables of type `char` are represented with a single quote (use double quotes for a `String` - see below)

```
'c' 'C' '4' ' ' '['
```

**`boolean`**

Named after George Boole (1815-1864), `boolean` refers to Boolean Algebra, which is mathematical logic using truth values. There are only two possible values for a boolean variable:

```
true, false
```

Hopefully you noticed that there are several different ways to store an integer value. Why so many? The easy answer: for efficiency. Say you wanted to upload the ages of all people living in the United States for statistical analysis. There are roughly 321 Million different ages that would needed to be manipulated (downloaded, saved, loaded, used in calculations, etc.). If they were stored as a `long` data type, that would be 8x321 Million bytes (roughly 2.5 gigabytes) of data. If you stored them as a `byte` data type, you would only need 321 megabytes. The program would be able to work 8 times faster (in theory) if the ages were stored as `byte` data types and not `long`. The same argument holds for real numbers when comparing the `float` and `double` data types. For many of you programs, use `int` data type for integers, and `double` data type for real numbers.

## Strongly typed languages?

Java, like many languages, are considered **strongly typed,** or strictly typed. Strongly typed languages will generate errors or be unable to compile when the types of data (values) do not match. The following line of code is an example of this.

```
int x = 34.5;
```

In this example, the compiler checks the data types and finds that the value your are trying to assign `x`, is not a whole number, so will generate an error and will not compile. The compiler will do a process to verify the data is accurately assigned and used based on the type of data or variable. This process is called **type checking**.

So in contrast a software that is **weakly typed**, or loosely typed, does not require a variable to be defined with a type and will typically allow for mixing data types in an expression. Javascript, Python, and Perl are three examples of weakly typed languages.

## Check Yourself

For the following questions, choose the data type for the suggested value that will store the right kind of data with the least amount of wasted memory.

1. The number of meters from the Earth to the sun as a whole number. (over 149 billion)

2. The first letter of your name.

3. An internet country code top-level domain. (Examples: .kr, .uk, .ca )

4. The current temperature up to the tenths place. (Example: 32.5)

5. The day of the month as a number.

6. Keep track of whether it is a workday or not (it either is, or isn’t).

7. The current year (and should work for at least the next 20,000 years)

8. A floating-point number with more accuracy than `float` can provide.

## §3.2 Declaring and Assigning Variables

As highlighted in the last section, there are different types of data, or data types, that exist. The number 4 is different than 4.0. Each is different than the letter ‘4’, and is much different than the word “four”. The type of data will dictate how much memory will be needed to hold, or store, the values. For example the number 4 is an integer, so will need to be an int, and will need 2 bytes of data. The table below illustrates the different size memory space each 4 will require because each are stored in a different way. (“four” is above the scope of this chapter).

| Given value | Type of data | Java data type | Number of bytes |
|---|---|---|---|
| `4` | integer | `int` | 4 (32 bits) |
| `4.0` | real number | `double` | 8 (64 bits) |
| `‘4’ (\u0034)` | letter | `char` | 2 (16 bits) |

Storing data in memory is similar to storing items in an attic. Each item will require a different size box. The boxes will be labeled, so items can be easily found. Likewise, in programming, when you store data in memory, you will give them a label. Declaring variables to store data is like labeling a box to store seasonal clothes.

There is a difference between **declaring** and **assigning** variables, and that distinction is very important. New programmers often do not differentiate between the two, but knowing the difference will help avoid many common mistakes.

### Declaring Variables

Declaring is “telling the computer that you are allocating space for a data type (`int`, `double`, etc.), but you don’t have to give it a value yet”.

```
int age; // There is no assigned value in age
```

All variables are declared with a declared data type and a name. The type is used to make sure that only the data of that type is stored in that memory location. The name of the variable is used to associate (label) the spot in memory where the value is. This is like the storage totes mentioned earlier with labels on them. Depending on the item (data type) being stored dictates how large the tote (memory) is.

All data is declared with two identifiers, followed by a semicolon.

```
<data type> <variable name>;
```

Examples:

```
char gender;     // the variable gender will hold a character value
                 // such as 'f'

int numOfSibs;   // to store the number of siblings

double gpa;      // hold the real number for a grade point average

String name;     // variable to hold the user's name
```

You cannot declare multiple variables that have the same name! That's like George Foreman naming all his sons "George". It just leads to too much confusion sitting around the dinner table. Compilers cannot have such confusion, so they will not allow you to declare variables with the same name.

It's fair to even declare a number of variables at once (assuming they are all of the same type):

```
int age, weight, height;
```

It's even fair to even declare and assign a number of variables at once (assuming they are all of the same type):

```
int age, weight = 150, height = 70;
// note that 'age' was not assigned a value, so attempting to
// access the state of 'age' will result in a compilation error
```

## Naming Variables

There are several rules to naming the variables in your program:

1. Start with a letter, dollar sign, or underscore*.
   (54 options: a,b,c, …, z, A, B, C, …, Z,$,_)

2. Variable names must contain only
   a. Letters (a,b,c, …, z, A, B, C, …, Z)
   b. Digit (0,1,2,...,9 )
   c. Dollar sign ( $ )
   d. Underscore ( _ )

3. Variables cannot be a **reserved word** (like `int`, `double`, `for`, …)

4. By convention (coding standards), all variables should start with a lowercase letter.

The following are valid variable identifiers:

```
a
age
this_variable_name_is_a_long_one_but_is_still_valid
thisVariableNameIsAlsoAcceptable
num1
num2
```

These variable names are valid but are discouraged:

```
FirstName
// starts with an uppercase letter

$name
// $ The dollar sign is intended for use only in mechanically
// generated code and is used internally by the compiler to decorate
// certain names

_LastName
// does not start with a lowercase letter

THIS_WILL_WORK
// does not start with a lowercase letter
```

The following are invalid variable names:

```
double
// invalid because double is a reserved identifier

2Times
// invalid because it starts with a number

first name
// invalid because it contains a space
```

**Self commenting … the way to program!**

What is the following block of code doing?

```
int a = 4;
int b = 5;
int c = 6;
int d = a*b*c;
int e = 2*(a*b + a*c + b*c);
System.out.println(d);
System.out.println(e);
```

It is not very clear, it is a bit of a riddle … you could add comments to help:

```
int a = 4; //length of a prism
int b = 5; //width of a prism
int c = 6; //height of a prism
int d = a*b*c; //calculate the volume of a prism
int e = 2*(a*b + a*c + b*c); //calculate the surface area
System.out.println(d); //display volume
System.out.println(e); //display surface area
```

Or even better, make your variable self commenting, meaning you can program your variables to comment themselves:

```
int length = 4;
int width = 5;
int height = 6;
int volumeOfPrism = length*width*height;
int surfaceArea = 2*(length*width + length*height + width *height);
System.out.println("Volume: " + volumeOfPrism);
System.out.println("Surface area: " + surfaceArea );
```

## Assigning Variables

Assigning a variable is "giving the variable a value". I like to refer to it as "breathing some life into the variable".

```
int age;
// age is declared, but not initialized (has no value assigned) … yet

age = 38;
// Now age has been assigned a value - it's 38!
```

You can assign a variable once it has been declared. Sometimes people even declare and assign them in one fell swoop, and is the preferred method for Java programmers:

```
// Declare 'age' as an int and give it a value of 38
int age = 38;
```

Oftentimes, programmers will declare a number of variables at once and only assign a few of them:

```
int age, weight = 160, height;
// 'age' has been declared, but not initialized
// 'weight' has a value of 160
// 'height' has been declared, but not initialized
```

You can even reassign it multiple times in a program - it's like recycling! **#CaptainPlanet**

You cannot assign a variable if it has not been declared (created). Try it. See what happens.

```
int age;
weight = 160; // BAD THINGS WILL HAPPEN
```

**Keep it consistent …**

Like many other programming languages, there are options and some flexibility to programming styles to complete tasks in Java. Declare and initializing variables in java is one such example. The following blocks of code each do the same thing, declare 3 grades and initialize them to 0.

Option 1:

```
int grade1 = 0;
int grade2 = 0;
int grade3 = 0;
```

Option 2:

```
int grade1, grade2, grade3;
grade1 = 0;
grade2 = 0;
grade3 = 0;
```

Option 3:

```
int grade1 = 0, grade2 = 0, grade3 = 0;
```

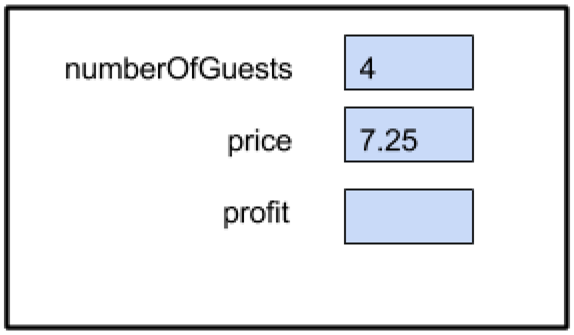
Which one should you go with if they are all valid options? You can easily find examples of each option above when reading code from other sources. Finding many variations on code is common. So, what option is correct? There is no set solution, however, typically there is a preferred way … just check what coding standards you should be using.

## Variables in Memory

Many times when testing and debugging a program, you will trace the execution. That is, you will be the compiler and run the program yourself on paper. You will go line by line in the program, keeping track of the variables and their values. To do this you will use diagrams to represent the state of the program, a snapshot of the current data values of the program.

Take a look at the diagrammed memory for the following lines of code:

```java
int numberOfGuests = 4;
double price = 7.25;
double profit;
```



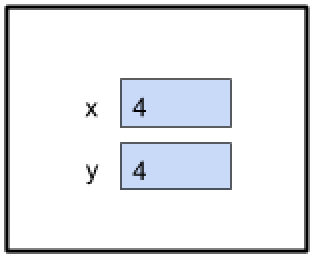
Trace through the following program (line numbers are shown to help with the trace and are not actually part of the code):

```java
1   public class MemoryDiagramEx {
2
3     public static void main(String[] args) {
4
5       int x = 4;
6       int y = x;
7
8       System.out.print("X = ");
9       System.out.println(x);
10      System.out.print("Y = ");
11      System.out.println(y);
12
13      x = 8;
14        System.out.print("X = ");
15        System.out.println(x);
16        System.out.print("Y = ");
17        System.out.println(y);
18      }
19    }
```

Let’s focus on lines 5 and 6 (shown here):

```
5         int x = 4;
6         int y = x;
```

The variables in memory would be 4 for both variables. Line 5 should be trivial, declare a variable named x to hold an integer, and assign the value of 4 to the variable x. Line 6 is a little less trivial, declare a variable named y to hold an integer … evaluate the value of x by looking up the value in memory, and assign that value of 4 to `y`. We denote memory with a box, and the variables names being the label of the memory locations:



So the output for the first block or `print` statements (lines 8-11) would be:

```
8         System.out.print("X = ");
9         System.out.println(x);
10        System.out.print("Y = ");
11        System.out.println(y);
```

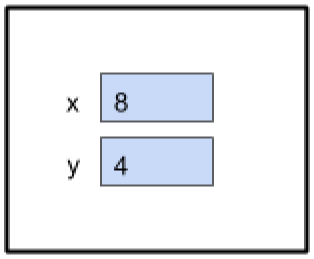
OUTPUT:

```
X = 4
Y = 4
```

Now let’s look at lines 12 and 13:

```
12
13        x = 8;
```

This code changes the value of x, assigns the value of `8` to variable x. So memory would now look like:



And the final output (lines 14-17):

```
14      System.out.print("X = ");
15      System.out.println(x);
16      System.out.print("Y = ");
17      System.out.println(y);
```

OUTPUT (prior output is grayed out)

```

X = 4
Y = 4
X = 8
Y = 4
```

Many early programmers would have both values change to `8`, because we set we set the value for variable `y` to the value of variable `x` with the statement `y = x;`

Early programmers link the variable `y` with `x`, thinking that if you change `x`, `y` would also change.

However, after we set the value of `y` to `4`, that statement is done, and the program continues. After that, the only way to change the value of `y` is to assign it using the assignment operator (the equal sign) again. For example, after the final output, you could change the value of `y` to `12`:

```
y = y*3;
```

**Changing the value of a variable**

The only way to change the value of a variable is to use the equal operator. For example

```
age = age + 1; //add a year to the age variable
```

As a rule of thumb this is true. There are a couple of exceptions to the rule, but are simply variations of the equal operator or is a special kind of operator. These will be discussed in the near future and include the auto-increment and auto-decrement.

The point is: **THE ONLY WAY TO CHANGE THE VALUE OF A VARIABLE IS THE ASSIGNMENT OPERATOR.**

**What are State Diagrams?**

If someone asked you “How are you feeling?”, you would give them the state you are in. Depending on the state you are in, would dictate what the other person would do. If you were in a sad state, they may tell you a joke, or give you a gift. If you were hurt, they would likely try to help. Likewise, many computer scientists, game developers, robotic engineers, AI developers, and others in related fields will use state diagrams to help develop software to decide what the program should do next.

Depending on all the data of the program/machine, the software will be in a particular state. The program will move to a different state when the data in the software changes enough to put it into a different state. An easy example would be a computer game, the game could be in a pause state, load state, gameplay state, ...etc.

State Diagrams are diagrams that represent a particular state. There are many different variations on how to set up these diagrams and where they are used. To scratch the surface, take a look at any of the following:

- https://en.wikipedia.org/wiki/State_diagram
- https://en.wikipedia.org/wiki/Finite-state_machine
- https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867

## Check Yourself

1. Label each line either declaration statement, assignment statement, or both.
   a. `int teaCups = 6;`
   b. `coasters = 4;`
   c. `int guests;`
   d. `guests = 5;`
   e. `double tableCost = 14590.72;`

f. `boolean inviteJabba;`

2. What are the values of the three variables after the code below is executed?

```
char first = 'a', second;
second = first;
first = 'y';
char last = first;
```

3. Explain why each chosen variable identifier below is not valid or against convention.

```
1number
char
WholeNumber
small change
smile^_^
```

## §3.3 Primitive Data Types and Objects

### Objects

We've already talked about primitive data types quite a bit. But there is another type of data, too. It's called an **object**. We don't dive too deep into objects in this course, but you should be aware of their existence. Think of objects as different programs that have some attributes and some abilities. Now, this is a really misleading oversimplification, but it'll do for now.

Primitive data types are easier to deal with than objects (or classes). Here's an example of how you could create a variable called `age` that holds an integer:

```
int age;
```

And here's how you would create a variable called `listOfStudents` that is an `ArrayList` (an `ArrayList` is a class, and creating a variable in your program that is a class is a bit more work and they are a story for a different day).

```
ArrayList listOfStudentNames<String> = new ArrayList<String>();
```

You might be thinking to yourself, "Self, that is sooooo much more work! I'm glad we only are learning about primitive data types for right now. So much less typing!"

### The `String` Class

But wait! There is a class we will be using! It's called a `String`, and it's really just a bunch of characters glued together. And since we use `Strings` so much in Java, there is a shortcut to creating one - you don't need to go through the hoops that you would for most other classes; you can just declare it like a primitive data type (although there are some under-the-hood things going on that you'll have to be careful of - we'll look at them later):

```
String name = "Walter White";
```

So it's totally legit to declare and assign a `String` as if it were a primitive data type, but if you really wanted to, you could follow the convention for declaring and assigning an object (this is actually called **instantiating** when dealing with objects, but that's also for another day):

```
String name = new String("Walter White");
```

Note that a `String` is denoted by using a `double` quote (whereas a `char` uses single quotes).

## Check Yourself

1. What is wrong with the following code?

```java
char name = "Colonel Mustard";
```

2. What is wrong with the following code?

```java
String name = 'Professor Plum';
```

## §3.4 Constants

### Constants

A **constant** is like a variable in that a name and type are identified, but unlike a variable, the value is hard-coded into the software. The constant (variable) is assigned and can never be changed. The naming convention is that all the letters are capitalized and words are glued together with an underscore. Additionally, the keyword **`final`** is used, which signifies a constant. Example:

```
final double PI = 3.14;
System.out.println("Circumference = " + (PI*10));
```

Any attempt for a program to modify a constant will result in explosions (i.e. errors):

```
final int NUMBER_OF_FINGERS = 5;
NUMBER_OF_FINGERS = 10;   // explosion… i.e.

Exception … java.lang.Error: Unresolved compilation problem:
  The final local variable NUMBER_OF_FINGERS cannot be assigned.
```

There are a few good reasons to use constants. Imagine we are writing an application that references a website where help guides and tutorials are present. Furthermore, let's imagine that this help center website is referenced from several different locations in the code. If we manually typed the URL in every section of the code, we open ourselves up to errors (what if we type it right eight out of nine times? Then there is the possibility someone clicks on the URL and it doesn't function properly). Another good reason that using a constant is because (and this URL example is perfect), we only have to change the value of it once. So if we have a variable named "`HELP_PAGE_URL`" and set it equal to "`http://www.stackoverflow.org`", we could easily update the URL by changing just that one line of code - everywhere that references "`HELP_PAGE_URL`" can stay the same!

```
public class HelpMethods {
  public static final String HELP_PAGE_URL = "http://www.stackoverflow.com";

  public static void main (String args[]) {
    System.out.println("For help, go to: " + HELP_PAGE_URL);
  }
}
```

## §3.5 Casting

### What is Casting?

**Casting** is necessary in strongly typed languages (like Java) because sometimes you just really want to treat a variable of one type as another type. One of the biggest reasons to do this is because of the funky results you get from integer division.

```
int num1 = 15;
int num2 = 6;
System.out.println(num1/num2);
```

You would expect the result to be `2.5` (because that's what 15 divided by 6 is!), but because both 15 and 6 are integers, the result is provided as an integer. So the computer will **truncate** the answer (that means it chops off the decimal and anything after the decimal). So the answer is really `2`. That's crazy, right?

#### How to Cast

There are a few different ways to cast. Let's look at the example of the two numbers above. The right way to do this would be to cast one (or both!) of the variables to be a double. To cast a variable to another type, you just need to parenthetically precede the variable with the type you want it to be. Note that this is a "one and done" thing - casting does not change the type of the variable permanently - it just bends that variable to your will for an instant.

Before considering this example, you need to know that in *mixed-mode arithmetic,* all variables involved are *automatically promoted* to the least restrictive type. So if we cast one of the `int` variables to a `double`, both of them will be considered a `double` and the answer will also be a `double`.

```
int num1 = 15;
int num2 =  6;
System.out.println((double)num1/num2);
```

The result here would be `2.5` because a `double` is involved in the computation. It's pretty neat that we didn't need to cast `num2` to also be a `double`. Of course, we *could* have cast both `num1` and `num2` to be `double`, but we don't need to.

So the right way to cast a variable is to use the parentheses before a variable that you want to treat as a different type. But it's quicker to do it the wrong way! Since it is true that introducing a `double` to an integer calculation will make the answer a `double`, why not just introduce a `double` literal into the equation? Since any number times 1 results in that number, this is valid:

```
int num1 = 15;
int num2 =  6;
System.out.println(1.0*num1/num2);
```

Sure, it’s not the “right” way to cast, but it works! One common mistake is to cast improperly. There is a logic error in this code:

```
int num1 = 15;
int num2 =  6;
System.out.println(1.0*(num1/num2));
```

This is because - thanks to the order of operations - `num1` is divided by `num2` (as integers, which results in `2`) and *then* the answer is cast as a `double`, so the programmer should have cast `num1` as a `double` instead.

In this example, recall that a variable of type `char` can have a value of any character in the UNICODE standard (like a "9", " ", or "ö"). But it’s possible to cast a `char` to an `int` to see the numerical equivalent.

```
char character = 'g';
System.out.println(character);
System.out.println((int)character);
```

The result of this program is:

```
g
103
```

This gets a little confusing because new programmers may try something like

```
System.out.println('g' + 'g');
```

hoping to get `gg` but instead they would get `206` (because the int equivalent of the char ‘g’ is 103, and we got two of them!).

Casting has a much bigger reach when we start talking about objects and classes, but that won’t happen for a while.

**Casting to Format**
There is a way to format numbers in Java, but you can accomplish the same thing without the `NumberFormat` class (although stackoverflow has a nice thread on this).

Let’s imagine we want to format a number to look like dollars and cents. Without the decimal formatter, we can come pretty close. This is true for all instances when you want to round or truncate to a certain decimal place.

Let’s say we have a variable of type `double` called `amount`, and we want to show the first two decimal places (although it has may have several figures past the decimal point). Let’s say the value of `amount` is `4.5678`.

```
amount = amount * 100; // 456.78
amount = (int) amount; // 456
amount = amount/100.0; // 4.56
```

Now, this was a fine example of how to truncate, but if you wanted to round, you could do that very easily with a call to `Math.round()`.

```
amount = amount * 100; // 456.78
amount = (int)Math.round(amount); // 457
amount = amount/100.0; // 4.57
```

## Check Yourself

1. What is the purpose of casting?

2. What are the results of the following casts?
   A. ```
      double height = 5.9;
      System.out.println((int)height);
      ```
   B. ```
      int days = 365;
      double weeks = (double)days / 7;
      ```
   C. ```
      int randomNum = 75;
      System.out.println((char)randomNum);
      ```
      ```

## §3.6 Order of Operations

### Order of Operations

**Precedence**
Just like real math, Java subscribes to the order of operations (remember PEMDAS?). There are a few tweaks in Java, but pretty much the precedence should seem familiar. One thing to note is that casting takes precedence over mathematical operators. So in an expression where there are mathematical operations (like multiplication or addition) and also a cast, the cast gets processed first.

After that, the rules are basically the same. The computer will process multiplying and dividing first (as it is encountered left to right), then addition and subtraction (as it is encountered left to right). You can always toss in parentheses if you want to explicitly add and subtract some numbers before multiplying - just like in real math!

The only other issue is the modulo operator (%). That has the same precedence as multiplication and division.

| OPERATOR | HOW TO PROCESS | LEVEL |
|---|---|---|
| `()` | Want something processed first? Use parentheses! | 4 |
| casting | This gets processed before math thingies do | 3 |
| `* / %` | In the order they occur, left to right | 2 |
| `+ -` | In the order they occur, left to right | 1 |

*Part of the precedence table in Java. There are many more operators we haven't learned about, so they're not in this table!*

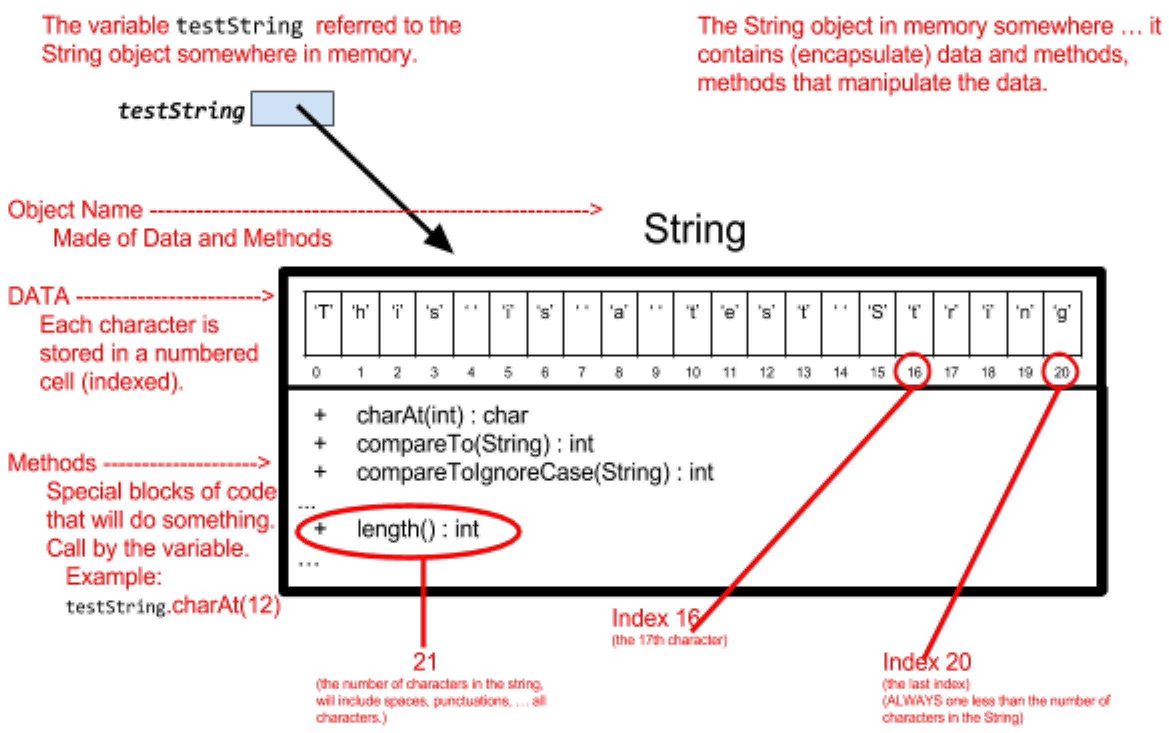
## §3.7 The `String` Class - an Object with methods

Before we get started here, you need to be well aware of what primitive data types are and what methods are. Recall the primitive data types like `short`, `int`, `long`, and `double` hold the actual value of the data. A methods is a block of code that can be called and executed by calling the name of the method. Methods are like mini-programs, or subprograms that can be ran. You have used method like `println` and `print`, and have actually created you own `main` methods (although you may not even know what that means until chapter 6).

We are not going to focus on Java classes and object for this chapter, however, you should recall that `String` data is not a primitive type of data. `Strings` are not primitive, but are rather complex; they hold data and methods to manipulate that data.

When `String` data is created, it takes up a larger space in memory that holds the characters that make up the string, and methods that can be used to manipulate the characters. You can think about the `String` variable like a bookmark (or favorite) to a website. The bookmark refers to a web page. The bookmark does not contain the web page itself, just the address of the web page. `String` variables are like that - the value of the variable is actually an address that refers to the larger spot in memory that contains the data. So the following block of code actually creates a variable `testString`, and a `String` object in memory with data and methods (see diagram)

```
String testString = "This is a test String";
```


The variable testString referred to the String object somewhere in memory.
The String object in memory somewhere … it contains (encapsulate) data and methods, methods that manipulate the data.
testString
Object Name ------------------------------------------------------->
Made of Data and Methods
String
DATA ------------------------->
Each character is stored in a numbered cell (indexed).
'T' 'h' 'i' 's' ' ' 'i' 's' ' ' 'a' ' ' 't' 'e' 's' 't' ' ' 'S' 't' 'r' 'i' 'n' 'g'
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Methods ---------------------->
Special blocks of code that will do something. Call by the variable.
Example:
testString.charAt(12)
+ charAt(int) : char
+ compareTo(String) : int
+ compareToIgnoreCase(String) : int
...
+ length() : int
...
21
(the number of characters in the string, will include spaces, punctuations, … all characters.)
Index 16
(the 17th character)
Index 20
(the last index)
(ALWAYS one less than the number of characters in the String)

Check out the the Relevant API in the Appendix for sample methods and examples of a few classes (including `String`).

## Check Yourself

1. Why are there multiple methods for `indexOf`?

2. How does a `String` (which is an *object*) differ from a variable with a primitive data type?

3. What is the difference between a `String` and a `char`?

4. Describe the difference between these two lines of code:

   ```
   System.out.println("" + 3 + 4);
   ```

   and

   ```
   System.out.println(3 + 4);
   ```

5. Consider the following code:

   ```
   public class stringQuestion01 {
     public static void main (String[] args) {
       String lyrics = "Row row row your boat.";
       int pos = lyrics.indexOf("row");
       System.out.print(lyrics.substring(pos));
     }
   }
   ```

   What is displayed on the screen when the program is run?

6. Consider the following code segment:

```
String str1 = "Happy ";
String str2 = str1;
str2 += "Birthday";
str2.substring(6);
System.out.println(str1 + str2);
```

What is the output when the code is executed?

7. Consider the following code segment:

```
String s1 = "FLCC";
String s2 = s1;
String s3 = s2;
```

After this code is executed, which of the following expressions would evaluate to `true`?

| | |
|---|---|
| I | `s1.equals(s3);` |
| II | `s1 == s2;` |
| III | `s1 == s3;` |

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and II

## §3.8 Important Classes

There are roughly 4,240 classes in the Java (Standard Edition, 8) language. Classes are mini-programs that you can leverage in your own code. Of these classes, there are only a handful that we will use this semester. Each class has it's own **methods** (or certain things each class can do). In the appendix entitled "Relevant API", you can dig deeper into the classes we'll be using and some of the methods that are useful in each class.

The classes we will use include `Array`, `Math`, `Random`, `Scanner`, and `String`. It will probably be a little easier to understand the distinction between classes and methods by looking at a specific class and examining a few of the methods in that class.

### `Math` Class

The Math class in Java has around one hundred methods. You can check them out here, but - spoiler alert - most of them are kind of boring. The commonplace methods include abs (for absolute value), round (for rounding), pow (for raising one number to the power of another), min (for finding the smallest number out of a few numbers), and other typical math functions. Below, consider this selection from the "Relevant API" which demonstrates some of the methods and how to use them:

**`Math.pow(double, double)`**

This is the real way to do exponents in Java. The first parameter is the base, and the second is the exponent. A call to `Math.pow()` requires two numbers - each a `double` - so if you pass in two variables of type `int`, the result will still be a `double`.

**`Math.random()`**

This method requires no parameters, and will return a `double` between `0` (inclusively) and `1` (exclusively). That means that you may get `0.0`, `.45023`, `.9596942`, `.9999999`, but you'll never get `1`. If you want a number that is `1` or bigger, you'll need to multiply by 10 (or 100, or 1000, etc.).

```
System.out.println(Math.random());
// 0.6017376710307702
```

The above example will generate a random `double` greater than or equal to `0`, but less than `1`.

```
System.out.println(Math.random()*10);
// 7.1687284712872543
```

The above example will generate a random `double` greater than or equal to `0`, but less than `1`. After that, the number will be multiplied by 10 to make it bigger - the result is that you'll get a number between `0` and `9.999999999999999`...

```
System.out.println((int)(Math.random()*10));
// 3
```

The above example will generate a random `double` greater than or equal to 0, but less than 1. After that, the number will be multiplied by 10 to make it bigger - and then it will be cast as an `int`. So the smallest the result will be is `0`, and the largest is `9`.

```
System.out.println((int)(Math.random()*10) + 1);
// 10
```

The above example will generate a random `double` greater than or equal to 0, but less than 1. After that, the number will be multiplied by 10 to make it bigger - and then it will be cast as an `int`. And then it will have `1` added to it, so the smallest number is `1` and the largest it can be is `10`.

## §3.9 Basic Input

### Getting Basic Input from the User

One of the most elemental tasks in software design is getting input from the user. The walkthrough below may seem complicated, but after a little while, this will be second nature to you. The sample is provided in its entirety, and then broken down afterwards:

#### Code for Input

```
Scanner scanner = new Scanner(System.in);
System.out.print("What is your age? ");
int age;
age = scanner.nextInt();

System.out.println("Your age is: " + age);
```

#### Walkthrough

```
Scanner scanner = new Scanner(System.in);
```

This line of code creates an instance of the Scanner class. You'll need this class if you want to gather input from the user. It's just the way Java is. The syntax is daunting, but it makes sense down the road.

```
System.out.print("What is your age? ");
```

This is called "prompting the user". Here we are outputting something to the user and asking them to type something in.

```
int age;
```

Here we are merely declaring a variable (that can hold integers). We've named it "age", and we intend to somehow take whatever the user has entered and put that value into `age`.

```
age = scanner.nextInt();
```

And here's where that happens! We are asking the `Scanner` that we created (named "scanner") to use the `nextInt` method to store whatever has been typed right into the variable `age`. Don't forget that `Scanner` is one of those 4,240 classes that come with

Java, and the `nextInt` method is a specific method to `Scanner`. If we were asking the user for a piece of information that is of type `double`, we would have used the `nextDouble()` method in `Scanner`. Or `nextByte()` if we were looking for a `byte`.

```
System.out.println("Your age is: " + age);
```

Simple output here. We are literally outputting “Your age is: “ and then outputting the value that has been stored in `age`.

# INTRODUCTION TO SELECTIONS

*"Live by the curly brace, die by the curly brace."*

**- Dave Ghidiu -**

## §4.1 Boolean Logic

### Relational Operators and `if` Statements

The most fundamental part of programming (other than knowing the syntax and language) is reasoning. Learning a computer language is very similar to learning a foreign language - you can kinda-sorta-maybe get your point across if you know the words, but until you can elegantly stitch together a coherent thought, you aren't going to get much done.

Up until this point, we've really only looked at computations and calculations - we haven't had the computer do what it does best - make decisions. A quick primer on Boolean Logic will help frame a conversation on `if` statements and other selections.

There are only a handful of **relational operators** (things you can use to make a decision). Just like the arithmetic operators (`+`, `-`, `*`, `/`, and `%`), relational operators are used to compare two things (usually numbers). It is important to note that these relational operators will only work with primitive data types. There are different ways for comparing the values of objects such as a `String` or `ArrayList` - that will be covered much later. For now, you can get by knowing that if you are comparing any two objects then you will need to use `equals()` to see if they are equal, and use `compareTo()` to see which one is greater. But again, that's a story for a different time.

#### Equals

The first reaction is to use the "=" if we want to compare to numbers, but that won't work in Java (recall that the equal sign is used to *assign* values to a variable). Instead, we use two of them, jammed right next to each other:

`==`

Happily, the equals operator is easy to use. You just stick it between to numbers to see if those numbers are equal, and the computer will return either `true` or `false` (it won't spit out the difference between them).

```java
int i = 10;
int j = 150/15;

if (i == j) {
  ... // Some special, sweet code
}

... // Other normal, boring code
```

In this case, the special, sweet code between the two brackets would be executed because the value of `i` does, indeed, equal the value of `j`. Also, the other, normal, boring code would be executed because that code is in the main program and will run regardless of the `if` statement (by the way, we'll look more at `if` statements in a few short moments - all you need to know now is that if the part in parentheses is `true`, then the code between the brackets after the `if` statement will be run). But let's look at what happens when the two variables don't equal each other:

```
int i = 20;
int j = 150/15;

if (i == j) {
  ... // Some special, sweet code
}

... // Other normal, boring code
```

Well, in this case, the special, sweet code is *not* executed, although the normal, boring code will still be run - that code resides in the main program and is business as usual.

**Not Equal**
Just as the == means equal, Java needs a way to determine if two values are not equal.

`!=`

Syntactically, this is used in the same manner as an equals operator above:

```
int i = 20;
int j = 150/15;

if (i != j) {
  ... // Some special, sweet code
}

... // Other normal, boring code
```

In this example, the special, sweet code will be run because `i` has a value of `20` and `j` has a value of `10`, so they are not equal and `i != j` is `true`. Pro-tip: In computer nerdom land, they refer to the "`!`" as a "bang".

**Greater Than**
If you wish to see if one number is greater than another number, use:

`>`

This will return `true` if the first number is greater than the second, `false` otherwise.

```
int i = 200;
int j = 40;

if (i > j) {
  ... // Some special, sweet code
}

... // Other normal, boring code
```

In this code, since the value of `i` is `200` and the value of `j` is only `40`, the special, sweet code is executed. However, check out the following code:

```
int i = 200;
int j = 40;

if (i > j * 5) {
  ... // Some special, sweet code
}

... // Other normal, boring code
```

In this case, the special, sweet code will not run because the value of `i` is `200` and the value of `j` times `5` is also `200`. Technically, `i` is not greater than `j`, so the code block in the `if` statement brackets does not run.

**Greater Than or Equal To**
In math, we use to ≥ denote a "greater than or equal to". However, that key doesn't exist on a keyboard. So we use

`>=`

It's pretty darn similar to the greater than operator, but it also returns true if the two values are equal, as well.

```
int i = 200;
int j = 40;

if (i > j * 5) {
  ... // Some special, sweet code
}

... // Other normal, boring code
```

This would return `true` because both values (`i` and `j * 5`) evaluate to 200.

**Less Than**
This is used just like the greater than operator, except it sees if the first number is less than the second number.

`<`

Consider this example:

```
int i = 200;
int j = 400;

if (i < j) {
  ... // Some special, sweet code
}

... // Other normal, boring code
```

Because `200` is less than `400`, the conditional is true and the special, sweet code will run.

**Less Than or Equal To**
If you've made it this far, you probably know how this is going to play out. The symbol for less than or equal to is:

`<=`

And it returns `true` if the expression on the left side evaluates to a value smaller than the one on the right side.

```
int i = 200;
int j = -10;

if (i <= j) {
  ... // Some special, sweet code
}

... // Other normal, boring code
```

Boo! The special, sweet code is not executed and we never get to see what happens! The program just goes to the normal, boring code.

## Boolean Operators

Boolean operators are a little less intuitive than the relational operators (mostly because we deal with relational operators all the time, and have been formally trained on them in school). On the other hand, Boolean operators - while intuitive - may require a bit more thinking.

If you were fortunate enough to take a basic logic class (it used to be in the curriculum for high school math), you may remember the words not, and, or, and maybe even exclusive or.

In Java, we are really concerned with only three of the operators

**NOT**
The NOT operator is also known as a "logical inverter". We use a "bang" to denote the NOT symbol (in symbolic logic, it would be a "~").

`!`

This changes the value of any boolean value. It's a toggle switch. So if a statement evaluates to `true` and you jam a bang in front of it then the statement becomes `false`. And vice versa.

```
int i = 10;
int j = 150/15;

if ( !(i == j) ) {
  ... // Some special, sweet code
}

... // Other normal, boring code
```

So in this case, the conditional `(i == j)` would be `true`. However, since there is a NOT in front of the expression, (`!(i == j))`, the conditional actually evaluates to `false`. It is mildly interesting that you could put another bang in front of that original bang to reinstate the statement to `true`. And again to toggle it back to `false`. Of course, in this particular example, a savvy programmer would probably have just used the logical inverter:

```
int i = 10;
int j = 150/15;

if (i != j) {
  ... // Some special, sweet code
}

... // Other normal, boring code
```

**AND**
This is pretty much what it sounds like. For an expression to be `true`, both parts (the first AND the second) need to be `true`. We like to use the ampersand to represent AND. In fact, we like it so much, we use two in a row:

`&&`

For an AND statement to be `true`, *both* parts need to evaluate to `true`.

```
int i = 20;
int j = 31;

if ((i >= 20) &&  (j < 100)) {
  ... // Some special, sweet code
}

... // Other normal, boring code
```

In the code above, since `i >= 20` is `true` and `j < 100` is `true`, then both parts of the conditional are `true`, and the code in the `if` statement is executed.

```
int i = 20;
int j = 31;

if ((i > 50) && (j != 40)) {
  ... // Some special, sweet code
}

... // Other normal, boring code
```

In the segment above, the code in the `if` statement block is not executed because the conditional in the `if` statement does not evaluate to `true`. While `j != 40` is indeed `true`, it doesn't matter because `i` is not greater than `50`. At this point, the computer knows that the conditional in the `if` statement is `false` and the code in the brackets is not run.

This is known as "**short circuit evaluation**". Since Java looked at the `(i > 50)` expression first and determined it was `false`, the computer doesn't even need to look at the second clause because it doesn't matter - the `if` statement conditional is `false` and the program skips the code block in the brackets. For the record, the code in the second clause could even cause a runtime error, and in this case the software wouldn't crash.

**OR**
For an OR statement to be `true`, only *one* of the components need to be true - but it is okay if they both are `true`. The character that represents OR in Java is the "pipe". That's the vertical line that usually is on the key by the "Enter" key.

||

Let's look at some examples:

```
int i = 20;
int j = 31;

if ((i > 10) || (j != 40)) {
  ... // Some special, sweet code
}

... // Other normal, boring code
```

This one is a no-brainer - both conditions are true, so the special, sweet code is executed. Short circuit evaluation happens here, too - since `i > 10` is true, the computer doesn't even bother to look at `j != 40` because the OR statement is already `true`.

## Check Yourself

1. Describe what each of these symbols mean:
    a. `==`
    b. `!`
    c. `!=`
    d. `<`
    e. `<=`
    f. `>`
    g. `>=`
    h. `&&`
    i. `||`

2. What are the truth values of the following expressions?
    a. `4 > 44`
    b. `(3 + 4) <= 8`
    c. `(7 * 5) % 2 == 1`
    d. `(4 >= 2) && (4 <= 8)`
    e. `!true`
    f. `!false`
    g. `(8/2 == 4) || false`

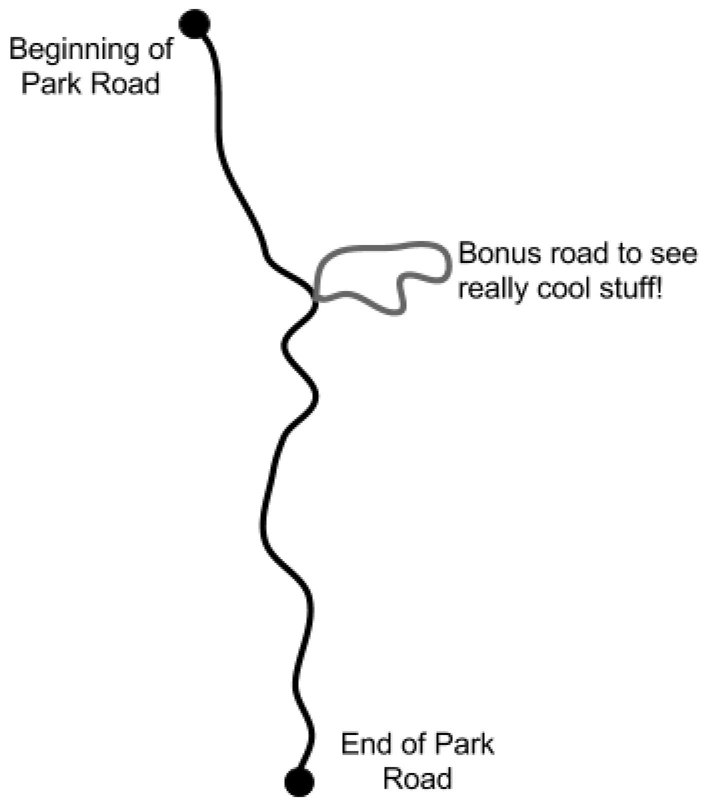
3. Describe short-circuit evaluation, and why it is so important.

## §4.2 `if` Statements

### Conditional Statements

**`if` Statement**
An `if` statement is what allows programs to make decisions, and take bonus pathways. They're pretty much what make computers cool. Not to undermine your work so far, but making software that has predetermined outcomes isn't the epitome of programming!

Think of an `if` statement as a scenic overlook on the highway. Suppose you're driving through Yellowstone National Park. Well, there's really only one road that goes through the park, and you have to drive the length of it. But there are little bonus roads you can take along the way; you don't have to, and even if you do, you'll *still* have to drive the *entire road* to the end. But *if* you do take these bonus roads, you may see some cool things.



That's what an `if` statement is. It's an optional block of code that may get executed. All `if` statements have a **condition**, and if that condition evaluates to `true`, then a whole bunch of things can happen. If the condition is `false`, then the code in between the `if` statement brackets won't execute. It's one or the other - there is no room for ambiguity. Here's a good example:

```
System.out.print("Enter a number: ");
int num = scanner.nextInt();

if (num > 5) {
      System.out.println("Greater than five!");
}

System.out.println("Game over.");
```

In this example, if the user punched "6" into the computer, then the condition `num > 5` (which is really a question)  would return `true`, and the code in the brackets would be run. If not, then the code in the brackets would be ignored. In either case, “`Game over.`” would be output because it’s just plain ole’ code, not linked to an `if` statement.

Anything can be in the condition, so long as the stuff you jam in there can be evaluated to `true` or `false`. Good thing you know about Boolean Logic! So the following are valid examples of the condition:

```
if (num > 5) {
      // code to run if num is greater than 5
}

if (num <= 10) {
      // code to run if num is less than or equal to 10
}

if (num == 15) {
      // code to run if num precisely equals 15
}

if (num != 8) {
      // code to run if num does not equal 8
}

if (!(num > 88)) {
      // code to run if num is not greater than 88
}

if (firstName.equals("Dave")) {
      // code to run if firstName equals "Dave"
      // Note that a String can't use == to test for equality
      // because a String isn’t a primitive data type - it’s an
      // object!
}

if (num % 2 == 0) {
      // code to run if num is even
```

```
        // This is CLASSIC computer nerd code!
}

boolean isInstalled = true;
if (isInstalled == true) {
        // code to run if isInstalled is true
}


boolean isInstalled = true;
if (isInstalled) {
        // code to run if isInstalled is true
}
```

A few things. First, recall that we use == to test to see if a variable is equal to a value (or another variable). This only works with primitive data types; it doesn't work with objects. So if you are attempting to compare an `int`, `double`, `boolean`, or `char` (as well as the other primitives), you can use the ==. All other things - including `String` - require you to use the `.equals()` *method*. A method is like a small, self-contained program that you'll read about later.

The other thing that is worth mentioning is the funny business with the difference between

```
if (isInstalled == true)
```

*vs.*

```
if (isInstalled)
```

These both do the same exact thing. The only difference is that the second option is more concise. There is no need to compare the variable to `true` or `false` - it happens automatically.

## Common Errors

### Equality

```
if (firstName == "Dave") {
        // some code here
}
```

The problem here is that we can't use == to compare the value of a `String`. Sure, it might work in some IDEs. But that's due to an optimization - it's not accurate all the time. Different compilers may handle this code differently. Don't believe me? Sooner or later you'll be using == to compare the values of a `String` and you'll get wonky results. On that day, you'll thank. By the way - the correct comparison (as we've learned earlier) should be `if (firstName.equals("Dave"))`.

**Assignment**

```
if (num = 10) {
      // some code here
}
```

In this example, we probably meant to see if `num` was equal to `10`, but recall that the single equal sign really assigns a value to the variable. So we've inadvertently set the value of `num` to `10`. By the way, this code will compile and run, and will most likely result in a logical error somewhere.

**Semicolon**

```
if (num == 10); {
      // some code here
}
```

Ahh! The classic semicolon error (in reverse!). Here, the semicolon signifies the end of the `if` statement, so the code that we *think* is contingent on the condition (because it is in the brackets) is actually not part of the `if` statement. The code in the brackets will run regardless of the condition, because the computer sees it as regular old code that is part of the program.

**Braces**

```
if (num == 10)
      // some code here
      // some more code here
```

Okay, so, this actually may not always be an error. Sometimes. You see, you technically only need the curly braces for an `if` statement if there is more than one line of code that should be executed if the condition is `true`. So as long as `// some code here` refers to only one line of code, everything is great. But the minute there is more than one line of code, the program will not work as intended. In the example above, `// some more code here` will be run regardless of the state of the condition. Even though the second line of code is indented and *looks* like it belongs to the `if` statement, it does not. This code is identical to the following (therefore, it's *always* good to get in the habit of using brackets with `if` statements - even if there is only one line of code involved):

```
if (num == 10) {
      // some code here
}
```

```
        // some more code here
```

## `if-else` Statements

A twist on the `if` statement is an `if-else` statement. In this structure, the condition is tested. If the condition is found to be `true`, then the code in the brackets is executed. But if the condition is `false`, then code written in the `else` part of the block will be executed.

```
if (num %2 == 0) {
      System.out.println("Even number.");
} else {
      System.out.println("Odd number.");
}
```

Looking at this code, let’s say `num` is `8`. Since `8` mod `2` is `0`, the first code block is run and “`Even number.`” is output on the screen. But if `num` was `9`, and `9` mod `2` is `1` (which is not `0`), then “`Odd number.`” is printed in the console window.Unlike an `if` statement (where something may happen or it may not), in an `if-else` statement, one of the the two outcomes will always necessarily happen.

Let’s look at another example:

```
Scanner scanner = new Scanner(System.in);
System.out.print("Enter a number: ");
int num = scanner.nextInt();

if ((num * 2) > 50) {
  System.out.println("Nice!");
} else {
  System.out.println("Not nice.");
}
```

In this example, we can expect one of two outcomes - either the word “Nice” will be output on the screen, or the phrase “Not nice.” In this case, the user would have to enter a number bigger than 25 to get the word “Nice”.

## Check Yourself

1. Decide if the following code segments will output “YES” or “NO”:

   a.
   ```
   int a = 10;
   if (a / 5 == 3) {
     System.out.println("YES");
   } else {
     System.out.println("NO");
   ```

   b.
   ```
   boolean b = true;
   if (b) {
     System.out.println("YES");
   } else {
     System.out.println("NO");
   ```

   c.
   ```
   char c = 'a';
   if (c > 'A') {
     System.out.println("YES");
   } else {
     System.out.println("NO");
   }
   ```

   d.
   ```
   double d = 55.5
   if (d*10 > 555) {
     System.out.println("YES");
   } else {
     System.out.println("NO");
   ```

   e.
   ```
   int e = 10;
   int f = 20;
   if ((e > 30) || (f > e)) {
     System.out.println("YES");
   } else {
     System.out.println("NO");
   ```

2. What is the difference between an `if` statement and an `if-else` statement?

3. Write a small program that generates a random number (using the `Math.random()` method), and if the result is less than `.5`, display “Heads”. Otherwise, display “Tails”.

# COMPLEX SELECTIONS

"The 'what-ifs' and 'should-haves' will eat your brain"

**- John O'Callaghan -**

## §5.1 Multiple Conditionals

### Multiple `if-else` Constructs

So, things can get a little wacky here. In a regular `if-else` statement, there are exactly two outcomes. But sometimes you want more than two choices. It's possible to use multiple `if-else` statement during these times. Keep in mind that, just like an `if-else` statement, only one outcome is possible.

```
System.out.print("Enter grade: ");
String letterGrade;
int grade = scanner.nextInt();

if (grade > 90) {
  letterGrade = "A";
} else if (grade > 80) {
  letterGrade = "B";
} else if (grade > 70) {
  letterGrade = "C";
} else if (grade > 65) {
  letterGrade = "D";
} else {
  letterGrade = "F";
}

System.out.println("The grade is: " + letterGrade);
```

In this case, the program will assign a value to `letterGrade` based on the value of `grade`. Think of this as a sieve - if grade is `72`, then the first `if` statement fails, so does the second. But the condition in the third `if` statement is `true`, so `letterGrade` is assigned a value of "`C`", and then the `if`-block stops and moves on to the next line of code (in this case, a `System.out.print()` statement). Note that there are logical mistakes that can be made. Consider the following code (which is just the upside down version of the code we just looked at):

```
String letterGrade = "";

if (grade > 65) {
  letterGrade = "D";
} else if (grade > 70) {
  letterGrade = "C";
} else if (grade > 80) {
  letterGrade = "B";
} else if (grade > 90) {
  letterGrade = "A";
} else {
  letterGrade = "F";
}

System.out.println("The grade is: " + letterGrade);
```

The problem here - even though everything might look OK - is that because of the way the `if` statements are structured, the user will most likely get erroneous results. Let's consider the case of the `72` again. We know it should be a "`C`", but in this code, a "`D`" is returned! That's because the first test - is `grade` greater than `65` - is `true`. And because an `if-else` block can only have one outcome, the first outcome that is `true` is returned. That means that all grades above a `65` will earn a "`D`", and anything `65` or less would result in an "`F`". This is a pretty common logic error, and it's why I prefer to deal with `switch` statements (when possible). If you can't use a `switch`, you'll just need to be extra careful and test the code thoroughly.

The other kernel of wisdom in this code snippet resides in the first line - assigning (or initializing) the value of `letterGrade`. It's always good to assign a value to a variable before that variable is tested in some `if` statements. Even though we know that once the `if-else` block is done, `letterGrade` should have a value, there might be times when we *think* this is true but it really isn't. So if the whole `if-else` block executes and `letterGrade` *still* doesn't have a value, when the last line runs and tries to output the value of `letterGrade`, bad things will happen. Like, your software will crash (more likely, it won't even compile). If you get a "**`local variable may not have been initialized`**" error, it's because there is a chance that the variable will never get a value assigned to it.

## Check Yourself

1. Is there a difference between a whole bunch of `if` statements and one construct that is a number of `if-else` statements?

2. Consider this code segment:

```
if (age > 62) {
  status = "retire";
} else if (age >= 40) {
  Status = "over the hill";
} else if (age >= 35) {
  status = "run for president";
} else if (age > 24) {
  status = "rent a car";
} else if (age >= 18) {
  status = "lotto";
}
```

What is the output for each of these values of `age`?

A. `70`
B. `61`
C. `24`
D. `35`
E. `36`
F. `34`
G. `17`
H. `-1`

3. There are issues with each of the following lines of code. Find them, and describe them.

   A.
```
int num = 10:
if (num > 15); {
  System.out.println("Bigger than 15");
}
```

   B.
```
Scanner scanner = next Scanner(System.in);
System.out.print("Enter a name: ");
String choice = scanner.nextLine();

if (choice == "The Rock") {
  System.out.println("Can you smell what The Rock is cooking?!?");
}
```

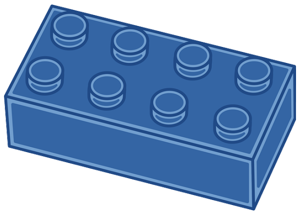
   C.
```
double gpa = 2.49;
if (gpa > 3.49) {
  letterGrade = 'A';
}; else if (gpa > 2.99) {
  letterGrade = 'A-';
}; else {
  letterGrade = 'B';
}
```

## §5.2 Compound Conditionals

A **compound conditional** is nothing more than an if statement that tests multiple conditions.

### And

A conditional can have more than one component - for instance, a Lego sorting machine may be concerned about not only the width of a Lego brick, but the length too:



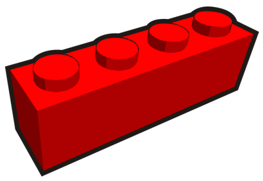
In this case, the Lego sorting algorithm may do a preliminary sort and want to consider any brick that is at least two rows wide and four columns long. The key concept is *and* - both conditions must be true. The code may look like this:

```
if (brickWidth >= 2 && brickLength >= 4) {
      // do something cool
}
```

For clarity, engineers may include each individual condition in parentheses. This is a good habit to get into as it will most certainly prevent errors later on:

```
if ((brickWidth >= 2) && (brickLength >= 4)) {
      // do something cool
}
```

Note that one of the optimizations that Java compilers make at runtime is **short-circuit evaluation**. In the case of `&&`, the compiler will look at the first conditional. If the test fails, the compiler knows that even if the second part is true the code isn't going to execute. So why bother wasting time to see what the second part is?



```
if ((brickWidth >= 2) && (brickLength >= 4)) {
      // do something cool
}
```

Looking at the code, it doesn't really matter what the value of `brickLength` is because the totality of the `if` statement will be `false` in this case. In fact, the second condition may even include code that would cause a runtime error, but the compiler will not fault because the second condition doesn't even get considered:

```
int brickwidth = 1;
if ((brickWidth >= 2) && (5/0 > 1)) {
      // do something cool
}
```

Eventhough `5/0` should throw an error, the program works fine because that specific piece of code is never evaluated.

## Or

An *or* statement is represented by the symbol ||. Software engineers refer to this symbol, the vertical line above the "Enter" key, a "pipe". The or statement requires that at least one of the conditions be true - but if they both are, that's cool too!

Going back to the Lego sorting example, let's imagine that the sorter now only cares about colors of bricks, and at this stage the algorithm should be sorting out bricks that are either red or bricks that are yellow.

```
if ((brickColor.equals("red") || brickColor.equals("yellow") {
      // do something cool
}
```

Note that short circuit evaluation applies with `or` statements, too. So if a brick is evaluated and it is red, then the second part of the conditional is not even tested.

## Ands and Ors

It is possible to mix and match `and` with `or` statements. It is critical to use parentheses when doing this because not only will it make for clearer code, but the precedence of && is greater than that of ||, so without parentheses, it is very easy to make a mistake. The same rules apply; && statements are contingent on both components being `true`, whereas an || statement only needs one to be `true`.

```
if (((a > 10) && (b < 50)) || (c == 30))   {
      // do something cool
}
```

The rules for &&s and ||s still hold - an AND statement requires both parts to be `true`, whereas an OR statement only needs one to be `true`. But using parenthesis is even more critical here because && takes precedence over `||.` So in the next two code segments, the output would be different based on nothing more than parenthesis:

```
if (A || B && C)
```

In the above example, the computer first sees if `B` and `C` are both `true`. If they are, then it will see if `A` is `true`.

```
if ((A || B) && C)
```

In the above example, the computer first sees if either `A` or `B` are `true`. If one of them is, then it will see if C is `true`.

## Other Operators

There are a few other operators that are not relevant right now, but may be handy down the road. Look for the ^ bitwise operator, as well as & and | (all of which are described here).

### !

The exclamation point (or "bang" as it is known in the biz) means "not". It flips the truth value of any boolean value (which includes the result of a logical test). Consider this code:

```
boolean a = true;
System.out.println(a + " " + !a);
```

The output would be:

```
true false
```

This is handy in a few different situations. Most notably, this is great for toggling the truth value of a boolean. A pervasive example is seen in websites - the code to expand or collapse some content.

Simple Collapsible
Click on the button to toggle between showing and hiding content.
Simple collapsible

In this case, there is probably a variable called `isVisible` and it is initially set to `true`. When the button is clicked, the program has to flip the state of `isVisible`. There are two ways to do this:

```
isVisible = !isVisible;
```

or

```
if (isVisible == true) {
      isVisible = false;
} else {
      isVisible = true;
}
```

Clearly, the first example is preferred.

## DeMorgan's Law

So there is this odd behavior when negating compound conditionals. If you've studied logic, then you are probably familiar with DeMorgan's Law (and if you are a fan of the Showtime series Dexter, Dexter Morgan and Deborah Morgan are both nods to DeMorgan's Law, as this post suggests).

Essentially, the law dictates that if we negate an AND statement, we are really looking at negating both components and replacing the AND with an OR:

Given

```
boolean a = true;
boolean b = false;
```

Then the following two equivalences are true:

`!(a && b)` is equivalent to `(!a || !b)`

`and`

`!(a || b)` is equivalent to `(!a && !b)`

All this talk about DeMorgan's Law may seem in the weeds a bit, but you may find that a logical error manifests in your code because of this!

## Check Yourself

1. What is the output of this code?

```
boolean a = true, b = true, c = false, d = true;

if (a || b && c || d) {
  System.out.println("Who lives in a pineapple under the sea?");
} else {
  System.out.println("SpongeBob!");
}
```

2. Describe the significance of this code segment:

```
boolean isLightOn = false;
isLightOn = !isLightOn;
```

3. Write down statements that are logically equivalent to the following:

   A. `!(a && b)`
   B. `!(c || d)`
   C. `!(!e && f)`
   D. `!(g || !h)`
   E. `!(!i && !j)`

## §5.3 `switch` Statements

### `switch` Statement

A `switch` statement is a more readable version of `if-else` statements - and slightly more restrictive. In general, `switch` statements are more readable to humans and provide a logical structure for program decisions. Unlike `if` statements, they cannot easily process a range (for instance, if the value of `grade` is greater than `79.49` but less than `89.5`).

Moreover a `switch` statement only works with variables of type `byte`, `short`, `char`, or `int`. A `switch` can also work with a `String` and a few other special cases, though nothing we'll examine here. Don't forget that with `Strings`, we can't use the `==` relational operator, but that's okay because Java handles all the comparing in a `switch` statement.

The following code demonstrates the implementation and visual simplicity of a `switch` statement:

```
int dayOfWeek = 5;
String day = "";

switch(dayOfWeek) {
      case 1:     day = "Monday";
                  break;
      case 2:     day = "Tuesday";
                  break;
      case 3:     day = "Wednesday";
                  break;
      case 4:     day = "Thursday";
                  break;
      case 5:     day = "Friday";
                  break;
      case 6:     day = "Saturday";
                  break;
      case 7:     day = "Sunday";
                  break;
      default:    day = "no day";
                  break;
}

System.out.println("The day is " + day);
```

```
The day is Friday
```

There are two things to notice in the example above: the `break` statement and the `default` statement. There's more about `break` in the next section, but let's look at `default`. The code for `default` gets used only when the variable that is being switched does not fall into any of the cases. In our example about days of the week, this means that if the user entered 8 as input, then the computer would compare that input against all the cases. Since there isn't a `case` for 8, then the `default` code is executed.

## Falling Through

A `break` statement is what tells the computer to stop examining the code in the `switch` statement. Most likely, we'll see a `break` statement after every `case`. But that isn't always going to happen. Let's look at Kickstarter as a real world example.

If you've never seen Kickstarter, it's a website where entrepreneurs can list their product and visitors to the website can decide to "back" the project by giving some money. Typically, there are different levels. Let's imagine a Kickstarter for a board game called The Cones of Dunshire. There could be several different levels of giving:

| | |
|---|---|
| $5 | Bumper sticker |
| $15 | T-Shirt |
| $50 | One copy of Cones of Dunshire |
| $90 | Ledgerman Hat |
| $150 | Figurine collection (Arbiter, Wizard, Maverick, Warrior, and Corporal) |

In Kickstarter, usually if a giver backs at a certain level, they will get the reward at that level *and all the previous levels*. In Java, one way to accomplish this is to omit the `break` statements and letting the code fall through:

```
int amount = scanner.nextInt(); // Get input from user
String reward = "";

switch (amount) {
      case 150:   reward += "figurines";
      case 90:    reward += " hat";
      case 50:    reward += " game";
      case 15:    reward += " shirt";
      case 5:     reward += " sticker";
                  break;
      default:    reward = "none";
}

System.out.println("You will receive: " + reward;
```

Because there is no break statement, all the cases after the entry point are executed. In the Cones of Dunshire example, let's imagine the backer pledged $50. Then the `switch` statement will not execute the code for the case 150 and case 90, but *will* execute case 50. After that, the subsequent lines of code will be executed (until a break is encountered).

```
You will receive:  game shirt sticker
```

This is known as "**falling through**", and many newer programmers encounter this by accident (omitting the break statements accidentally).

## Multiple Case Labels

Occasionally, there could be a time when multiple case labels should behave the same way. In these cases, it is preferred to consolidate them:

```
int num = scanner.nextInt(); // Get a number from the user

switch (num) {
      case 2:
      case 3:
      case 5:
      case 7:     System.out.println("Prime number!");
                  break;
      case 4:
      case 6:
      case 8:
      case 10:    System.out.println("Composite!");
                  break;
}
```

This is really nothing more than a perversion of falling through, but it looks pretty nice and easy to read.

## Check Yourself

1. Describe what the notion of falling through is, and compare it to `if` statements and `if-else` statements.

## §5.4 Scope

### Defining Scope

**Scope** refers to the visibility of a variable. The ability to recall a variable is granted exclusively to methods (loops, statements) where the variable lives. For instance, consider this code:

```
int num = 10;
if (num % 2 == 0) {
  String remainder = "even";
}
System.out.println("The number is " + remainder);
```

In executing this code, we might expect that the output in the console would be:

```
The number is even
```

But in reality, there is an error (probably a **`cannot find symbol`** error)! This is because the call to the variable `remainder` is outside the `if` statement where `remainder` was declared. Because of scope, `remainder` is only available within the `if` statement. So to ameliorate the issue - and to write better code - it's a better idea to declare the variable before the `if` statement:

```
int num = 10;
String remainder;
if (num % 2 == 0) {
  remainder = "even";
}
System.out.println("The number is " + remainder);
```

In this example, we are getting closer! Conceptually, this should work (because the scope of `remainder` isn't an issue). Again, the output should be what we want because the program can access the variable remainder (because it was declared outside of the `if` statement):

```
The number is even
```

But we still have a problem… even though we think the code should work - it doesn't. We will most likely get an error (specifically, **`variable remainder may not have been initialized`**). This doesn't make sense because we *know* that if `num` is `10`, then the `if` statement

```
if (num % 2 == 0)
```

fires, and `remainder` takes on a value - `"even"`. But the computer doesn't necessarily see things this way. Because as written, the code opens up the possibility of `remainder` not having a value. Consider if `num` was `11`. In that case, the `if` statement doesn't trigger, and we are left trying to output a variable that does not have any state!

The best way to avoid this is to declare the variable and load it with a dummy value:

```
int num = 10;
String remainder = "";
if (num % 2 == 0) {
  remainder = "even";
}
System.out.println("The number is " + remainder);
```

This code is perfect! The variable remainder is scoped such that any code that needs access to remainder can see it, and it is assigned a dummy value (alternatively, we could have ensured that remainder has a value before it is output, as in the following code).:

```
int num = 10;
String remainder;
if (num % 2 == 0) {
  remainder = "even";
} else {
  remainder = "false";
}
System.out.println("The number is " + remainder);
```

Either of these solutions is acceptable, but the example immediately above is preferred because `remainder` will always have a dependable value *because we coded every possibility*.

An easy way to remember scope is that variables live and die in the brackets they are born in. For instance in the following code, `probability` cannot be referenced outside of the `if` statement:

```
if (headsOrTails.equals("heads")) {
  double probability = .5;
}

System.out.println(probability); // error!
```

This means that it's completely legal to have two (or more!) different variables with the same name! But that is not a good idea. At all.

## Local and Global Variables

A global variable can be referenced from anywhere in the program. It is declared at the top level of a program - outside from any methods.

```
public class Gradebook {

  public int numOfStudents = 30;

  public static void main (String[] args) {
    System.out.println(numOfStudents);
    int totalPoints = 1200;
    System.out.println(computeAvg(totalPoints));
  }

  public static double computeAvg (int points) {
    return points/numOfStudents;
      // 40
      // Great success!
  }
}
```

In this example, it is okay to reference `numOfStudents` from the `computeAvg` method because `numOfStudents` is visible to all methods. That's the beauty of a global variable! As an aside, while there may be times to use global variables, you should try to avoid them if possible. They can cause confusion down the road, and unless there are compelling reasons to use global variables, you should avoid them.

One of the pitfalls of global variables is that there can be confusion with local variables that have the same name. Let's look at the same code, but this time let's look at the method `computeAvg` a bit closer (the change is highlighted):

```
public class Gradebook {

      int numOfStudents = 30;

      public static void main (String[] args) {

            System.out.println(numOfStudents);
            int totalPoints = 1200;
            System.out.println(computeAvg(totalPoints));

      }

      public static double computeAvg (int points) {

            int numOfStudents = 10;
            return points/numOfStudents;
```

```
                // 120
                // Great success?

            }
        }
```

In the event of a collision, the local variable will always trump the global variable.

We'll talk more about scope in the next chapter.

## Check Yourself

1. What do you think will happen when this code is executed?

```
int num = 10;
int num2 = 20;

if (num > 5) {
  int num2 = 30;
  System.out.println(num2);
}

System.out.println(num2);
```

2. What do you think will happen when this code is executed?

```
for (int i = 0; i < 10; i++) {
  System.out.println(i + " ");
}

for (int i = 0; i < 10; i++) {
  System.out.println(2*i + " ");
}
```

3. In your own words, describe 'scope'.

# METHODS

*"Though this be madness, yet there is method in it."*

**- William Shakespeare -**

## §6.1 Why use Methods?

So far, the programs studied in this book have been about practicing the fundamentals so they are short programs. Typical programs like phone apps, word processors, web browsers, and games require thousands of lines of code. Managing a program like that requires breaking the larger functionality of a program into smaller functions. Java offers an organized way to contain code in modules with a single purpose: **methods**. You may recognize methods by other names in different languages: functions, procedures, or subroutines, for instance.

### *Methods To The Rescue*

If your program is a united league of super heroes, the methods are the individual super heroes, each with a specific superpower. Don't forget that YOU are the programmer, which gives you the awesome capability to, using code, endow the hero with superpowers you invent and then name them. When disaster strikes and there are problems to solve you call them into action by name, "`adderGirl()`, add the numbers `23423` and `282757`, quick! `diceRollerGuy()`, give me a random number up to `6`! `sortBoy()`, sort the list of targets by strategic value! `glitchGal()`, make that villain's computer crash!"

Can we stretch this analogy a bit further? To explore how we might declare our own methods, let's head to the lab and mix some super secret serums to imbue `glitchGal` with power! Note: since methods are really about one primary action, programmers almost always name them using a verb. Let's go with `glitchGal`'s nickname which can also work as a verb, glitch.

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

    glitch(); // Call glitch into action!
    System.out.println("If glitch doesn't break program, we get back here");

  } // end of main method

  public static void glitch() {
    // barf out scary-looking random characters within the ascii subset
    // of unicode
    int count = 1;

    while (count <= 1000) {
      System.out.print( " " + (char)((int)(Math.random()*128)) );
      //chars 0-127
      count++;
    }

    count = count / 0; // this looks dangerous
  } // end of glitch method
}
```

You should try running this code, but even if you don't, take a close look at it, and try to figure out what it does — the comments will help.

Did you notice that `glitch()` has a **header** that looks very similar to `main()`? The header is also known as a **method signature**.

```
public static void glitch() {
  //...
}
```

The code inside the braces is the body of the glitch method and defines what should happen when `glitch()` is called. The header plus the body is called a **method declaration**.

Notice that `glitch()` and `main()` are equals when it comes to indenting, too. This is because they are both methods, but `main` is a bit special. The main method is always the first hero to be called into action when a problem needs solving; it is the leader of the team. In other words, when a Java program is executed, it looks for a `main` method to start executing. Even if it was listed after the `glitch()` declaration, the program would start with `main()`.

One of `main()`'s commands is to call `glitch()` into action. In the main() method, this statement does not *declare* glitch():

```
glitch();
```

Instead, it causes a jump to the code inside the `glitch()` declaration below the main method declaration — `main()` asks `glitch()` to do her part in solving the problem, and she takes over until she is done with her task. After `glitch()` is done, `main()` would normally take command again. In this case, `glitch()` may break everything with a divide by zero!

Java classes can contain many methods to jump between. You never know when `diceRollerGuy()` and `adderGirl()` will be needed! The code below shows how this might be organized, with the bodies of the methods unfinished — methods with incomplete bodies are called **method stubs**, and sometimes programmers use them while outlining larger programs.

```
public class GlitchLabExperiment {

  public static void main(String[] args) {
    // code that calls glitch(), diceRoller(), and adder() to fight super
    // villains!

  } //end of main

  public static void glitch() {
    // code to glitch: maybe calls to add() and rollDice() could go here…
  }

  public static int rollDice() {
    int result;
    // code to set result randomly...
    return result;
  }

  public static double add(double a, double b) {
    int sum;
    // code to add a and b and store in sum
    return sum;
  }
}
```

Even with only these stubs, try to imagine how the flow of the program would jump from one method to the next. From `main()` there could be a call to `rollDice()` to help make a decision, then when control comes back to `main()` maybe a call to `add()`. When `add()` is finished and, once again, control goes back to `main()`, `glitch()` could be called. Notice that any method can call other methods. So `glitch()` could also make a couple of calls to `rollDice()` or `add()` when it needs to. Eventually, control would usually come back to `main()` and, when the end of `main()` is reached, the program would end.

## *Check Yourself*

1. What makes the `main` method special?

2. What term is used to describe methods declarations with enough detail to be called but have bodies that are incomplete?

3. Because they are used for actions, it is good practice to name methods using a verb, true or false?

## §6.2 Method Benefits (Not Just Blocks of Code)

Containing code in sections might make you think of blocks of code, like the block of code that is part of an `if` statement between the opening and closing braces.

```
if (method > block) {
    methodScore = methodScore + 1;
    System.out.println("Score one for methods");
}
```

This is a valid observation. Blocks of code surrounded by braces in Java have some of the same characteristics as methods. For instance, they have their own **scope** for any variables declared inside them making that variable unreachable from the outside.

```
if (method > block) {
  int test = 5; // test is declared here and can't escape
}

System.out.println(test); // test is an unrecognized identifier here!
```

However, there are some key reasons to write a method instead of just blocks of code:

1. It has an identifier: **A method is *named*,** meaning you can call it into action by name whenever you need it.

2. You can argue with it: **A method can take arguments**, data that is passed to it so that it can be flexible.

3. It gives back: **A method can return data** to whichever method called it into action, once it has completed its task.

4. **It's reusable**: A method that can be called by name, be given different sets of data to work on, and return a result can be useful to call on many times in a program. A good example is a program with a menu - the method that displays the  menu can be called repeatedly. Sometimes methods are made to be used in many programs, too.

5. It has secrets: **A method is encapsulated**, meaning it can be called into action even if you can't see the code that makes it work. Think of `println()`, for instance! Even if you

write the method yourself, you may never look at it again — one less piece of a problem to hold in your mind.

6. **Some blocks of code want to be methods when they grow up**. Yes, this is metaphorical, but you should always imagine a block of code this way, and occasionally, it will make sense for you to fulfill its dream — take a block of code and make a new method out of it.

## *Check Yourself*

1. What problems can you guess might happen if you copy and paste the same code in several places within a program, especially over time?

2. What can you do to avoid the problems in question 1?

3. The following code produces an unrecognized identifier error on secret in `main()`. Why?

```
public class Scope {
  public static void whisper() {
    int secret = 481516;
  }

  static void main(String[] args) {
    whisper();
    System.out.println(secret);
  }
}
```

## §6.3 Predefined Methods: Learn from Math

The Java language specification includes a set of standard libraries for programmers to take advantage of. So, let's take advantage.

`Math` is a class in a standard library package called `java.lang`. We'd also find other common classes like `String` and `System` (remember `System.out.println()` ?) in the `java.lang` package. Because it has such commonly used classes, the `java.lang` package is automatically imported into every Java program — unlike the `java.util` package classes, such as `Scanner`, which need to be explicitly imported.

You can check out the `Math` class documentation here:
https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

And you can consult the Relevant API in the appendix of this book for a look into the most common methods we will use from the `Math` class.

`Math` has many methods we can use to solve problems, but our biggest problem right now is learning how to use and make our own methods, so we will also study how the `Math` methods work.

### *Using Returned Values*

Let's start with `random()`.

```
double test = Math.random();
// assigns 'test' a random number >= 0.0 and < 1.0
```

It's part of the imported `Math` class, so to reference it, we connect it to `Math` with a **dot operator** — sometimes called a **member operator** and, technically, a **separator**. You can read it as "Math dot random", but it means, "Call the random method that's a member of the `Math` class."

`random()` is a handy method for adding some unpredictable elements to a program, and it's notable because it gives back. In other words, when you call this method it returns a value, that random number we want. Note that a call to random() will only return a double that is greater than or equal to `0` and less than `1` - so you'll end up with numbers like `0.001`, `0.654`, `0.99999`, and so on.

Most of the `Math` methods return values, which makes them very helpful. Think of each method in `Math` as a mini-program that does one very specific task. Because of this, every method also requires YOU, the programmer, to be very specific with how you handle the information it returns to you. Consider this example:

```
int test = Math.random();  //syntax error!
```

The type of data returned by `random()` is `double` and you can't assign it to an integer variable — Java won't force a `double`'s round peg into an `int`'s square hole because it would lose data.

You can get a hint about how `random()` is defined in the documentation: https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html#random—

| | |
|---|---|
| `static double` | `random()`<br>Returns a positive double greater than or equal to 0.0 and less than 1.0 |

See the data type to the left of `random()`? That's the **return type**. You can make methods that return a value of any data type, or return nothing. Use the keyword **void** as your return type, like `main()` does, as a placeholder to show that a method doesn't return a value.

### Getting More Out Of `Math.random()`

Let's talk about using `random()` to get integer numbers in any range. Since we can't get Java to assign `random()`'s returned `double` value to an `int` automatically, let's force it using typecasting, instead.

```
int test = (int)Math.random();  //No syntax error, but always zero!
```

This removes the syntax error, but doesn't give us anything usable. It will always truncate the number, removing everything after the decimal point. This is why it's called **type narrowing** — we lose information. All results from `random()`, like `0.5`, `0.353`, or `0.98`, for example, will become an integer value of zero.

To get random integer results, you can treat the resulting `random()` `double` as a percentage. Remember, 0.50 is 50%. So, if we multiplied 80 by 0.5 we'd get 40.0. If we multiplied 80 by 0.0, we'd get zero. If we multiplied 80 by 0.01 we'd get 8.0. If we multiplied 80 by 0.99, we'd get 79.2 — which doesn't quite get us to 80. If we convert all of those results to `int`s, we can get a random result between 0 and 79 inclusive (never an 80).

Here's the code to do that:

```
int test = (int)(Math.random() * 80);
//integers from 0 to 79 inclusive
```

Note the very important parenthesis around `Math.random() * 80`. If you were to remove them, you would first typecast the `random()` result into an `int`, making it a zero. Of course, that would then multiply the 80 by zero and you'd be back to getting zero every time.

How about 1 to 80, instead, and 0 to 80?

```
int test = (int)(Math.random() * 80) + 1;
// integers from 1 to 80 inclusive

int test = (int)(Math.random() * 81);
// integers from 0 to 81 inclusive
```

Incidentally, you may want to research the `Random` class (that's also in the Relevant API) that is part of the `java.util` package for some more convenient random number generating methods.

## *More Ways to Use Methods*

`Math.random()`'s `double` return type means this method can be called anywhere we can place a `double` value. Think that through - anywhere you might put a variable with a `double` value in it, or just type a `double` literal into code, you could also put a method that returns a `double` value. That's because when it completes its work, it gives back a `double` to use in the place it was called from.

So, like before, we can call it when assigning a value to a variable.

```
double randomPercentage = Math.random();
```

We can *also* call a method as an **argument** to another method call — more on arguments later.

```
System.out.println(123.456);
// a double value passed to println()

System.out.println(Math.random());
// returned double value passed to println()
```

We can *also* use a call to a method as part of an expression like this:

```
double result = 2.5 + 670 + Math.random();
```

To help you understand how to use methods that return values, let's examine how the code above plays out when executed.

It really doesn't complete work from left to right, does it? Before the variable named `result` can get a new value assigned to it, we have to solve the expression to the right of that assignment operator (`=`). The expression is going to involve some adding because of the addition operator, but the first real action is to find out what `random()` is going to give us!

So, it jumps over to the `random()` method, does the work, and gives us back, we'll say, `0.33`. Now, the expression is simplified to `2.5 + 670 + 0.33`, which is solved to become `672.83`. At last, the task of the assignment operator can be completed by storing `672.83` in the variable named `result`. Mission accomplished.

Incidentally, many programmers use the expression **rhs** to refer to the code on the right hand side of an equals sign and **lhs** to refer to the code on the left hand side.

**The Power of Methods Returning Values**

Imagine that code with whatever crazy math is used to generate the random numbers in place of the simple call to `Math.random()` (like the linear congruential method - it's a thing). It might end up taking multiple lines and would certainly look more complex and take longer for a reader to decipher.

So, even though you can make and use methods with a `void` return type, you should find that most methods complete a task and *return some result*. That way they can be used to complete some larger task. This is a great way to break complex problems into sets of smaller problems that are each more simple and easier to solve. Standard libraries of code like the `Math` class, that provide commonly needed methods like `random()` do this for you, but you will want to make your own methods, too.

Many companies have their own libraries that they design and use for their software. Individual programmers will use the company-wide libraries, and oftentimes write their own libraries for other programmers in the company to use.

## *Passing Arguments to Parameters*

For our next `Math` method, let's look at `floor()`. It returns the largest number that is less than or equal to the number given. It's a bit like truncating a double to an `int`, but the result is still a `double`. The method header for `floor()` in the documentation looks like this:

```
public static double floor(double a)
```

Look at that conspicuous “`double a`” between the parenthesis. That area holds a **parameter list**, where several variables, called **parameters**, can be assigned values from the outside.

How do we pass values to the parameters in a method? When you call the method, you simply list any values in the same order as the parameters, and the values will get assigned to the corresponding parameters. You’ve been doing this for awhile now with `println()`, for instance, when you pass it `String` literals.

```
Math.floor(55.95); //pass the argument 55.95 to a parameter in floor
```

When you call a method, whatever elements you type between the parentheses are called arguments. In the example above, `55.95` is the argument. So, the argument, `55.95`, is passed to `floor()` and is assigned to the parameter, `double a`, written into the `floor()` header where it's declared.

**Method Call and Method Declaration**

Be careful to make a distinction between a method *call,* and a method declaration*.* The *call* passes the arguments, the *declaration* has parameters to catch the argument values, and then, the body of the method *declaration* runs. In short, a *call* passes arguments to the parameters in the method *declaration* so it can run.

Add to that, the ability for some methods to return a value back to where it was called and you have the process of calling a method in a nutshell.

You won’t see any results from that last line of code, by the way. The returned value from `floor()` goes nowhere! It’s a very common mistake to invoke a method but not store the result anywhere. Let’s fix that:

```
double flooredNumber = Math.floor(55.95);
//pass 55.95, get 55.0 returned

System.out.println(flooredNumber);
```

So, now we see the method get some results. A value of `55` is returned and then our statement sets `flooredNumber` to `55.0`. It might be obvious what `floor()` is even good for, though.

**Round A Number**

Note that `floor()` doesn’t round by itself, but we can add a touch of math to get a rounded result. Since a low number like 55.001 all the way up to a high number like

55.9999 will give floor a 55, how can we get it to round?

When rounding to the nearest one, a value of 5 in the tenths digit or above means round up, right? What if we just added 0.5 to the number? Then 55.001 would be 55.501 and `floor()` would still return 55. That same 0.5 added to 55.999, though, would give us 56.499. Send that to `floor()` and you'll get 56. That's what we want!

Okay, so let's round a number using `floor()`.

```
double roundedNumber = Math.floor(55.95 + 0.5); // 56!
System.out.println("Rounded number: " + roundedNumber);
```

That seems to work! Notice that in place of a single value, we can use an expression as an argument to a method. The statement doesn't send the whole expression `55.95 + 0.5` to `floor()`. Instead, it solves the expression first. So, in this case, it sends `56.45` as the argument.

You have been doing this for awhile with `println()`, by the way. Just look at the output statement above. The `String` literal, `"Rounded number: "` is added to the variable, `roundedNumber`, before it is sent as a new `String` to `println()`.

Can we pass variables to methods when we call them? Not exactly. Technically, a call with a variable as an argument will pass the value that is in the variable to the method. Let's expand our `floor()` example a bit.

```
double original = 123.2;

original = original + 0.5;
// actually not good practice - original is misnamed now!

double roundedNumber = Math.floor(original);
// 123.0

System.out.println("Rounded number: " + roundedNumber);
```

Focusing on `Math.floor(original)`, the variable named `original` is the argument, but the variable itself doesn't get passed to `floor()`. Instead, the value it stores, `128.2`, is passed. Some programming languages have multiple passing types, like passing by reference, but Java always passes by value.

### What about `Math.round()`?

You might have noticed `round()` in the list of `Math` methods and wonder why we didn't just use that instead of the example using `floor()`.

If you look closely at the headers for the different versions of `round()` you'll see they use the data types `float` and `long`. These work fine if you are familiar with how to write `float` literals — tack an `f` on the end — or want to pass a `double`, and typecast the resulting `long` value into an `int`. This seemed a distraction from the main points, but it's a fine method. Try it out!

### Multiple Parameters

Below you'll find a list of some useful `Math` methods to look into, but first let's look at an example of calling a method with multiple parameters. Here's the header for `min()`.

```
public static double min(double a, double b)
```

We can see that it has a return type of `double`, so we'll get a value back if we call it. We can also see that it has two parameters that are both `double`s. Besides that, the documentation describes that it returns the lesser value of the two that were passed to it. That's all we need to know to use it! Let's setup a program to let a user control the volume knob on a certain, infamous guitar amplifier.

```
Scanner scanner = new Scanner(System.in);

double volumeTarget = scanner.nextDouble();

double actualVolume = Math.min(11, volumeTarget);
//limits actualVolume to 11!

System.out.println("Volume set to " + actualVolume);
```

The value, `11`, is sent to `min()`'s `a` parameter, and the value in `volumeTarget`, whatever the user entered, is sent to `min()`'s `b` parameter. It sends back whichever is lower so that any user trying to set the volume above `11` won't be able to do so. However, any value below `11` that is entered by the user will be assigned to `actualVolume` just fine. You could solve this with an `if` statement but this is a nice, short and simple solution.

The order of the arguments doesn't matter when calling `min`, but many methods have very specific purposes for each parameter. For instance, `pow()`.

```
public static double pow(double a, double b)
```

Importantly, the specification states that `pow()` “Returns the value of the first argument raised to the power of the second argument.” In the example below, notice how the order of the arguments truly does matter when using `pow()`.

```
System.out.println(pow(10, 2)); //outputs 100
System.out.println(pow(2, 10)); //outputs 1024
```

It would be a nicer method if the parameters had more meaningful names. Maybe ‘base’ and ‘exponent’, for instance. Wouldn’t that make it easier to understand? Keep that in mind when you make your own methods!

Before we do that, here’s a list of some `Math` methods you will want to study (but be sure to check out the Relevant API in the appendix, as well):

- `abs()` for finding an absolute value.
- `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()` for dealing with angles - in radians, be careful!
- `toDegrees()` and `toRadians()` for converting between degrees and radians (see above).
- `ceil()`, `floor()`, and `round()` for rounding numbers.
- `log()` for finding logarithms.
- `min()` and `max()` for choosing compared values.
- `random()` for getting random numbers.
- `power()` for calculating exponents.

You can find details at Oracle’s Math documentation web page:
https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

**Are Classes the Real Superheroes?**

While the methods-are-heroes analogy may help you imagine calling heroes into the fray to use their skills, a method should really be focused on one task. This is why the hero method examples had such a narrow focus - `diceRollerGuy()` is pretty limited, after all.

Most great superheroes in fiction are more nuanced and able to handle many tasks. The `Math` class has many methods that all revolve around arithmetic. So, you could think of `Math` as “TheMathmetician” and all the various methods as “TheMathematician's” superpowers.

The idea of a class being a type of object and its methods revolving around that object's data will be much more important when you define more complex classes. For now, you can use classes as simple collections of related methods.

## *Check Yourself*

1. What must be present in the header for `getStuff()` for the following statement to execute?

```
int numberOfThings = getStuff();
```

2. In the code below, what term is used to describe the two items in parentheses?

```
doIncredibleTricks(55, userNumber);
```

3. In the code below, what term is used to describe the two items in parentheses?

```
public static void doIncredibleTricks(int count, double num) {
  //logic omitted...
}
```

## §6.4 Defining Your Own Methods

When it comes to working with methods, there is only one more rewarding experience than finding a method you can use that is already written: writing your own awesome methods.

You *will* need to declare your own methods. Why? Well, if you are writing something you are passionate about, chances are it includes something unique! If someone is *paying* you to write a commercial program, there better be something to make it stand out so customers will choose it over competing software.

There is plenty of available code out there to support all kinds of projects, and you will want to take advantage of it when appropriate, but the secret goodness that YOU add to make it great and make it yours will need to live in methods. Let's get to it — but hey, if you haven't read the above sections, you really need to!

### *When to Write a Method*

The first step is to decide when you want to add a method. Often, you will be writing code and think of a task to perform and decide that it belongs in a method. Why? Maybe you think you'll reuse it. Maybe you just think it will help keep the code more readable and clear.

The purpose of our example program below is to get a movie's star rating from the user, who is probably a reviewer, and then display one of three possible summaries of what that star rating means, so they can copy and paste it into their review article. Here's the code, so far:

```
import java.util.Scanner;

public class StarRating {

  public static void main(String[] args) {

    Scanner scanner = new Scanner(System.in);
    int starRating = scanner.nextInt();

    // output a summary of what the rating means

  }
}
```

So, you are about to write some `if` statements and it dawns on you: there are several parts of a larger program that may need these rating summaries. Maybe a web page that shows a chart of what the star ratings mean, and then, in some database views that aren't in the original article, it will also be shown. You decide to move the code that outputs the summary to its own method because it will be reusable.

This method will do *one* thing, print a message to standard output based on a star rating that is sent to it. Sometimes it helps to start by writing the method call first to imagine how you want to use it. Note that as we add on to this code, we'll be putting anything we add in bold so that it sticks out (just like in Lego instructions!).

```
import java.util.Scanner;

public class StarRating {

  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    int starRating = scanner.nextInt();

    // output a summary of what the rating means
    // SummarizeRating(starRating);

  }
}
```

### Writing a Method Header

Now, we can analyze the call to the method and decide how the header will be written.

The header of a method declaration — sometimes called a method definition — needs modifiers. So far, we only know about `public` and `static`, so we'll start with those:

```
public static
//modifiers
```

Next, we need to add our return type. How do we know what return type to include? Look at how you are calling the method. Is the call part of a calculation? Is the call being assigned to a variable? Is the call being used as an argument in another method call? Answering yes to any of those questions would mean a value needs to be returned by the method. In this case, we can answer no to all of the questions, so we have nothing to return. To show that there is no return value and therefore no data type to return, we type `void` - just like `main()` uses.

```
public static void
//            return type
```

Next comes the identifier, the name. We already thought of that when we typed the call.

```
public static void SummarizeRating
//                 indentifier
```

Now, we need parentheses and the parameter list that goes inside. To know what we need, look for any arguments sent in the method call. If there are no arguments we can leave the parentheses empty.

However, we placed `starRating` in the parenthesis as an argument, so we *will* need a parameter to catch its value. We can name it whatever we want - something meaningful is best - but the data type *needs* to match the data type that is sent in the argument. Since `starRating` is an `int`, we will use that data type for the parameter.

```
public static void SummarizeRating(int rating)
//                                parameter list
```

The method header is complete, but we should add the opening and closing braces to set up the beginning and end of the method body. If there was a return type that wasn't `void`, we'd also need to add a `return` statement. More on that later.

Now, we have a complete method declaration for a method that does... nothing. After all, the body is empty. We made a method **stub** for `summarizeRating()`. We should uncomment the method call and check for errors. This way we know if we have our method header and our call matching, even before we work on the body of the method.

```
import java.util.Scanner;

public class StarRating {

  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    int starRating = scanner.nextInt();
    //output a summary of what the rating means
    summarizeRating(starRating);
  }

  public static void summarizeRating(int rating) {
    //needs filling
  }
}
```

Hopefully when we execute this program, it will compile and run. And do nothing when `summarizeRating()` is called. If everything works, it's time to write the body of the method!

## *Writing a Method Body*

As mentioned earlier, one benefit of adding methods to complex programs is that it breaks a larger problem into smaller problems. One way you may feel that advantage is when you start writing the body of a method. You really should - almost - forget the rest of the program. Everything you need is in the header of the method. If it doesn't tell you everything you need, you may have formed it incorrectly. Let's look at method declaration, alone.

```
public static void summarizeRating(int rating) {

}
```

You may think you need to know more about `starRating()`, but why? We have our parameter, `rating`, right there in the header. A parameter is like a variable declaration that has been secretly assigned a value when the method begins. It *always* gets a value. This is true, because nobody can call this method into action without sending the right kind of data as an argument. We have the luxury of letting variables from other methods slip our minds.

We have only this one variable, `rating`, to worry about. We also know from the return type being `void`, that there is nothing to return. We need to know the purpose of the method, of course, as most method names don't give the whole story. This method's identifier is meaningful enough that it does help, at least.

Let's add the code to summarize the rating to our method body. Again, just focus on this method alone.

```
public static void summarizeRating(int rating) {
  if (rating == 1) {
    System.out.println("Watch some cat videos, instead.");
  } else if (rating == 2) {
    System.out.println("Middle of the road.");
  } else if (rating == 3) {
    System.out.println("See this now!");
  } else {
    System.out.println("Invalid");
  }
}
```

Great! Now, we can run this program. So maybe you are thinking, "what about input?", or, "how can this work when `rating` isn't assigned a value?". This is why I said you should *almost* forget the rest of the program. This method does the one action we intended and only that action. That is how it should be. The `main()` method takes care of input and calling this method. It *will* run.

## *Writing Return Statements*

The methods in the `Math` class all return values. Let's write our own version of one we looked at, earlier: `min()`.

Here's the code we'll use to test this new version of `min().` Notice that in this code, `min()` is used in an assignment statement and it is also used as the argument in a call to `println().` If `min()` didn't return a value after it was called, there would be no way for these statements to execute!

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

    int lowNum = min(5, 2);
    System.out.println(lowNum);
    System.out.println(min(2, 5));

  }
}
```

For the header, let's just copy how `min()` was written in the `Math` documentation

```
public static double min(double first, double second)
```

Remember that the purpose of `min()` is to return the lower of two values sent to it. This is why there are two parameters. I decided on `first` and `second`, instead of `a` and `b`, because it has a bit more meaning, maybe, but mostly to stamp this method as our version. Incidentally, there wouldn't be a problem for the computer to distinguish between ours and the one that belongs to `Math` because we aren't calling it with using the `Math` class and the dot operator, and we are declaring it inside the `CustomMin` class, not `Math`.

Here's one way to declare `min()`:

```
public static int min(int first, int second) {

  int lowerNumber;

  if (first < second) {
    lowerNumber = first;
  } else {
    lowerNumber = second;
  }

  return lowerNumber;

}
```

Using a `return` statement is something new we need to pay attention to. Without a `return` statement that can be reached, the Java compiler will report it as a syntax error. It knows you *intend* to return a value by your non-`void` return type, so leaving the possibility of no return statement is impossible to resolve.

Also, whatever value is returned needs to match the data type listed as the return type. Notice we are returning the value in `lowerNumber` which was declared as an `int`, matching the return type exactly.

In the example above, we follow some logic to copy the smaller value into `lowerNumber`, and then wait until the end to return the value in `lowerNumber`. Return statements don't have to come at the end of a method, though. As soon as a `return` statement is reached, no matter where in the code it is placed, it will cause the method to stop running and return the value to where the method was called from.

We can use this knowledge to rewrite a shorter solution for our method body, and even eliminate the need for `lowerNumber`, altogether.

```
public static int min( int first, int second ) {

  if (first < second) {
    return first;
  }

  return second;

}
```

Short and sweet, right? Notice how we could have used an `else` to enclose that last return, and might have been inclined to do so from writing `if` statements in the past. However, there is no reason to. If `first` is less than `second` and the `if` statement's first block of code is executed, the `return` statement immediately ends, sending the value of `first` back to where the method was called from. That means the second `return` won't be reached! If `first` isn't less than `second`, the block is skipped, and the next statement returns `second`, instead.

If you've learned about the conditional operator ( `? :` ), you could write this method body in one return statement!

One more example. Here is a complete method declaration for a method called `add`.

```
public static int add(int a, int b) {
  return a + b;
}
```

Yes, this is a silly method. You would be better off just using the addition operator to add integers together, *but* it does show how you can use an expression in a `return` statement. As you may expect, the expression will be calculated and the resulting value will be returned to where `add()` was called.

### *Go Big*

With the ability to write your own methods, you can tackle much larger and complex programs. You can naturally break the program into smaller methods with a single purpose, focus on the algorithm for just one method at a time, and look for opportunities to reuse the methods you've written.

## *Check Yourself*

1. Write code to call the method with the header shown below.

```
public static void eatVeggies(int veggieCount)
```

2. Write code to output what a call to the method gives back when declared with the header shown below.

```
public static double getPortions (double total)
```

3. Write a method header (only) that would work for the method call below.

```
rotateAndDivide(123.2, num * 1.2);
```

4. Write a method header (only) that would work for the method call below.

```
int beakerMeasurement = getLiquid(12, 54.2);
```

# LOOPS

"You should only use 'while – true' loops when you don't know when the loop is going to end …

… but you should always know when the loop is going to end."

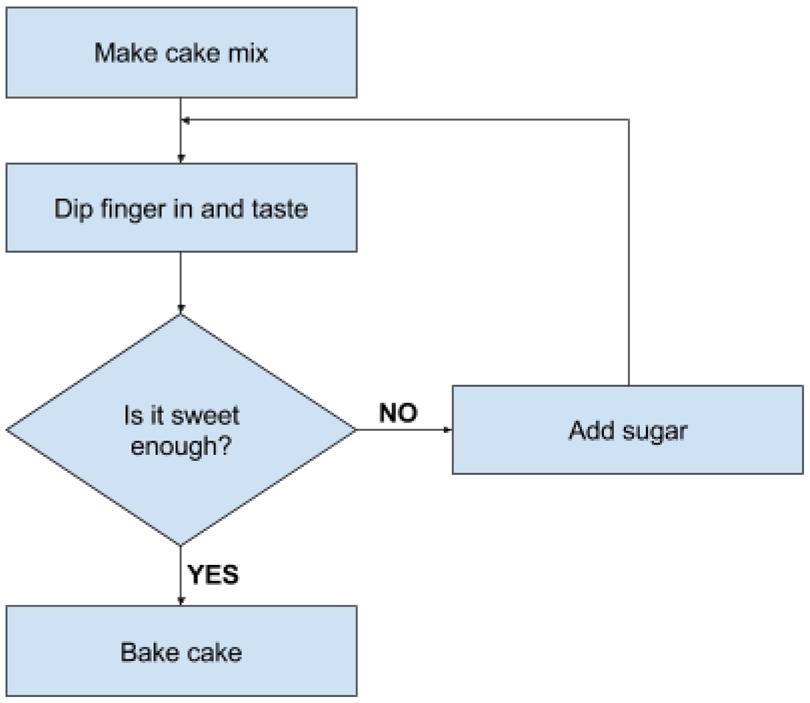
**- Dr. Kotok -**

## §7.1 Control Structures - Loops

You know the `if` statement, a control structure that can have the program choose to execute a block of code, or skip it. There are several other control structures that will manipulate the flow of code and repeat sections of code, over and over again. The repetition of a block of code is called an **iteration**.

This chapter will start by studying the `while` statement heavily and then introduce the other two main types of looping statements. But for now, think of a loop (in terms of program flow) as a few lines of code that get executed until a condition is met.

In a non-computer science realm, baking a cake is a good example of implementing loops. Think of mixing the batter. As you mix, you might stick your finger in the batter and see if it tastes good. If it isn't sweet enough, then you would add sugar. And then dip your finger in and taste it again. If it isn't sweet enough, more sugar. Then taste it. Then add more sugar. Then taste it. When you taste it, and it is sweet enough, then move onto the next phase.



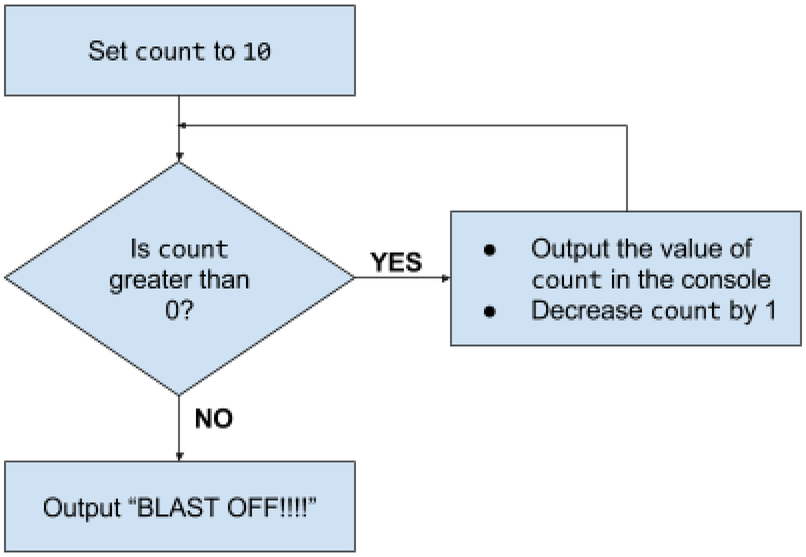
*A flowchart for baking a cake*

## §7.2 The `while` statement

The best iteration statement to start with is the `while` statement because structurally it is the same as the `if` statement:

```
while ( <boolean expression> ) {
  //BODY of the loop
  //<statements to repeat while true>
}
```

Just like the `if` statement, the code within the body will execute if the conditional, the boolean expression, evaluates to be `true`. However, unlike the `if` statement, after the body of a `while` statement runs, it goes back to the condition and test it again. Depending on the result, the program may repeat the code or move on to the lines of code following the `while` loop. The following example will display a countdown from 10 to 1, and print some text:



*A flowchart for counting down from 10 to 1*

```
//example: display a countdown
int count = 10;

while (count > 0) {
  System.out.println(count + "!");
  count--;  //same as: count = count - 1;
}

System.out.println("BLAST OFF!!!!");
```

OUTPUT:

```
10!
9!
8!
7!
6!
5!
4!
3!
2!
1!
BLAST OFF!!!!
```

To better understand the order of execution of the program, let's break it down a little and walk through the program line by line (code is numbered to help with this):

```
1   //example: display a count-down
2   int count = 10;
3   while (count > 0) {
4     System.out.println( count + "!");
5     count--;  //same as: count = count - 1;
6   }
7
8   System.out.println("BLAST OFF!!!!");
```

Here's what's happening:

- The `while` statement starts on line 3, and will test the condition (`count > 0`)
- The `count` is initialized to `10`, so this expression evaluates to `true`. Because the boolean expression is `true`, the body of the loop will be executed, starting with line 4
- The `count` is printed, and then the code in line 5 decreases the value of `count` by one (so now it is `9`)
- Line 6 ends the body of the `while` loop, so the control jumps back to line 3 and tests the conditional statement again
- The value of `count` is now `9`, so the expression evaluates to `true` again and will run all the code in the body of the loop

- The flow of execution runs in this pattern until the value of `count` reaches `0` on line 5. After the value of `count` decreases to 0, the test on line 3 will evaluate to `false`, and the body of the `while` loop is skipped, so control goes to line 7.

- Finally, the last line gets executed and prints “BLAST OFF!!!!”

In summary, a `while` loop operates like this:

1. Evaluate the boolean expression (conditional) to be `true` or `false`
2. If `true`, execute the body, and go back to step 1
3. If `false`, skip the body and go to the next statement after the loop

## §7.3 The LCV

The variable that controls the loop is appropriately called the **Loop Control Variable (LCV)**. When you have an error working with iterations, it is likely an issue with the LCV. There are three things that need to happen for the loop to run correctly:

1. Initialize the LCV before the loop starts
2. The conditional uses the LCV (this is called the terminal expression)
3. Update/change the LCV in the body of the loop

The following examples will run forever because there is an issue with how the LCV is used, resulting in an **infinite loop**. An infinite loop occurs when the code in a loop is executed but the loop is poorly designed and there is no way for the conditional statement to evaluate to `false`, so the loop iterates forever.

### Examples of Infinite Loops

In this case, the terminal expression does not use the LCV so the loop will continue to countdown forever, displaying negative numbers:

```java
//example: display a countdown
int count = 10;
int value = 10;

while (value > 0) {
  System.out.println( count + "!");
  count--;  //same as: count = count - 1;
}

System.out.println("BLAST OFF!!!!");
```

OUTPUT:

```
10!
9!
8!
7!
6!
5!
4!
3!
2!
1!
0!
-1!
-2!
...
```

In this example, the body does not update or change the LCV so the program will print 10 over and over and over and over…

```
//example: display a countdown
int count = 10;

while (count > 0) {
  System.out.println( count + "!");
}
System.out.println("BLAST OFF!!!!");
```

OUTPUT:

```
10!
10!
10!
10!
10!
10!
10!
10!
10!
10!
10!
10!
10!
...
```

Don’t panic! Although this sounds catastrophic, it isn’t that bad. Every programmer creates infinite loops every now and then. It’s easy to fix - just terminate the program from running. Different IDEs have different mechanisms for this, so make sure you know how to end a process.

## §7.4 Common `while` loops

There are common algorithms or tasks done with the `while` statement:

1. Running instructions some number of times
2. Accumulating
3. Counting
4. Search (a ***sentinel*** controlled loop)

### Running instructions some number of times

The first countdown block of code is an example of the "running a certain number of times" algorithm. Note that you don't need to know *precisely* how many times the instructions will be run, just that they will be run a finite number of times.

Unlike the countdown example, many times you will start with a LCV initialized to zero. In the conditional, you will check to see that the LCV is less than the number of times you want to run it. The last step in the body of the loop will increase the LCV by 1. The following example will run 5 times, and display random six-sided dice rolls.

```
//example: running x times
int count = 0;
int numOfTimesToRoll = 5;

while (count < numOfTimesToRoll) {
  int value = (int)(Math.random()*6) + 1
  System.out.println(value);
}
```

There are several variation of this style of loop. For example, you may want to display the values and have them numbered. To do this, you can start `count` with a value of `1`, and then use check to see that count is less than or equal to `numOfTimesToRoll` in the conditional:

```
//example: running x times
int count = 1;
int numOfTimesToRoll = 5;

while (count <= numOfTimesToRoll) {
  int value = (int) (Math.random()*6) + 1;
  System.out.println( count + ". " + value );
  count++; // same as count = count + 1;
}
```

## Accumulating

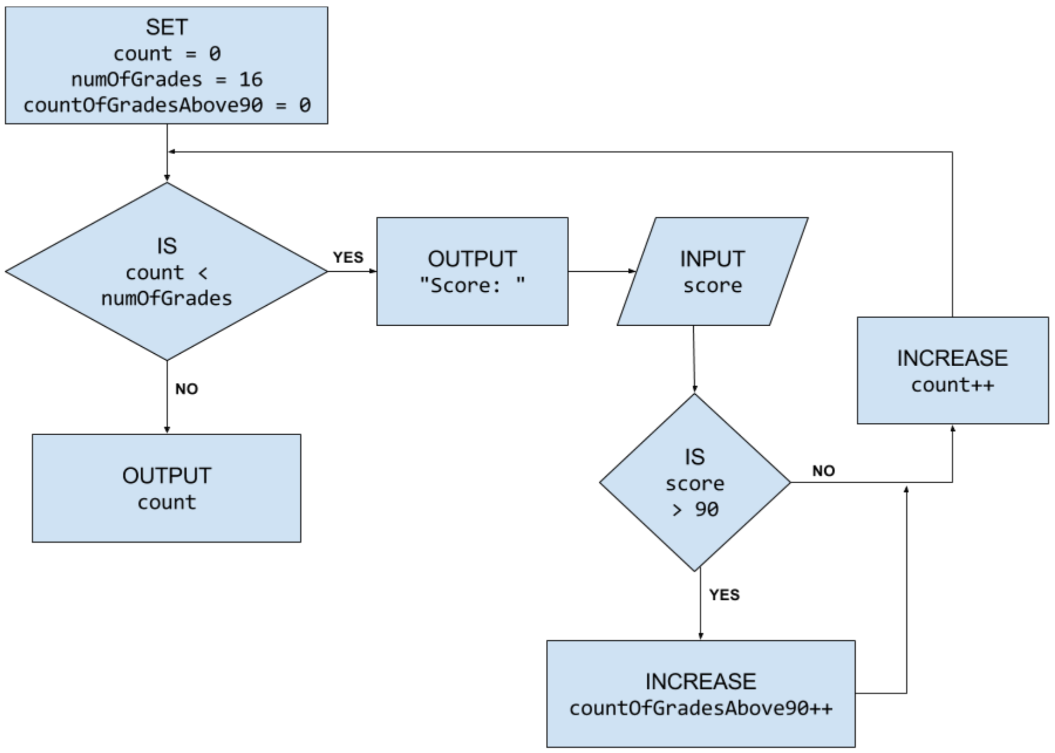
A popular variation of the loop running a count number of times, is accumulating, or adding to a running sum. To do this, you will create a sum variable before the loop starts and keep adding to it in the loop. The following example will get five values from the user and display the average. The key part of this is keeping track of the running sum (accumulating the scores).

```
//example: accumulating
int count = 0;
int numOfScores = 5;
int sum = 0; //key to accumulating

while (count < numOfScores) {
  System.out.print("Score: ");
  int score = input.nextInt();
  sum = sum + score;  //could have used: sum += score;
  count++; //same as count = count + 1;
}

double avg = (double) sum / numOfScores;
double roundedAvg = Math.round (avg * 10) / 10.0;
System.out.println("Average = " + roundedAvg);
```

## Counting

Another common use is a counting loop. For example, you would like to count the number of grades above a 90 you have. The key to this algorithm is the nested `if` statement used within the `while` loop. Naming of the variables is important to help keep track of what variable is holding what value. The algorithm is:

1. Loop through your 16 grades
2. If any grade is above a 90, count it.
3. Display the count at the end.



*A flowchart for the grade counting algorithm*

```
//example: counting
int count = 0;
int numOfGrades = 16;
int countOfGradesAbove90= 0; //key to counting

while (count < numOfGrades) {
  System.out.print("Score: ");
  int score = input.nextInt();
  if (score > 90){
    ++countOfGradesAbove90;
  }

  ++count;
}

System.out.println("You have " + countOfGradesAbove90 + " grades above 90!");
```

## §7.5 The for Loop

Recall from section 7.3, there are three things that need to happen for the loop to run correctly:

1. Initialize the LCV before the loop starts,
2. The conditional uses the LCV (this is called the terminal expression)
3. Update/change the LCV in the body of the loop.

There is a control structure that does just that in one declaration. The `for` statement.

```
for (<initialization statement>; <conditional>; <update statement>) {
  //Code to loop
}
```

In terms of execution the following order runs when the compiler enters the `for` statement:

1. Run the initialization statement
2. Evaluate the conditional to be `true` or `false`
    a. If `false`, skip the body of the `for` loop, (DONE - do not do steps 3,4)
    b. If `true`, enter the body of the `for` loop, and run it.
3. Run the update statement.
4. Repeat (goto step 2)

### Example 1: Output the integers between 1 and 10

```
// output the integers between 1 and 10, inclusive
for (int i = 0; i <= 10; i++) {
  System.out.print(i + " ");
}
```

OUTPUT:

```
1 2 3 4 5 6 7 8 9 10
```

Note that although the last number to be output is `10`, the actual value of `i` is `11`. This is oftentimes confusing for people, but it makes sense when you think about it. The variable `i` changes with each iteration, and the loop continues to run while `i` is less than or equal to `10`. So for the loop to end, the value of `i` must exceed `10`, and since it goes up by 1 each iteration, the last known value of `i` is `11`.

## *Example 2: Output multiples of 5*

```
// Print the first 6 multiples of 5
for (int i = 0; i < 6; i++) {
  System.out.print(Math.pow(5, i) + " ");
}
```

OUTPUT:

```
0 5 25 125 625 3125
```

Note that in this case, the conditional specifies that `i < 6`, not `i <= 6`. That's because we started at `0` and wanted six iterations. If we had used `i <= 6`, we would have output seven numbers.

## Example 3: Print a table for squares and cubes

```
//print a table of perfect squares and cubes
System.out.println("value\tsquared\tcubed");
System.out.println("------------------------");

for(int i = 1; i<=10; i++){
  System.out.println(i +"\t\t" + (i*i) + "\t\t" + (i*i*i) );
}
```

OUTPUT:

```
value  squared       cubed
------------------------
1      1             1
2      4             8
3      9             27
4      16            64
5      25            125
6      36            216
7      49            343
8      64            512
9      81            729
10     100           1000
```

## §7.6 The do-while loop

The `while` and `for` statements are pretest loops; that is, they test the condition first and at the beginning of each pass through the loop.

Java also provides a posttest loop: the `do-while` statement. This type of loop is useful when you need to run the body of the loop at least once.

For example, we can use a `do-while` loop to keep reading input until it's valid:

```
Scanner in = new Scanner(System.in);
boolean okay;

do {
    System.out.print("Enter a number: ");

    if (in.hasNextDouble()) {
        okay = true;
    } else {
        okay = false;
        String word = in.next(); //read in bogus input
        System.err.println(word + " is not a number");
    }
} while (!okay);

double x = in.nextDouble();
```

Although this code looks complicated, it is essentially only three steps:

1. Display a prompt
2. Check the input; if invalid, display an error and start over
3. Read the input

The code uses a flag variable, `okay`, to indicate whether we need to repeat the loop body. If `hasNextDouble()` returns `false`, we consume the invalid input by calling `next()`. We then display an error message via `System.err`. The loop terminates when `hasNextDouble()` returns `true`.

It's worth mentioning that `do-while` loops are rarely used, but they are a legitimate construct and you should acquaint yourself with them. There's also a loop that we did not review here - the `for-each` loop. It is contingent on data structures like arrays, so we'll examine the `for-each` loop when we look at arrays.

## §7.7 break and continue

Sometimes neither a pretest nor a posttest loop will provide exactly what you need. In the previous example, the "test" needed to happen in the middle of the loop. As a result, we used a flag variable and a nested `if-else` statement.

A simpler way to solve this problem is to use a `break` statement (remember these from our discussion on `switch`?). When a program reaches a `break` statement, it immediately exits the current loop and cedes control back to the main program flow.

In the following example, we build upon our model where we want to get a valid `double` from the user. If they enter a `double`, the loop ends. If the input is not a `double`, then an error message comes up and asks the user to enter a number (again):

```
Scanner in = new Scanner(System.in);
while (true) {

  System.out.print("Enter a number: ");
  if (in.hasNextDouble()) {
    break;
  }

  String word = in.next();
  System.err.println(word + " is not a number");

}

double x = in.nextDouble();
```

Since the conditional in the `while` loop will always evaluate to `true` (because it *literally* is `true`), the only hope this program has of not continuing forever is if the `break` command is encountered. Happily, the program is structured in such a way that a `break` will be executed whenever a valid `double` is entered.

Using `true` as a conditional in a `while` loop is an idiom that means "loop forever", or in this case "loop until you get to a `break` statement." Most programmers try to avoid `while-true` loops because things can easily go awry when intentionally creating infinite loops, though some instances such as the above situation are rare exceptions.

In addition to the `break` statement, which exits the loop, Java provides a **`continue`** statement that moves on to the next iteration. For example, the following code reads integers from the keyboard and computes a running total. The `continue` statement causes the program to skip over any negative values.

```
Scanner in = new Scanner(System.in);
int x = -1;
int sum = 0;

while (x != 0) {
  x = in.nextInt();
  if (x <= 0) {
        continue;
    }
    System.out.println("Adding " + x);
    sum += x;
}
```

Although `break` and `continue` statements give you more control of the loop execution, they can make code difficult to understand and debug. Use them sparingly.

# ARRAYS

*Q: Why did the programmer quit their job?*

*A: Because they didn't get arrays!*

**- Programmer Folklore -**

## §8.1 What is an Array?

Congratulations! You've made it to your first data storage type! The world of data storage is a rabbit hole of theory, efficiency, and practicality. Arrays are the first stop on the tour of storage methods.

You may be thinking to yourself, "Self, why would I need more ways to store data? I've been using variables for quite some time now." The reality is that most meaningful programs require the storage of multiple data.

### *Description of an array*

Consider software designed to keep track of attendance. Each student has between 30 and 45 days of class in a typical semester. So for every student, there are 30 to 45 different variables. Perhaps the naming convention is `day01`, `day02`, `day03`, ... , `day44`, and `day45`. This *could* get the job done, but it's a little messy because any time we have to track 45 variables, we're bound to make a mistake. And this is for just one student! In a class of 20 students, we would have to rethink our naming convention (and encumber many more variables). Now we are looking at something like `student01Day01`, `student01Day02`, `student01Day03`, …, `student01Day45`, then we have to have the next set of 45 variables: `student02Day01`, `student02Day02`, `student02Day03`, …, `student02Day45`. Then we pivot to `student03` and then `student04` and go all the way up to `student20Day45`. All in, we're looking at 900 variables. And keeping track of those is not attenable.

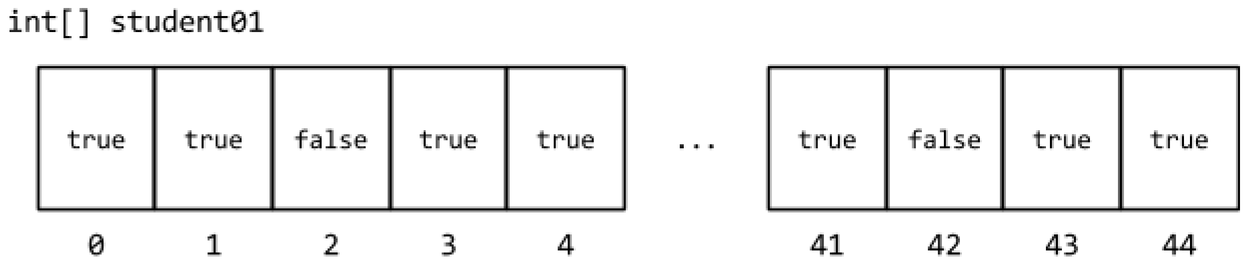
This is where **arrays** come in. They are one variable that can hold a lot of data! It's easier to visualize an array if you think of a row of lockers or a row of mailboxes:

*Arrays in real life*

In this fashion, the entire construct of the 15 mailboxes is really one object - it's a wooden plank nailed to a few wooden posts; but it's really just one giant construction that holds mailboxes. That's only one variable (in the words of the immortal Jack Black, "Dude! If you get the nachos stuck together, that's one nacho!")! It happens to be the case that this variable has room to hold 15 mailboxes. Each mailbox can hold one (and only one) datum - in this case, a piece of mail -, so this entire oversized mailbox can hold 15 pieces of mail.

### *Diagramming an array*

In the realm of Computer Science, we typically simplify the drawing of an array as just a whole bunch of adjacent boxes. In the example below, let's call the array `student01`. Let's imagine that each **element** in the array (that is, each box) is capable of holding one `boolean` value.



We'll be using this diagram a lot more, so get used to it! The word "`student01`" is the name of the array, each element has a value in it (since this is an array of type `boolean`, every value must be either `true` or `false` - in this case, it looks like the student was absent on the third day of class and the forty-third day of class), and the numbers on the bottom are the ***index numbers*** for the elements in the array.

Remember that arrays are "zero-indexed", so the first day of class in this example is really day 0. That really is a little confusing, because the first `false` appears over index 2 but it's really the third day of class - not the second. Likewise, the second `false` appears over the number 42 but is really the 43rd day of class.

## *Check Yourself*

1. In your own words, furnish an example of a program that would need an array.

2. Draw a diagram of an array named `grades` that can hold twelve values of type `double`. Make up your own values for each element.

## §8.2 Basic Arrays

### *Properties of arrays*

All arrays have a name, a length, and a data type. It turns out arrays are really good at storing multiple values, but not so good at storing data types of multiple kinds. They also aren't very good at all for changing the number of things they can hold (that is, if an array is created and can store 10 items, then we can't later change it to store 11 items - this one doesn't go to 11. It stays at 10).

It's also worth noting that arrays are zero-indexed. That means that the first item is actually number 0. That's peculiar, but get used to it because that's how arrays roll.

### *Declaring an array*

To create an array, we'll need to declare and assign it. Interestingly, we don't need to use the word "array", we'll just use the square brackets: `[ ]`

```
boolean[] student01 = new boolean[45];
```

This one line of code tells us a bunch of things:

- `boolean` tells us two things
  - The `[]` tells us that it actually *is* an array
  - That this array can only hold data that are of type `boolean`
- `student01` is the name of the array
- `new boolean[45]` tells us that the array can hold 45 items.

You may see a declaration that puts the square brackets *after* the variable name:

```
boolean student01[];
```

That's perfectly cromulent, too - it works just as well. But from a readability standpoint, putting the brackets after the data type is preferred.

This whole array thing *should* strike a chord with you:

```
String[] args
```

The parameter of the `main` method in any Java program is actually an array of `Strings`. It's a little complicated, but essentially the input that we've been typing in and having the `Scanner` read is really an array of `Strings` called `args`.

### *Hardcoding values in an array*

While most of the time the array will be loaded with values once the program is running, there are instances when an array is hardcoded with data. To do this, the declaration doesn't change but the assignment (the right hand side) looks a bit different:

```
String[] workWeek = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday"};
```

It isn't rare to see an array pre-loaded with values, but it also isn't super typical. You will know when you need to hardcode the values - don't worry.

Note that there are many different collections in Java - don't get confused by the canonical classes in Java such as `ArrayLists`, `LinkedLists`, *`Array`*, `Arrays`, *`ArrayType`*, *`ArrayReference`*, and many other classes. Don't worry - you'll get there one day. For now, just try to wrap your head around what an array is.

## *Check Yourself*

1. Write a segment of code that declares an array named `arrMatey` that can hold eight elements of type `String`.

2. Write code that would hardcode the following data into the array of `doubles` called `radioStations`:

```
88.5  91.3  95.1  103.5 105.7
```

## §8.3 Accessing Information in Arrays

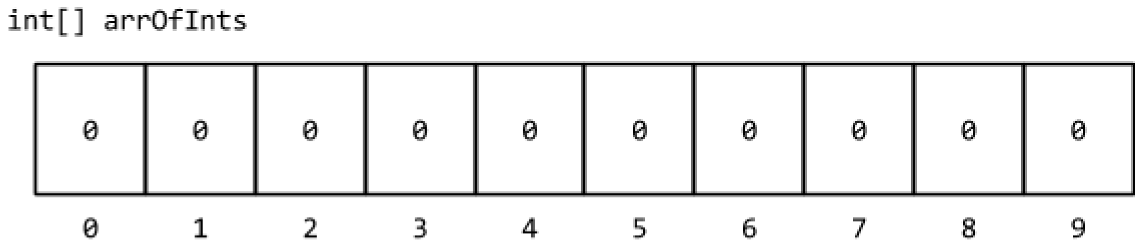
Knowing a bit about arrays is great in theory, but the practical way to handle arrays takes some technical skills.

### *Populating an array*

For starters, let's go over how to populate an array (that is, give it values - unless it is given values in the initial declaration like our previous example of the days of the week). By default, whenever an array is created, every element is filled with the default value that corresponds to the data type. For instance,

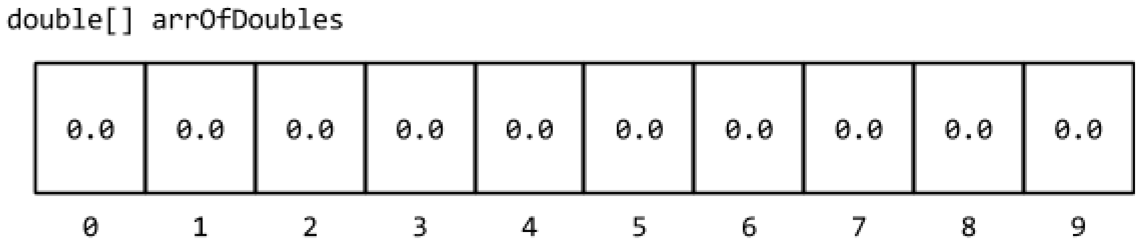
```
int[] arrOfInts = new int[10];
```

Looks like this:



. And an array of `doubles` called `arrOfDoubles` looks like this:

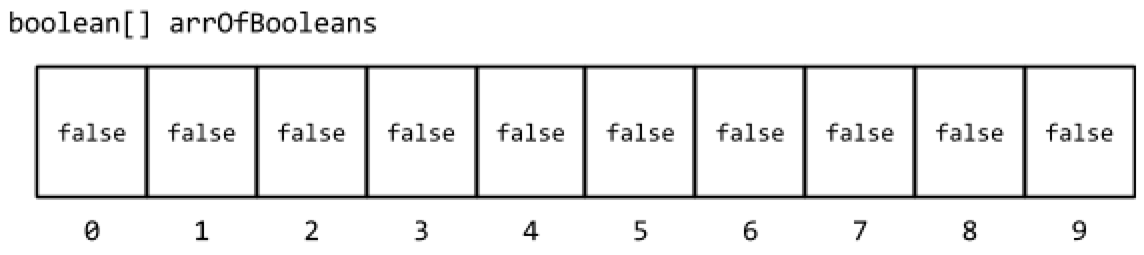
```
double[] arrOfDoubles = new double[10];
```



An array of `boolean` declared and assigned as follows:

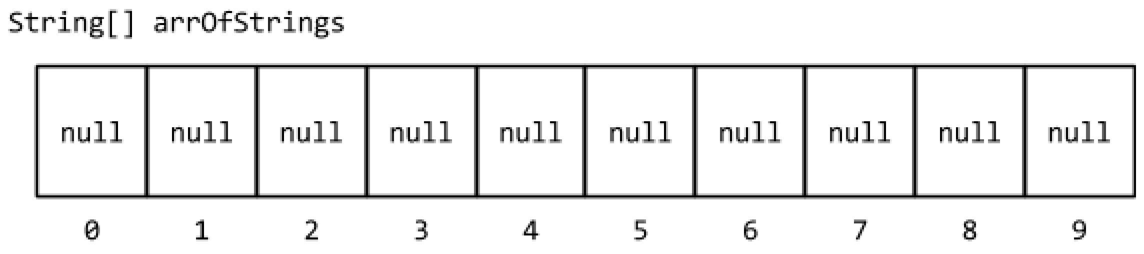
```
boolean[] arrOfBooleans = new boolean[10];
```

Looks like this:



And even though a `String` isn't a primitive data type, an array of `Strings` can still be created. We may expect that each element would be the empty string, or `""`, but instead an array of any object - not just `String` - is full of `nulls`.

```
String[] arrOfStrings = new String[10];
```



Knowing that an array is by default created with the default version of each type in every element, it is natural to want to put our own values in each element. Let's look at the array we talked about a minute ago - `arrOfInts`. There are two ways we could populate this array - we can have the code do it, or we can have the user do it. Let's look at both.

### Populating an array with values [using code]

In this example, let's agree that we want to populate the array with powers of 2. So each element will be 2 raised to a power. The clearest way to do this is to actually hand code this:

```
int[] arrOfInts = new arrOfInts[10];
arrOfInts[0] = 1;
arrOfInts[1] = 2;
arrOfInts[2] = 4;
arrOfInts[3] = 8;
arrOfInts[4] = 16;
arrOfInts[5] = 32;
arrOfInts[6] = 64;
arrOfInts[7] = 128;
arrOfInts[8] = 256;
```

```
        arrOfInts[9] = 512;
```

That’s kind of boring, but at least it’s clear. We can individually access any element just by putting the index number in the square brackets. By the way, we could have used a loop to automate this task - we’ll talk about iterating later on.

**Populating an array with values [from the user]**

```
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter value 0: ");
        arrOfInts[0] = scanner.nextInt();

        System.out.print("Enter value 1: ");
        arrOfInts[1] = scanner.nextInt();

        System.out.print("Enter value 2: ");
        arrOfInts[2] = scanner.nextInt();

        System.out.print("Enter value 3: ");
        arrOfInts[3] = scanner.nextInt();

        System.out.print("Enter value 4: ");
        arrOfInts[4] = scanner.nextInt();

        System.out.print("Enter value 5: ");
        arrOfInts[5] = scanner.nextInt();

        System.out.print("Enter value 6: ");
        arrOfInts[6] = scanner.nextInt();

        System.out.print("Enter value 7: ");
        arrOfInts[7] = scanner.nextInt();

        System.out.print("Enter value 8: ");
        arrOfInts[8] = scanner.nextInt();

        System.out.print("Enter value 9: ");
        arrOfInts[9] = scanner.nextInt();
```

Again, nothing super fascinating here, but it’s a good way to show how to **load** values from the user into specific slots in the array.

## *Outputting values of an array*

Outputting the values of an array, however, isn't as clear cut as you may think. For instance, in section 8.2 we talked about an array named `workWeek`:

```
String[] workWeek = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday"};
```

We could reasonably expect that a call to `System.out.println` with the `workWeek` as a parameter would output something like:

HOPEFUL OUTPUT:

```
Monday, Tuesday, Wednesday, Thursday, Friday
```

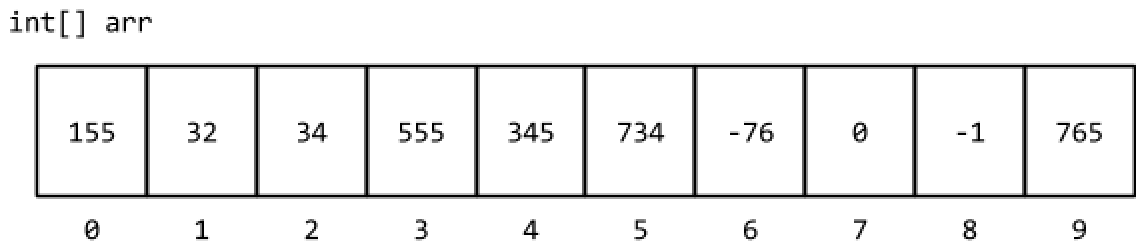
But instead, we get:

ACTUAL OUTPUT:

```
java.lang.String;@6d06d69c
```

As it turns out, any call to `System.out.print` with an object as the parameter will always output the address in memory that the object lives in (that is kind of the truth, but you won't find out I'm lying until much later).

So we have to output the values by referencing each element we want to output. Of course, this would be super easy with a loop (see section 8.4 if you can't wait), but let's do it item by item for the time being. Here are some basic lines of code that work well. Let's use the following diagram for the array called `arr`:



And let's look at a few different ways to retrieve datum from specific elements:

```
System.out.println("The first element is " + arr[0]);
// 155

System.out.println("The second element is " + arr[1]);
// 32

int length = arr.length;
System.out.println("The last element is " + arr[length]);
// ERROR! Arr.length = 10, so we can't output arr[10] without
// getting an ArrayIndexOutOfBoundsException

int length = arr.length-1;
System.out.println("The last element is " + arr[length]);
// 765

System.out.println("Pick a number between 0 and " + (arr.length-1)); // 5
int choice = scanner.nextInt();

// Make sure "choice" is a valid box in the array!
if (choice >= 0 && choice < arr.length) {
  System.out.println("The value in element " + choice + " is " + arr[choice]);
}
// 734
```

## *The `.length` Property*

The astute reader will notice that there was a quick little call to `arr.length` in that code. Every array has a length - that is, the number of boxes in the array. We typically refer to that as the **physical length**. So `arr` has a physical length of 10 (even though the greatest index is 9). Therefore, any call to `arr.length` will always be the number of cells in the array. Likewise, a call to `arr[arr.length-1]` will always reference the last element in the array.

I know that's confusing! Why wouldn't we be able to use `arr[arr.length]` to access the last element of an array? Well, it's because of the value of `arr.length`. If there are 10 elements, the value of `arr.length` is `10`. But the last element is really `arr[9]`. So if we attempt to reference `arr[arr.length]`, we are really referencing `arr[10]`, which is a big no-no.

## *Check Yourself*

1. Consider the following code:

   ```
   int[] arr = {-2, -1, 0, 1, 2};
   int element = arr[3];
   ```

   What is the value of `element`?

2. Consider the code segment below:

   ```
   int[] arr = {2, 4, 6, 8, 10};
   System.out.println(arr[1] + arr[4]);
   ```

   What is the output of the code?

3. Consider the code segment below:

   ```
   int[] arr = {1, 3, 5, 9, 11};
   System.out.println(arr[arr[1]]);
   ```

   What is the output of the code?

4. Consider the code segment below:

   ```
   int[] arr = {10, 20, 30, 40, 50};
   System.out.println(arr[5]);
   ```

   This code causes an error. What is the error, and why is it caused?

## §8.4 Iterating Through Arrays

With arrays comes the task of iterating through arrays. This means walking through every element of an array and checking the value stored in each one.

### *Examples of iterating through arrays*

For demonstrative purposes, let's look at three different times when it would be beneficial to iterate through an array:

1. If you want to output every element of the array to the screen (or a file, if you're super nerdy)

2. If you want to sum all the numbers in an array - perhaps to find the average

3. Searching an array for a specific value

4. If you want to look at every element in the array and find the biggest (or smallest!) value.

Typically this is done with a loop that starts at 0 and goes until the length of the array. It may seem hard to believe now, but there will be many times when you don't know the length of the array so you'll have to use the `.length` attribute. Don't worry - hopefully after the third example, this will make sense.

#### Outputting an array

Let's agree that we have an array named `arr`, and it holds a bunch of `Strings`. Actually, it doesn't really matter what `arr` holds because this code segment will work for any array. It also doesn't matter if we know how many elements are in `arr` - the code will *still* work even if the length is not known.

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

System.out.println();
```

The `for` statement almost never changes - the loop starts at `0` and goes to `arr.length` (or `whateverTheNameOfTheArrayIs.length`). Note that there's a `System.out.println()` command after the loop iterates - this is just to add a new line so that if the program had more output, it wouldn’t be on the same line as the values we just output.

### Summing the array

While we most likely won’t write too many programs that add all the values in an array, summing them up is a valuable lesson while learning how to step through every element in an array.

Let’s think about this - we need to walk through an array (that’s what we call it when we **traverse** every single element in an array), look at the value stored in each element of the array, and then keep a running total.

So let’s imagine we have an array called `arrayOfNumbers`, and it holds a bunch of `doubles`.

```
double[] arrayOfNumbers = new double[10];
// <code to populate each cell of the array with a double>
// Let’s not code this now - let’s assume that it has already
// been populated with values somehow
//
// We need a variable to keep track of the running total.

double total = 0;
// Now that we have an array with values in it and a variable to
// keep track of the total, let's use a loop to walk through the
// array

for (int i = 0; i < arrayOfNumbers.length; i++) {
  // We need to peek into the cell and add the value of that
  // element to the running total - that's "total"

  total += arrayOfNumbers[i];

}

// By the time we get to the end, "total" should be the sum of
// all the numbers in the array. Holy cow! That wasn't so bad!
```

So the value of `total` is actually the sum of all the numbers in the array. We’ve done our job!

But wait - there's more! It's super easy right now to compute the average of all those numbers. We just have to divide the sum by the number of terms:

```
System.out.println("The average is: " + (total/arrayOfNumbers.length));
```

You may be thinking that dividing by `arrayOfNumbers.length` isn't the proper number to divide by because it is off by one. But it *is* the right value - remember that `arrayOfNumbers.length` refers to the physical length of the array; how many elements are actually stored in there.

### Searching an array for a specific value

This is one of the most popular things to do with arrays and is worth investment in understanding it thoroughly. We can think of it as a giant game of Go Fish - we have a whole bunch of cards in our hand, someone asks us for a specific one, we look at each card, and either find it or tell our opponent to go fish. Let's imagine we have an array, called `arr`, and it contains many items (for simplicity, let's say it has 100 items). Maybe we know that `arr` holds a bunch of `int`s, and we are curious about if one specific one, say 33, is in the array.

Well, first we'd have to start at the first element and then look to see if the item in index 0 is the one we are looking for. If it is, great! We can stop looking. But if the first element is not 33, we need to move to the second element (and again look to see if the item in it is 33). If we don't find a 33, we move on to the next. And the next. And either we stop when we find it, or we get to the end and realize we haven't found 33, so it must not be in the array. Let's see what that looks like:

```
// Declare and assign the array
int[] arr = new int[100];

// Randomly populate 'arr' - is 33 one of those numbers? Who can say!
for (int i = 0; i < arr.length; i++) {
  arr[i] = (int)(Math.random()*100) + 1;
}

// Create a boolean flag and set it to 'false' - we'll change it if we
// find 33
boolean found = false;
```

```
// Start to iterate through the array and look for 33
// If we find 33, let's end the search with a break
for (int i = 0; i < arr.length; i++) {
  if (arr[i] == 33) {
    found = true;
    System.out.println("33 resides at index " + i);
    break;
  }
}
```

Now there are a few different search algorithms. This is the most basic - start from the first, and peek into each sequential element until you find it or you get to the end. This is what's known as a **sequential search** or a **linear search**. A more robust search, a **binary search**, will be explored in future classes.

### Finding the maximum value in an array

Again, this is mostly an exercise in learning about arrays (although there are a few cases when you'll want to know this stuff). For this example, let's have an array called `arr` and let's have it hold a bunch of `int`s. Furthermore, let's assume every element has a value in it.

So here's the game plan. Let's look at the first element in the array, and let's consider it to be the biggest number in the array. It's the biggest number we've encountered so far as we traverse the array, so by definition it's the largest number we know about in the array. I know - and you know - that by the time we've visited every element, the value of the biggest number will most likely change. But for right now, it's the biggest number. We'll go through the array, element by element, and compare it to the biggest number. If it's the case that the current element has a number bigger than the biggest number, let's reassign the biggest number to be that value.

```
// We need to have a variable to hold the biggest number. Let's
// go ahead and set it equal to the first element (since it needs
// to have a value). This gives us the further optimization of
// not having to start at element 0 because we already consider
// the value in element 0 to be the biggest number when this
// starts.

int biggestNumber = arr[0];


// Now let's walk through the array and look at each value. If
// the value happens to be bigger than our current biggest
```

```
        // number, let's reassign the value of biggestNumber to be
        // whatever we happen to be looking at

        for (int i = 1; i < arr.length; i++) {
            if (arr[i] > biggestNumber) {
                biggestNumber = arr[i];
            }
        }

        System.out.println("The biggest number is: " + biggestNumber);
```

These three different examples have similarities - the `for` statement in the loop is almost identical. When iterating through an array, it is super important to visit *every single element* in the array, starting at index 0 and going through the last element.

## *for-each* Loops

As promised in the chapter about loops, we will now look at the last kind of loop - a “`for-each`” loop. These loops are designed for programmers like you who want to iterate through arrays, but don’t like all the clutter. They have the additional benefit of reducing mistakes because the actual iterator is hidden.

Here is a simple code segment that uses a traditional `for` loop to output values in an array:

```
int[] testArray = {1, 3, 5, 6, 7};

for (int i = 0; i < testArray.length; i++) {
  System.out.print(testArray[i] + " ");
}
```

It’s pretty straightforward - the program iterates through the array and outputs each element. But since iterating through arrays is such a common task, we can tighten up the code with a `for-each` loop:

```
int[] testArray = {1, 3, 5, 6, 7};

for (int i : testArray) {
  System.out.print(i + " ");
}
```

In this case, the `i` is not really the iterator (in fact, all iteration is handled under the hood and we couldn't even access that information if we wanted to). The `i` is really the value of each element. This loop control expression is read as, "For every element in `testArray`, let's consider `i` as the actual value in each element."

## *Check Yourself*

For questions 1 - 4, consider the following array declaration and assignment:

```
int[] arrOfInts = {1, 2, 3, 4, 5};
```

1. What is the value of `arrOfInts.length`?

2. What is displayed on the screen when the following code is executed?

```
System.out.println(arrOfInts[3]);
```

3. What is displayed on the screen when the following code is executed?

```
System.out.println(arrOfInts[arrOfInts[3]]);
```

4. There is one error in this code that will cause the program to crash. What is it?

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

# Stay tuned for CSC-190!

CSC 190 CS2: Object-oriented Software Development (4-1) 4 hrs.

CS2: Object-Oriented Software Development covers algorithm development and object-oriented design and development for large-scale software and graphical user interfaces (GUIs). This course is the second in a series of three required programming courses for a traditional computer science degree. Topics to be covered include objects and classes, procedural vs. object-oriented programming, reference data types, class libraries, class design, class abstraction and encapsulation, inheritance and polymorphism, exception handling, abstract classes, graphical user interfaces (GUIs), and event-driven programming. Prerequisite: CSC 115 with a grade of C or better. View Course Syllabus

**This one little trick will make you never store data the same way again! Take CSC-190 to find out why!**

# APPENDIX A: Relevant API

*The following API are selected from different classes explored in class. Note that the APIs are not complete - there are many methods not discussed in this document because they aren't used as much. However, links to the official API are provided.*

`Arrays` | `Math` | `Random` | `Scanner` | `String`

## *Arrays*[1]

**PRO-TIP**

The `Arrays` class does not need to be instantiated to use it. You know how with `Scanner` you have to create a new `Scanner` in every program?

```
Scanner scanner = new Scanner(System.in);
```

That's called instantiating (because you are creating an *instance* of the `Scanner`). You don't have to do that with the `Arrays` class because it is *static*. Stay tuned for what that means - it's on the horizon for now.

**`Arrays.copyOf()`**

This method takes in two parameters (the original array, and the length of the new array) and will return a new array with the values of the original array followed by the default values of a new array. Consider if you had an array of `int`s that was size 5 and you used `copyOf()`:

```
int[] arrayOne = {1, 3, 5, 3, 6};
int[] arrayTow = Arrays.copyOf(arrayOne, 7);
// [1, 3, 5, 3, 6, 0, 0]
```

Note that if the second parameter - the length of the new array - is less than the length of the original array, the new array would only have the first few values (`copyOf()` will truncate the array).

**`Arrays.sort()`**

[1] For a complete listing of all the sweet things that the `Arrays` class can do, check out: https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html

This will sort the array. Note that this is a void method; the method does not return anything. Compare the correct way to sort an array using this method versus the incorrect way:

```
// How to sort an array:
int[] testArray = {5, 2, 8, 6, 1};
// [5, 2, 8, 6, 1]
Arrays.sort(testArray);
// [1, 2, 5, 6, 8]

// How NOT to sort an array:
int[] testArray = {5, 2, 8, 6, 1};
testArray = Arrays.sort(testArray);
// This code will not work because the call to
// Arrays.sort does not return an array.
```

**`Arrays.toString()`**

This method helps you output the contents of an array. Compare the proper way to output the contents of an array with the improper way:

```
// How to print an array:
int[] testArray = {5, 2, 8, 6, 1};
System.out.print(Arrays.toString(testArray));

// How NOT to print an array:
int[] testArray = {5, 2, 8, 6, 1};
System.out.print(testArray);
// This will just output the memory address of the array
```

## *Math*[2]

**PRO-TIP**

The `Math` class also does not need to be instantiated to use it. The `Math` class has a few constants we will use often, and a few methods that are helpful.

[2] For a complete listing of all the sweet things that the `Math` class can do, check out: https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

**`Math.PI`**

This will return the best decimal equivalent of Pi ($\pi$). Some different systems (versions of Java, specifications, etc.) may return a few more or less digits, but `Math.PI` is the best approximation of Pi).

```
    System.out.println("The value of Pi is: " + Math.PI);
    // The value of Pi is: 3.141592653589793
```

**`Math.E`**

Same thing as Math.PI, although it returns the base of the natural logarithm.

```
    System.out.println("The value of e is: " + Math.E);
    // The value of e is: 2.718281828459045
```

**`Math.abs()`**

Calling the `Math.abs()` function will return the absolute value of the argument that is passed in. But here's the cool thing - there are actually four different `Math.abs()` functions in the `Math` class (a cool example of *overloading*, but that's for another day). You can pass `Math.abs()` an `int`, and it will return the absolute value of that `int` as an `int`. You can also pass it a `float` and get a `float` back, a `double` to get a `double` back, and a `long` to get a `long` returned.

```
    System.out.println(Math.abs(4.5);
    // 4.5
    System.out.println(Math.abs(-3);
    // 3
```

**`Math.max()`**

This method takes two numbers (`double`, `float`, `int`, or `long`) and return the greater of the two. Just like `Math.abs()`, there are four different methods (based on the parameters that are input). If two numbers are passed in (one `int` and one `double`), Java will automatically promote the `int` to a `double` and the return type will be a `double`.

```
    System.out.println(Math.max(15, 7.1);
    // 15.0
```

It's noteworthy to mention that if you have three numbers, you can use a call to `Math.max()` as one of the parameters for another `Math.max()`. Pro-tip: this is true for any argument - you can pass in an expression that evaluates to the type of identifier that is required by the method.

```
System.out.println(Math.max(4.9, Math.max(2, 6));
// 6.0
```

In the example above, the computer will analyze the call to `Math.max(2, 6)` first, and the "winner" will go up against `4.9`.

### `Math.min()`

Just like `Math.max()`, but this returns the smaller of the two values.

### `Math.pow(double, double)`

This is the real way to do exponents in Java. The first parameter is the base, and the second is the exponent. A call to `Math.pow()` requires to numbers - each a `double` - so if you pass in two variables of type `int`, the result will still be a `double`.

### `Math.random()`

This method requires no parameters, and will return a `double` between `0` (inclusively) and `1` (exclusively). That means that you may get `0.0`, `.45023`, `.9596942`, `.9999999`, but you'll never get `1`. If you want a number that is `1` or bigger, you'll need to multiply by 10 (or 100, or 1000, etc.).

```
System.out.println(Math.random());
// 0.6017376710307702
```

The above example will generate a random `double` greater than or equal to `0`, but less than `1`.

```
System.out.println(Math.random()*10);
// 7.1687284712872543
```

The above example will generate a random `double` greater than or equal to `0`, but less than `1`. After that, the number will be multiplied by 10 to make it bigger - the result is that you'll get a number between `0` and `9.999999999999999`...

```
System.out.println((int)(Math.random()*10));
// 3
```

The above example will generate a random `double` greater than or equal to 0, but less than 1. After that, the number will be multiplied by 10 to make it bigger - and then it will be cast as an `int`. So the smallest the result will be is `0`, and the largest is `9`.

```
System.out.println((int)(Math.random()*10) + 1);
// 10
```

The above example will generate a random `double` greater than or equal to 0, but less than 1. After that, the number will be multiplied by 10 to make it bigger - and then it will be cast as an `int`. And then it will have `1` added to it, so the smallest number is `1` and the largest it can be is `10`.

### `Math.round()`

`Math.round()` takes in a `float` (although if you pass in a variable that can be promoted to `float`, such as an `int`, `byte`, `short`, `long`, `double` - they will all work) and will return a `long`, rounded to the nearest whole number.

```
System.out.println(Math.round(4.49995849));
// 4
```

### `Math.sqrt()`

The square root method in the `Math` class takes in a `double` and returns a `double` (it's okay if you give the `sqrt()` method an `int` - it will automatically be promoted to a `double`). The result is the correctly rounded, positive square root of the parameter passed in.

## *Random*[3]

**PRO-TIP**

The `Random` class needs to be instantiated to use it. There are two different constructors you can use, but in general you will use the default constructor:

```
Random random = new Random();
```

That's called instantiating (because you are creating an *instance* of the `Random` class). Just so you know - there really is no such thing as a random number when dealing with computers (in fact, some people think human brains cannot even generate purely random numbers). In the realm of computer science, we consider the *psuedorandom* numbers to be as close as we can get to genuinely random.

[3] For a complete listing of all the sweet things that the `Random` class can do, check out: https://docs.oracle.com/javase/8/docs/api/java/util/Random.html

You can also “seed” the `Random` object when constructing it - see the Java Vocabulary document entry for “Constructor”. This means that random numbers will be generated, but they will be the same every time you run the program - this is helpful for debugging.

**`nextBoolean()`**

A call to this will return a boolean value that is either true or false. Similar to flipping a coin.

```
Random random = new Random();

System.out.println(random.nextBoolean());
// true or false

System.out.println(random.nextBoolean());
// true or false

boolean willRun = random.nextBoolean();
if (willRun == true) {
      System.out.println("HEADS");
} else {
      System.out.println("TAILS");
}
```

**`nextDouble()`**

The same thing as `nextBoolean`, but it returns a `double`. The catch is that this will return a number between `0.0` and `1.0`.

```
Random random = new Random();
double randomDouble = random.nextDouble();
System.out.println(randomDouble);
// Maybe 0.435345 or 0.999345345 or 0.0 or 0.3456474525
```

**`nextInt()`**

The same thing as `nextBoolean`, but it returns an `int`. This number will be between `-2,147,483,648` and `2,147,483,647`.

```
        Random random = new Random();
        int randomInt;
        randomInt = random.nextInt();
        System.out.print(randomInt);
        // Your guess is as good as mine!
```

**nextInt(int)**

The same thing as `nextInt`, but it returns an `int` within a specified cap! This number will be between `0` and the cap (that is, it can be `0`, and it can be anything between 0 and the cap, but not the cap).

```
        Random random = new Random(50);
        int randomInt;
        randomInt = random.nextInt();
        System.out.print(randomInt);
        // Maybe 0, maybe 1, maybe 33, maybe 49, but NOT 50!!!!
```

**nextLong()**

The same thing as `nextBoolean`, but it returns a `long`. The catch is that this will not return a number in the entire range of `long` numbers - `Random` only uses 48 bit seeds, so the result will not be 64 bit.

**setSeed(long)**

This method will set the seed of the random number generator. Useful for debugging.

```
        Random random = new Random();
        random.setSeed(33);
        System.out.println(random.nextInt());
        // The output will be the same every time this program runs
```

## *Scanner*[4]

**close()**

This will “close” the `Scanner`. Creating a scanner but not closing it will cause a *warning*. This warning will not prohibit the program from running, but will be annoying in Eclipse as it will give a yellow underline. Closing the `Scanner` will just “delete” it from memory.

[4] For a complete listing of all the sweet things that the `Scanner` class can do, check out: https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html

**`hasNext()`**

This returns either `true` or `false`; it's essentially the scanner looking ahead to see if there is more input. The `hasNext()` method will look to see if there is a *token* that left to be examined, or if the input *stream* is done. Consider these examples:

```
// "streamOfIntegers" is a stream that contains integers. The
// input stream is: 77 33 23

Scanner scanner = new Scanner(streamOfIntegers);

if (scanner.hasNext() == true) {
  System.out.println(scanner.next());
}
// 77 will be output on the screen

if (scanner.hasNext() == true) {
  System.out.println(scanner.next());
}
// 33 will be output on the screen

if (scanner.hasNext() == true) {
  System.out.println(scanner.next());
}
// 23 will be output on the screen

if (scanner.hasNext() == true) {
  System.out.println(scanner.next());
}
// Nothing more will be output on the screen
```

It's a bit easier to see this in the framing of a while loop. Oftentimes, you will receive a stream of data (that is, a bunch of tokens separated by spaces) and will need to examine each element in that stream until there are no more elements. The easiest way to do that is with a while loop that terminates if there are no more tokens.

```
// "streamOfIntegers" is a stream that contains integers.
Scanner scanner = new Scanner(streamOfIntegers);

while (scanner.hasNext()) {
      System.out.println(scanner.nextInt());
}
// Every integer in the stream will be output on a new line
```

**`next()`**

This is a way to read in part of a line of text. In the biz, we call different pieces of input in one stream (so, like, multiple words or words and numbers all on one line) tokens. In class, we spend a lot of time learning about `nextLine()`, which will read in an entire line of text. But let's say you only want to read in part of the text (that would be a token!). Then you can use `next()`. The `next()` method will read in text from wherever the `Scanner` object is and take in the next token (by default, tokens are separated by spaces).

```
Scanner scanner = new Scanner(System.in);
System.out.print("Enter your first name, last name, and jersey number: 
");
// Larry Bird 33
String firstName = scanner.next(); // Larry
String lastName = scanner.next(); // Bird
int jerseyNumber = scanner.nextInt(); // 33
System.out.println(firstName + " " + lastName + " wears " + 
jerseyNumber);
// Larry Bird wears 33
```

**`nextDouble()`**

When used in an assignment statement with a variable of type `double`, a call to the `nextDouble` method will take the input from the user - whatever they typed in on the keyboard - and store the value in the variable. Bad things happen if you ask the user to type in a number, use `nextDouble()`, but the user types in a `String`.

```
Scanner scanner = new Scanner(System.in);
double number;
System.out.print("Enter a number: ");
number = scanner.nextDouble();
// Stores what the user types in as 'number'
```

**`nextInt()`**

When used in an assignment statement with a variable of type `int`, a call to the `nextInt()` method will take the input from the user and store the value in the variable.

```
Scanner scanner = new Scanner(System.in);
int number;
System.out.print("Enter a number: ");
number = scanner.nextInt();
// Stores what the user types as 'number'
```

But beware! There is a danger to relying on `nextInt()`! Know that merely calling nextInt() will push the scanner to read the nextInt(). So consider a data stream of 50, 40, 30, and 30.

```
while (scanner.hasNext()) {
if (scanner.nextInt() > 55) {
      System.out.println("The number is greater than 55");
} else if (scanner.nextInt() > 45) {
      System.out.println("The number is greater than 45");
} else if (scanner.nextInt() > 35) {
      System.out.println("The number is greater than 35");
} else if (scanner.nextInt() > 25) {
      System.out.println("The number is greater than 25");
}

// no output!?!?!?
```

The issue here is that the first number, `50`, is called by the first call to `scanner.nextInt()`. Since `50` is not greater than `55`, we *want* to compare `50` to `45`. But this is not what happens! Since `scanner.nextInt()` was called in the first `if` statement, the scanner has moved to the next token in the stream (`45`) - just by virtue of having the `nextInt()` method called!

The correct way to do this is to assign the value of `scanner.nextInt()` to a variable, and use that variable for the nested `if` statements:

```
while (scanner.hasNext()) {
int temp = scanner.nextInt();
if (temp > 55) {
      System.out.println("The number is greater than 55");
} else if (temp > 45) {
      System.out.println("The number is greater than 45");
} else if (temp > 35) {
      System.out.println("The number is greater than 35");
} else if (temp > 25) {
      System.out.println("The number is greater than 25");
}

// The number is greater than 45
```

**`nextLine()`**

When used in an assignment statement with a variable of type `String`, a call to the `nextLine()` method will take the input from the user and store the value in the variable. The difference between `nextLine()` and `next()` is that `nextLine()` will read in *all* the input to the end of the line, whereas `next()` will only read from the current position and go to the next *token* (basically a space). Because `String` is fairly inclusive, if the user types in a number instead, `nextLine()` will still assign that number as a `String`. The program will run, but that number is a `String` so the computer cannot do calculations on it.

```
Scanner scanner = new Scanner(System.in);
String name;
System.out.print("Enter a number: ");
name =  scanner.nextLine();
// Stores what the user types as 'name'
```

**`nextBoolean()`**

Just like all the other ones, except it returns a `boolean`.

**`nextByte()`**

Just like all the other ones, except it returns a `byte`.

**`nextFloat()`**

Just like all the other ones, except it returns a `float`.

**`nextLong()`**

Just like all the other ones, except it returns a `long`.

**`nextShort()`**

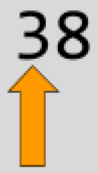
Just like all the other ones, except it returns a `short`.

**PRO-TIP**

When using a `Scanner`, sometimes weird things happen if you scan in numbers and then words. Here's the deal. Think of a `Scanner` as a program that looks at each piece of data in an input stream (when you punch keys on the keyboard, that's an input stream). This explanation may be a bit of an oversimplification, but bear with me.

```
Scanner scanner = new Scanner(System.in);
System.out.print("Enter your age: "); // user inputs 38
```

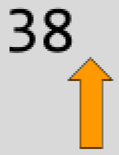
Well, at this point, the input stream looks like this (the orange pointer is where the `scanner` currently is):



```
    int age = scanner.nextInt();
```

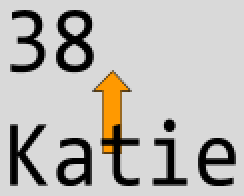
Now when this line of code executes, the scanner will read the input stream until it captures an `int`. So we would expect the `scanner` to read the `38` and then finish up it's job.

Which it does. So the `scanner` is now after the `38`.

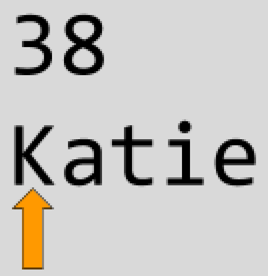


The problem is, it's still on that line of text. So if we ask it to read the next line with a call to `nextLine()`, the `scanner` will still be on the first line:

```
    System.out.print("Enter your name: "); // user inputs "Katie"
    String name = scanner.nextLine();
```

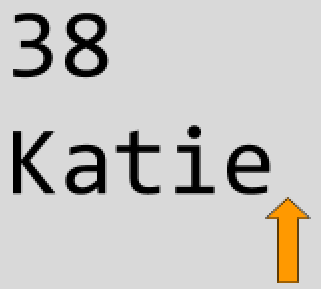


So the issue here is that the call to `scanner.nextLine()` is going to read the input up until it gets to a point where it has to go to the next line. As it turns out, the pointer is at a next line indicator right now, so the call to `scanner.nextLine()` is going to read in that next line mark, and nothing else.



So now the `scanner.nextLine()` code has executed, and we *want* the software to have read "Katie", but instead it read the next line indicator on the previous line. So, we have to invoke `nextLine()` *again* to have the pointer travel to the next next line indicator.

```
name = scanner.nextLine();
```



In this case, the input has now been read and stored. So all is good with the world. For the record, here is the complete code to scan in a number and then a `String`. Note that if you read in a `String`, there is no need to go to the next line before reading in a number - the call to read the `String` - `nextLine()` - will automatically put the pointer at the beginning of the next line (where the data for the number to read in is).

```
Scanner scanner = new Scanner(System.in);
System.out.print("Enter your age: "); // user inputs 38

System.out.print("Enter your name: "); // user inputs "Katie"
scanner.nextLine();
String name = scanner.nextLine();
```

## *String*[5]

**`charAt(int)`**

This method, when called on a `String`, will return the character at the specified index of the `String`. It is important to note that the index starts at 0 (so the seventh character is really at index six). It will return a `char`.

```
String testString = "This is a test String";
char position;
position = testString.charAt(6);
System.out.println(position); // s
```

**`compareTo(String)`**

This method will compare two variables of type `String`, and return a number. Yes, you read that right. A number is the answer. That number is the difference of the Unicode value of the first digit that is not the same in both `String` variables. Huh?

So, let's look at the two `String` variables, `s1` and `s2`:

```
String s1 = abcD;
String s2 = abcd;
```

Well, when a call to `System.out.print(s1.compareTo(s2))` is made, then the computer looks at the first character in each `String` - in this case, '`a`'. The computer will subtract the numerical value of the second '`a`' from the first '`a`'. Since both of the '`a`' characters are the same - `97` - the difference is `0`. So the computer moves on to the second character in both of the `String` variables and subtracts them. Well, '`b`' evaluates to `98`, so the difference between both the '`b`' characters is again, `0`. So the computer looks at the third character, '`c`'. Once more, the difference between the two decimal equivalents of each of these characters is `0`, so the computer moves on. Here's where things get interesting. The decimal equivalent of '`D`' is `68` whereas the decimal equivalent of '`d`' is `100`! So `68 - 100` yields `-32`. This means that not only are the two variables not the same, but that the first variable, `s1`, comes alphabetically (lexicographically) before `s2`.

That's true in any case - if the number returned after comparing two `String` variables is negative (no matter what the quantity), the first variable comes before the second. And vice versa.

[5] For a complete listing of all the sweet things that the `String` class can do, check out: https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

So the following code will reliably arrange two words alphabetically:

```
Scanner scanner = new Scanner(System.in);

System.out.print("Enter a word: ");
String word1 = scanner.nextLine();

System.out.print("Enter another word: ");
String word2 = scanner.nextLine();

if (s1.compareTo(s2) == 0) {
      System.out.println("The two words are equivalent.");
} else if (s1.compareTo(s2) < 0) {
      System.out.print(s1 + ", " + s2);
} else {
      System.out.print(s2 + ", " + s1);
}
```

**`compareToIgnoreCase(String)`**

Same exact thing as `compareTo()`, although case is ignored.

**`contains(String)`**

This method will check to see if the parameter, `str`, is part of the `String`. The `contains()` method will return a `boolean` value (`true` if `str` is part of the `String`, and `false` otherwise).

```
String s1 = "Hello world!");
String s2 = "lo";
if (s1.contains(s2) {
      System.out.println("The expression " + s2 + " is found in " +
s1);
} else {
      System.out.println("The expression " + s2 + " is NOT found in " +
s1);

// The expression lo is found in Hello World!
```

**`equals(Object)`**

When invoked, this will return `true` (a `boolean`) if the original `String` and the `String` passed in share the same characters in the same order (`false` otherwise). You would think that the parameter - of type `Object` - should be a `String`, but it is not. This is because of overriding inherited methods - a story for another day.

```
String firstString = "ABCdef";
String secondString = "ABCdef";
String thirdString = "abcdef";

System.out.println(firstString.equals(secondString)); // true
System.out.println(firstString.equals(thirdString)); // false
System.out.println(firstString.equalsIgnoreCase(thirdString)); // true
```

Note that this method is only valuable for discovering if two `String` variables are equal - it does not really give enough information if you are trying to sort the variables or alphabetize them. For that, you'll need to look at `compareTo()`.

**`equalsIgnoreCase(String)`**

Same thing as `equals()`, although this method does not differentiate between uppercase or lowercase.

**`indexOf(char)`**

Returns the position (zero-indexed!) of the first occurrence of the char that is passed in. If the char does not appear in the String, a value of -1 is returned. Otherwise, return an int that is the index of that char.

```
String testString = "In a galaxy far away...";
System.out.println(testString.indexOf('a')); // 3
System.out.println(testString.indexOf('q')); // -1
```

**`length()`**

Simply returns the length, as an `int`, of the `String`. This is just how many characters are in the `String` (so don't get confused about zero-indexes - this is a pure, human, natural way of counting.

```
String firstString = "This is a test";
System.out.println(firstString.length()); // 14
```

**`substring(int)`**

This will return a `String` - most likely smaller than the original `String` - that starts at the position passed in and goes until the end. Recall that any `String` is zero-indexed.

```
String testString = "Biggest ball of Twine.";
System.out.println(testString.substring(11)); // l of Twine.
```

**`substring(int beginIndex, int endIndex)`**

Very similar to the Substring method above, except that it accepts *two* `int` variables as input. The `String` that is returned will start at the first `int`, and end at the second `int` (although it is non-inclusive for the second parameter).

```
String testString = "Like a giant carbonated soda.";
System.out.println(testString.substring(11, 17)); // t carb
```

**`toLowerCase()`**

Returns a `String` that is a lowercase version of the original `String`. Note that this does NOT change the value of the original `String` (unless you specifically tell it to).

```
String testString = "Don’t YOU know that other kids are..."
System.out.println(testString.toLowerCase());
// don’t you know that other kids are...

System.out.println(testString);
// Don’t YOU know that other kids are...

testString = testString.toLowerCase(); // reassign the String
System.out.println(testString);
// don’t you know that other kids are...
```

**`toUpperCase()`**

Returns a `String` that is an uppercase version of the original `String`. Note that this does NOT change the value of the original `String` (unless you specifically tell it to).

```
String testString = "You make me wanna staple waffles to my face!";
System.out.println(testString.toUpperCase());
// YOU MAKE ME WANNA STAPLE WAFFLES TO MY FACE!

System.out.println(testString);
// “You make me wanna staple waffles to my face!

testString = testString.toUpperCase(); // reassign the String
System.out.println(testString);
// YOU MAKE ME WANNA STAPLE WAFFLES TO MY FACE!
```

**trim()**

Returns a `String` that has the leading and trailing whitespace removed (this can eliminate extra spaces before and after input).

```
String testString = "     Space… The final frontier...     ";
System.out.println(testString.trim());
// Space… The final frontier...

System.out.println(testString.length()); // 40
testString = testString.trim();
System.out.println(testString.length()); // 30
```

# APPENDIX B: Reserved Words

## *RESERVED WORDS*

There are some basic words in Java that have special meaning. For instance, “System” in System.out.println() has a specific meaning, and programmers cannot override that meaning.

| | | | | |
|---|---|---|---|---|
| `abstract` | `do` | `if` | `private` | `this` |
| `assert` | `double` | `implements` | `protected` | `throw` |
| `boolean` | `else` | `import` | `public` | `throws` |
| `break` | `enum` | `instanceof` | `return` | `transient` |
| `byte` | `extends` | `int` | `short` | `true` |
| `case` | `false` | `interface` | `static` | `try` |
| `catch` | `final` | `long` | `strictfp` | `void` |
| `char` | `finally` | `native` | `super` | `volatile` |
| `class` | `float` | `new` | `switch` | `while` |
| `const` | `for` | `null` | `synchronized` | `continue` |
| `default` | `goto` | `package` | | |

# APPENDIX C: Java Vocabulary

## *Abbreviations*

**API**
*Application Programmer Interface*
Think of the API as an instruction booklet for code. In iOS, the API informs developers of the functions that they have access to (for instance, there is an API for the camera on iPhones, so developers can use certain functions of the camera. However, there is no API for the Messages app, which is why app developers can't access any messages. This is not true for Android - there *is* an API for messages. Consequently, there are dozens of apps you can get that allow you to manage your messages (PushBullet and MightyText are some of them).

**IDE**
*Integrated Development Environment*
An IDE is software that gives you the tools to program Java. This shouldn't be confused with the JDK (that gives you the tools to compile and run Java) - the IDE helps you develop programs. Eclipse is a good example of an IDE. Most IDEs have an editor (where you type the code in - in Eclipse, the editor has features like autocomplete, code-collapsing, and partial compilation that underline incorrect code).

**JDK**
*Java Development Kit*
The Java Development Kit includes all the tools you need to develop in Java. Not only does it include the Java Runtime Environment (JRE), but all the other tools - like a compiler - that you'll need. If you download the JDK, you don't need to also download the JRE (it's included!).

**JRE**
*Java Runtime Environment*
Sometimes Java doesn't come installed on computers. If your computer isn't Java enabled, you'll need to download this if you want to run Java programs. It's not an app that you run on your computer; the JRE runs in the background.

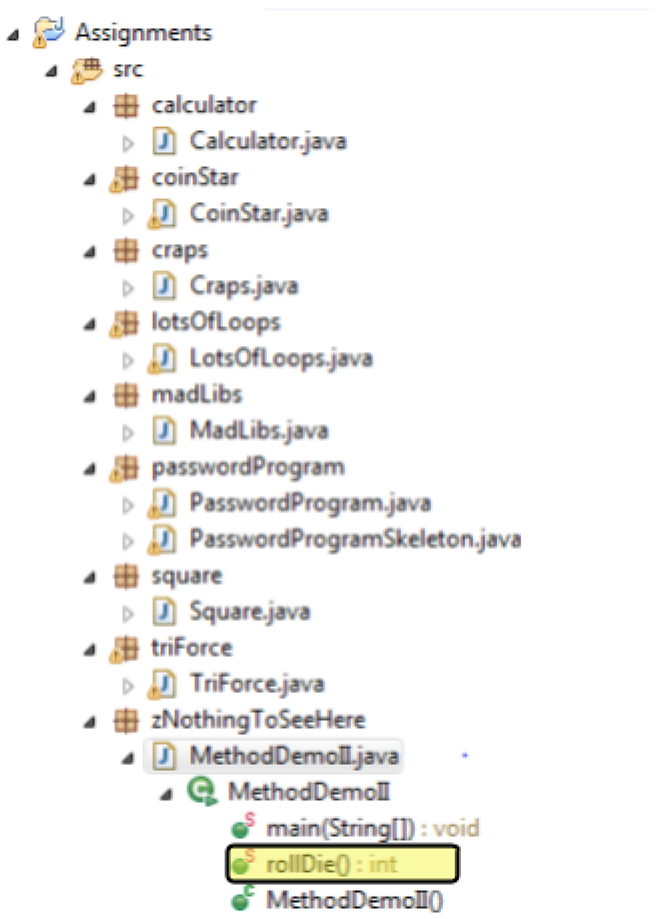
**JVM**
*Java Virtual Machine*
The Java Virtual Machine is what runs on your computer and allows you to execute Java

programs. It takes in a `.class` file (bytecode) and converts it to 1s and 0s. The JVM is specific to the platform it is running on (MacOS, Windows, mobile devices).

## *Vocabulary*

**Access Modifier**
Part of a method signature; this declares what access other classes have to the method. There are four options: `public`, `private`, `protected`, and default (that is, if no access modifier is listed).

- `public` means any other class can see (and use) the method
- `private` means only the class the method is defined in can see (and use) the method
- `protected` means any class in the package *and* any subclass of a class in the package (even if that subclass is in a different package) can see (and use) the method
- default means any class in the package can see (and use) the method



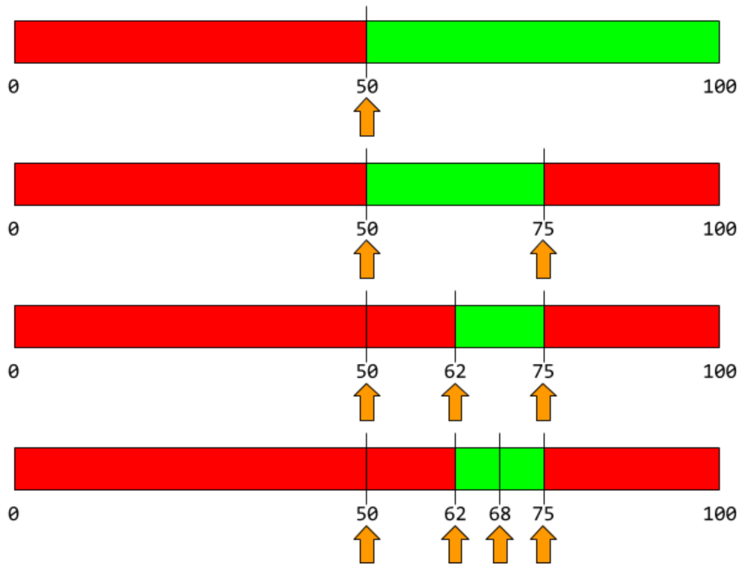
In this example, if `rollDie()` is `public`, any of the classes in the project `Assignments` can use `rollDie()` (we may have to instantiate the `MethodDemoII` class, but that's easy). If `rollDie()` is `private`, then *only* the `MethodDemoII` class can use `rollDie()`. If `rollDie()` is `protected`, then any other class in the `zNothingToSeeHere` package can use `rollDie()`, and any subclass of `MethodDemoII` that resides in other packages (but don't worry about subclasses yet).

## Binary Search

The binary search is a sweet algorithm that helps search for one item - in a list of many items - in the shortest time possible. Assuming that the items are sorted (like in a dictionary), and assuming that the data structure is a simple one (not a hash table, for instance), nothing is faster than the binary search. The search will always pick the middle item, and determine if the item it is looking for is before that middle item or after it. In doing so, half the items in the list are discarded. This process is repeated until the item is found.

In the example of a dictionary, let's imagine we are looking for the word "sword". We would probably open the dictionary up to the middle, and compare the word we see (perhaps it is "maven"), and determine if we need to examine from "maven" to "zap" or "abracadabra" to "maven". Turns out, we are interested in the "maven" to "zap" words. So, we estimate the page in between where we are and the last page, and open to it. Looks like we are on "sorcerer". Again, we determine we need to look at the pages from "sorcerer" to "zap". So we estimate halfway and turn to that page. The first word is "wand". Looks like we'll have to go to the pages between "sorcerer" and "wand". We go in the middle and end up at "unicorn". So now we know we need to look at the pages from "sorcerer" to "unicorn". Sooner or later, we'll zero in on "sword".

The worst case scenario of guesses in a binary search is log(n). That means, if there are 100 items, we are looking at log(100) which is 7. Here's a sweet diagram of searching for 68:



The best case scenario is one guess - so if the number is 50, and the first guess is 50, BAM! Done!

The worst case scenario is if the number is 2 (or 99, or 97, or any number that results in c). Here are the guesses:

```
50 25 13 7 4 3 2
50 75 87 93 96 98 99
```

**Bytecode**
When Java programs that you write are run, they don't get crunched down into 1s and 0s (whereas in other languages they would go directly from English to binary). The JVM will take the bytecode and crunch it into 1s and 0s for the specific platform the JVM is on. This helps make the experience the same across platforms. The original program is a `.java` file, but when it is crunched down into bytecode, it will be a `.class` file.

**Casting**
Telling the computer to treat one variable as a type that it isn't.

```
    double priceOfGas = 2.29;
    System.out.println(priceOfGas); // 2.29
    System.out.println((int)priceOfGas); // 2
```

This happens sometimes when outputting items using `System.out.print()` as any number that is concatenated to a `String` becomes a `String` automatically.

```
    System.out.println(6 + 3 + " Sum"); // 9 sum
    System.out.println("Sum " + 6 + 3); // Sum 63
    System.out.println("Sum " + (6 + 3)); // Sum 9
```

Variables will be automatically promoted during mixed-mode arithmetic (that is, if not all of the variables are of the same type). Automatic promotion means a restrictive type (like an `int`) will automatically be turned into a more accommodating type (such as a `double`).

```
    System.out.println(40 / 2); // 20
    System.out.println(40 / 2.0); // 20.0
    System.out.println(10 / 4); // 2
    System.out.println(10 / 4.0); // 2.5
    System.out.println(10.0 / 4.0); // 2.5
    System.out.println(10.0 / 4); // 2.5
    System.out.println((double)(10 / 4)); // 2.5
    System.out.println((double)8 / 3); // 2.0
```

**Class**
Often associated with an object, a class is a template for creating a instance. In general, classes need to be *instantiated*, at which point they become *objects*. It's usually easier to give an example of a class and object in real life. The classic example is an Automobile. The Automobile class has properties (such as model year, top speed, number of doors, color, and transmission type). The Automobile class also has a list of things it can do (such as drive forward, turn, go in reverse, and park). An Automobile as a class is just a concept - something we talk about. But once we breathe it life - once we create an instance of it and give it properties, it becomes real. So no one drives a generic automobile around, but people do drive Nissan Xterras around. My first Automobile was a 2001 Nissan Xterra, with a top speed of 105, 4 doors, blue, and standard. THAT is an object - a physical, specific manifestation of the idea of an Automobile. It's an oversimplification to say this, but you can think of a class as a little mini program, because classes typically have their own code.

**Command Line**
Some masochists like to program from the *command line*. This is like the good old days of DOS. Usually it's a black screen with gray, monospaced font. You see things like:

```
    javac HelloWorld.java
```

Unless you have a compelling reason to program from the command line, you are much better off using an IDE.

**Compiler**
High level languages need to be compiled so that they can be "crunched" down to ones and zeros (machine language). A compiler will look at the entire source code of a file and translate it to machine language, and then execute the code. Note that this differs from an *interpreter* because an interpreter will translate one statement at a time and then run it.

**Condition**
Conditions are used in `if` statements, and are the test to decide whether the code in an `if` statement block will execute or not.

**Concatenate**
Joining together (or "gluing" ) multiple `Strings` (usually characterized by a "+" in Java)

```
int age = 38;
System.out.println(“Hello, you are ” + age + “ years old.”);
```

**Constructor**
A *constructor* is a method in a class that specifies how an object of a specific class will be created. All this to say that a constructor is the specifics of how something is “born” in the program. Let’s look at the `Scanner` class. When you “create” an instance of it in your program, you write:

```
Scanner scanner = new Scanner(System.in);
```

This is a constructor! This particular constructor will create a `Scanner` that uses the input stream of the system. There are a few other ways to create a `Scanner`, too (for instance, instead of passing in `System.in`, you could pass in a filename).

Now let’s look at the `Random` class. There are a few different constructors for this object, too. You *usually* would do this:

```
Random random = new Random();
```

This is the default constructor (it has no parameters). This will get you a sweet random number generator. But let’s say you want to do some debugging in your software; it’s hard to see if the program is doing what you think it is doing (the output changes every time because, you know, random numbers). In that case, you can invoke the constructor that takes a parameter. In this example, you pass it a “seed” which is what the random number is based off of. So first you construct the `random` object:

```
Random random = new Random(1234);
```

I chose to use `1234` as an example - you could put any `long` data in there. But if you construct the `Random` object with a seed, all the output will be the same every time the program executes. Note that there are only two different constructors for `Random`, whereas `Scanner` has ten different ones.

**Deprecated**
No longer supported. Code that is deprecated *may* still work, but technically it doesn’t have to. So be careful. It’s a wise move to update existing code that contains deprecated features so that they work with the current rule set in Java.

**EPOCH**
In the computer realm, the *EPOCH* refers to January 1, 1970, at precisely 12:00AM. This is when we - as programmers - have decided we will start time. Most calculations of time are based on how many milliseconds have elapsed since then. You can get the number of milliseconds by using the command:

```
long numOfMillis = System.currentTimeMillis();
```

Note that this is a variable of type `long`, so you'll need to store it in a `long`.

**Errors**
There are three major error types in Java

**Syntax Error**
Analogous to a typo in Word, a *syntax error* will prevent the program from running. In Eclipse, the syntax error is underlined in red and if you hover over it, a hint pops up about what the error is. Examples include missing a semicolon or spelling a word incorrectly.

```
System.out.print1n("Hello World");
```

The error in the above line is in the .println portion - the software engineer used a one ("`1`") instead of a lowercase L ("`l`").

**Run-Time Error**
This occurs because the code is typed right, but as the program runs the computer runs into a situation that makes the program crash. For instance, a program that asks the user to enter two numbers and then divides them, will crash if the user enters `10` and `0` (because  the software attempts to divide by zero). Or perhaps the user is supposed to type in their age and they type in a bunch of letters instead - the software may crash because it can't store a whole bunch of letters in a variable that is of type `int`.

**Logical Error**
You may never find out that you've created a logical error because they don't prevent the program from running and they don't crash the program when it is running. A *logical error* occurs when the computer does exactly what you told it to do, but you told it to do the wrong thing. It's the same thing as giving someone directions to your house, but you tell them to turn left instead of right. Well, they followed the directions it's just that they didn't end up where they should have because they listened to you. This happens often when programming loops because computer scientists start counting at `0` instead of `1`, so you end up with "off-by-one errors" or "fencepost errors"

**Escape Character**
The character(s) that tell the computer, “Hey, listen, I know you *want* to do one thing to the output in quotation marks, but I’m telling you that you need to do something else. Trust me, little computer!”

> \
> Slash
> Whack

```
System.out.println("I'm not \"the norm\" - Chris Farley");
System.out.println(“This will\nbe a new line”);
System.out.println(This is\ta tab”);
```

**Instantiate**
The root word, *instance*, means a single version of something that can be copied. In the real world, there is a theoretical concept of a car. Everyone knows a car has wheels, doors, lights, and a color and can do certain things (move, park, idle). If we were to *instantiate* a car, we would breathe life into it. We would give it very elaborate specifications (the number of wheels, the number of doors, where the lights are, what color it is), and we’d say how fast it can go, how to park it, and what happens when it idles. So this instance of a car could be my 2001 Nissan Xterra. Or maybe it’s the Delorean from Back to the Future. Or maybe it’s a current model year Tesla S. In all of these examples, the abstract concept of a car has been realized with very specific details. Later in Java, the breadth of the car example will make sense. But for now, there are only a few classes we deal with that require instantiation (`Scanner`, for instance). The code to instantiate a `Scanner` requires that we give the instance a name - usually we use “`scanner`”. In this example, let’s call it “`fluffy`” because a `Scanner` is just so cute and adorable.

```
Scanner fluffy = new Scanner(System.in);
```

The only naming requirement is that it adheres to the naming conventions of variables in Java (must start with a lowercase letter, use camelBack writing when necessary, etc.).

**Interpreter**
An interpreter converts statements in the source code to machine code (or virtual machine code) *one at a time*, and then executes it right away. Interpreting translates the statements individually and then executes them one at a time. This is not to be confused with *compiling*, which will crunch down *all* the statements - the source code - and then execute the whole thing.

**Iterate**
A loop is said to iterate when it runs. Every time the loop repeats, it *iterates*. When discussing loops, the convention is for programmers to say something like, “This loop should iterated ten times,” or “In this particular iteration, the value of the variable ‘counter’ doesn’t increase. What gives?”

It is acceptable to refer to one of the particular times a loop runs as an iteration.

**Method**
Think of a method as a separate part of the program that will run when it is called. All methods have a *signature* - that is, an access modifier, a return type, method name, and parameters. Here’s a great example:

```
public double area (int width, int height) {
// code to compute area of a quadrilateral
}
```

Presumably, without examining the code, this method called `area` can run from any class (because it’s `public`), and takes in two parameters (or arguments) - namely, `width` and `height`, which are both of type `int`. It returns a `double` (so the main program better be expecting an answer of type `double`).

```
public double area (int radius) {
// code to compute area of a circle
}
```

This method is also `public`, also named `area`, and also returns a `double`. The difference is that this method takes in only one parameter - `radius`. If both of these methods were in the same class, the main program would not have a problem figuring out which one to use because it would dole out the workload to the method with the proper number of parameters. See overloading.

**Mixed Mode Arithmetic**
This happens when you have a mathematical operation with variables of different type (for instance a `double` and an `int`). In these cases, all the variables are automatically cast as the least restrictive type in the expression. So if two variables are divided - an `int` and a `double` -, then the `int` is automatically promoted to a `double` and the answer is provided as a `double`.

**Object**
An *object* is a manifestation of a *class*. The class specifies that each object that is derived from that class will have specific properties and specific functions. Each object from a class will have it's own, personal properties (so in the Automobile example, the Automobile class specifies that every object of that class (that is, every automobile that drives around) will have it's own model year, top speed, number of doors, color, and transmission type. But every instance of Automobile (every object of Automobile) will be able to drive forward, turn, go in reverse, and park.

**Overloading**
Method *overloading* occurs when a class has two methods with the same name, but the number (or type) of parameters is different. A good example is the `String` class. It has two methods named `substring()`. The computer knows which method to use based on the number of parameters. The `substring` method returns part of a `String`. In the case where there is one argument - an `int` -, the software will return all the letters in the `String`, starting from the number passed in and going to the end. In the other case, when two integers are passed in, the computer will return part of the `String` (including the first number, and all the characters *up to* the second number).

```
String thisString = "This is a String.";
System.out.println(thisString.substring(6);
// s a String.
System.out.println(thisString.substring(6, 12);
// s a St
```

**`private`**
The access modifier of private will limit the scope of the item (class, method, variable) to the class where the item resides.

**`protected`**
The access modifier of protected will limit the scope of the item (class, method, variable) to the package the item is in (and any subclasses of that class in a different package).

**`public`**
The access modifier of public will grant access to the item (class, method, variable) to "the world". This essentially means other classes in the project, regardless of if they are in different packages or not.

**Reserved Words**
There are around fifty words in Java that already have a specific meaning. Don't try to use them as variable names or class names (because terrible things will happen). Examples include:

| | | | |
|---|---|---|---|
| `abstract` | `double` | `int` | `super` |
| `assert` | `else` | `interface` | `switch` |
| `boolean` | `enum` | `long` | `synchronized` |
| `break` | `extends` | `native` | `this` |
| `byte` | `final` | `new` | `throw` |
| `case` | `finally` | `package` | `throws` |
| `catch` | `float` | `private` | `transient` |
| `char` | `for` | `protected` | `try` |
| `class` | `goto` | `public` | `void` |
| `const` | `if` | `return` | `volatile` |
| `continue` | `implements` | `short` | `while` |
| `default` | `import` | `static` | |
| `do` | `instanceof` | `strictfp` | |

**Stream**
A stream is a bunch of information given to the computer at one time. The colloquial expression "streaming music" means that bytes of data are coming in steadily. In Java, a data stream will usually have multiple tokens in it. For instance, this is a stream of integers:

```
45 675 234 65 134 98054 234 5690
```

Generally speaking, a stream uses spaces as a delimiter (that is, the stuff between each of the tokens).

**Token**
When reading in text into a program (for instance, using an instance of Scanner), a token refers to a chunk of the text. By default, the Scanner will consider spaces delimiters, so a token would be anything that has a space before and after it (the notions of tokens is helpful when you want

to read in only part of the input from a user:

```
Scanner scanner = new Scanner(System.in);
System.out.print("Please enter your first and last name: ");
// Fred Rogers
String firstName = scanner.next() // Fred
String lastName = scanner.next() // Rogers
System.out.println("Hello, " + firstName);
System.out.println("Shall I call you Mister " + lastName + "?");
```

**Truncate**
Chopping off anything after the decimal. This will happen automatically if you cast a `double` to an `int`. Note that *truncating* is not the same as rounding! If you truncate `3.9`, the value is `3`, not `4`!

```
double pi = 3.14159;
System.out.println(pi); // 3.14159
System.out.println((int) pi); // 3
```

# APPENDIX D: Answers to Textbook Questions

## CHAPTER 1

### Section 1.1

#1 True: a computer program is an algorithm built specifically for a computer to use.

#2 Typing data and commands in binary (ones and zeroes) is slow and mistakes are more likely. Like assembly languages, it is specific to one type of processor, so every program would need to be rewritten for each type of processor.

#3 Programmers do not need to compile their code for each platform their customers use, but instead rely on the JVM (Java Virtual Machine) to be installed on each customer's computer so it can interpret and execute the program. In other words, programmers can distribute to customers with Windows, Linux, or MacOS by writing and compiling only once. This perceived advantage is where the slogan, "write once, run anywhere" originated.

### Section 1.2

#1 False: An IDE is a convenient program to help programmers perform the tasks of writing, compiling, debugging but it isn't necessary. For instance, any text editor can be used to write code as it is stored in basic text format, javac is a command line program used to compile Java code into bytecode, and there are various methods and programs separate from IDEs that can be used to debug programs.

#2 Colorizing code, linting code to highlight errors and warnings, code collapsing, code completion (also called code assist, or autocomplete), file and project management, keystroke shortcuts, and integration with version control systems like Git.

#3 Many Java programs have code that relies on the code in multiple .java files. A project set up in an IDE makes it easy to manage these multiple-file programs, making it easy to forget about the dependencies, as well. It's best to copy entire projects, keep a workspace that manages all your projects "in the cloud", or use a cloud-based version control system if you need to work on programs from multiple locations.

## Section 1.3

#1 Typing the code, like writing notes, gives a learner an opportunity to process and try to understand what is being typed. Also, practice by typing is the *only* way to establish natural use of patterns that learners are not accustomed to, like semicolons at the end of statements, different rules for capitalization, using braces and square brackets, etc.

#2 comments

#3 token

#4 { and }, also called curly brackets. Braces are used to show the start and end of blocks of code, such as the body of code in the main method.

#5 The **main** method is the most prominent, but don't miss **print**. The **main** method is completely *defined (declared)* in the First class, but **print** is *called* in order to output some information. The print method is defined elsewhere in code that is automatically imported.

## Section 1.4

#1 False. There is enough flexibility within the syntax and even within common conventions for solutions to the same problem to have code that is written in different styles. Furthermore, there are often many different algorithms (specific processes to solve a problem) that can solve the same problem, so the entire approach can be very different, as well.

#2 False. Comments should be used when they help make code more readable and easier to understand. It is possible to have too many comments spreading out the code so it is harder and slower to read. However, learners are encouraged to err on the side of writing too many comments as they come to grips with new code. As expertise increases try to write code that is clear enough with less comments.

#3 Code is indented to show that it is structurally *inside* other code (also called nested). For instance, all of the code between the opening and closing braces of the main method is indented to show that it is inside the main method.

## Section 1.5

#1 An iteration is one cycle through a loop of repetition. If you are pedaling a bike and your right foot presses on the pedal as it's on the top, and it comes all the away around back to the top again, that is one iteration of pedaling. To get to the store on your bike, it will take many iterations of that action. The foot is causing the pedal to iterate through this loop of repetition.

#2 The `println` method inserts a newline character after whatever it is supposed to output, while `print` leaves the insertion point directly after the last character it output. In other words, `println` prints and begins a new line, while `print` just prints.

#3 Write and edit code, compile code, test code. If the program doesn't meet all the requirements, the programmer does it again! Note: another word for edit could be debug.

# CHAPTER 2

## Section 2.1

#1 The three types of comments are:

- **Line comment**: use 2 forward slashes with no space // - tells the compiler to ignore the rest of the line.
- **Block comment**: use a forward slash then an asterisk with no spaces to start a block comment, and end with an asterisk then a forward slash with no space /* … */ - the compile will ignore all text between the start and the end markers.
- **JavaDoc comment** (or Documentation Comment) - like a block comment, except there is an additional asterisk on the start /** … */ -The compiler treats them like a block comment, however, there is an additional application that can generate professional documentation from the comments and tags found within them.

#2 Everything a programmer should know about the program. Minimally, who wrote it, when it was created and/or modified, and what the program does (description of its purpose). Additional information could be added, such as the company, platform it was created on, version of java it was created in, version of the program itself, project specifications, lead designer, etc.

#3 a, b, d ( c. is not valid use of commenting because the variable declaration is itself in the comment and will be ignored by the compiler.)

#4 ANSWERS WILL VARY

| LETTER | STRENGTHS | WEAKNESSES |
|---|---|---|
| a | Clean and Simple<br>Easy to read | A lot of // (line comments) |
| b | Easy to write multiple lines | Does not look as polished as the other methods |
| c | Clean and Simple | Unless IDE sets it up to automatically put the * in each line, and be tedious. |
| d | Sharp and fancy :) | Takes more time to set this up. |
| e | Great for professional API and documentations | Difficult to figure out how to set up correctly. |
| f | Quick and simple | No heading for author, date, and purpose (but is it really needed?) |

#5

```java
//Author: Will McLaughlin
//Date: 6.29.17
//This program will print the alphabet to the standard console output.

public class AlphabetPrinter2 {

  //This is an application, so has a main method.
  public static void main(String[] args) {

    int anUnusedVariable = 0; //an unused variable

    //variable to hold the initial letter to be printed to the console
    //and is initialized to the letter a, the first character to print
    char letter = 'a';

    //loop through all the letters of the alphabet and print
    for(char ch = letter; ch <= 'z'; ++ch){
      System.out.print(ch);
    }
  }
}
```

**Section 2.2**

#1 `int`, `double`, and `String`

#2 A. `int`
B. `String`
C. `double`
D. `int`
E. `double`

#3 `int numClasses;`

#4 A. `int age;`
B. `String petName;`
C. `double total;`
D. `int numStudents;`
E. `double numSiblings;`

#5 Pascal case is also known as UpperCamelCase. It is like camelCase, but starts with an uppercase letter. (camelCase is used for variables, while PascalCase is used for naming classes.)

**Section 2.3**

#1 A. no 'i' in println. Needs to be: `System.out.println("Hello World!");`

B. Missing the addition operator in the argument expression. Should be:

```
System.out.println("Hello " + name );
```

C. Missing the closing double quote to end the string. Should be:

```
System.out.println("Hello \"World\"!");
```

#2 A. Output:

```
Hello
        “World”
!
```

B. Output (note: no space between ‘The’ and ‘quick’):

```
Thequick brown fox jumps over
the lazy
dog!
```

C. Output:

```
Sum of 1,3, and 5 = 9.
```

D. Output:

```
Who would like to see Ted talk?
```

#3 A. A few possible answers:

```
System.out.println("The quick brown fox jumps over\nthe lazy dog!");

System.out.println("The quick brown " +
                   "fox jumps over\n" +
                   "the lazy dog!");
```

B. One possible answer:

```
System.out.println("\"" + ai + "\", sometimes you gotta run before" +
      "you can walk.\"-Tony Stark");
```

**Section 2.4**

#1 Syntax, Runtime, and logic.

#2 A. Syntax error: Missing semicolon

B. Logic error: average of 3 numbers should be divided by 3

C. Syntax error: Cannot set a `double` (real) value to an `int` variable.

D. Syntax error: `println` is misspelled.

E. Logic error: area is equal to the product (`length * width`)

F. Syntax error: You need to add the `String` with the (1+3). i.e. add the '+' operator

```
System.out.println("Sum of 1 and 3 = " + (1+3) );
```

# CHAPTER 3

## Section 3.1

#1 `long`

#2 `char`

#3 `char` using two variables or as you will see: `String`

#4 `float`

#5 `byte`

#6 `boolean` (For example: `boolean workday = false;`)

#7 `short`

#8 `double`

## Section 3.2

#1 A. Both (Initialization statement)
B. Assignment statement
C. Declaration statement
D. Assignment statement
E. Both (Initialization statement)
F. Declaration statement

#2 `first` is 'y'
`second` is 'a'
`last` is 'y'

#3

| | |
|---|---|
| `1number` | - can't start with a number: invalid |
| `char` | - a reserved word for the character data type: invalid |
| `WholeNumber` | - variable starts with a capital letter: valid but against convention |
| `small change` | - space in variable name: invalid |
| `smile^_^` | - ^ is not a valid character to use in an identifier: invalid |

**Section 3.3**

#1 The name variable is declared with the primitive data type `char` which can only store a single symbol, not a `String` literal with multiple symbols like "Colonel Mustard". Replacing `char` with `String` would make this a valid statement that declares and *instantiates* a `String` object identified as `name` with an entire string of characters: `C`, `o`, `l`, `o`, `n`, `e`, `l`, ` `, `M`, `u`, `s`, `t`, `a`, `r`, and `d`!

#2 The single quotes around the `String` should only be used on `char` values written in your code (character literals). For instance, a single symbol `'i'` assigned to a `char` variable uses enclosing single quotes, and a `String` literal such as `"Professor Plum"` uses enclosing double quotes.

**Section 3.4**

No questions here - you're lucky!

**Section 3.5**

#1 Casting is a handy way to take some data and temporarily use it as a different type. This is handy for integer division when you want a decimal answer.

#2 A. The output will be `5`
B. The value of weeks will be `30.416666666666667`
C. The output will be `K` because the ASCII conversion of `75` is `K`

**Section 3.6**

No questions here again - you're super lucky!

**Section 3.7**

#1 There are `indexOf` methods that allow you to send just one argument, the character or `String` you are looking for but always from the start of the `String`, and also `indexOf` methods that allow you to begin your search at an index further inside the `String`.

#2 At this stage in your learning, the two important ways that an object is different from a variable are that the class that defines the object has methods built to act on the data of the object, and that the value of an object is a reference to a memory location where the data is stored. A variable *only stores a single element of data of one type.*

#3 A `char` variable stores a single symbol and is a primitive data type. A `String` object is defined by a class that that can manage many characters in sequence.

#4 The first line of code forces `3` and `4` to be considered `String`s, so the output will be `34`. But in the second line of code, `3` and `4` are added together first before outputting the number `7`.

#5 `row row your boat.`
When the index of "`row`" is found and stored in `pos`, "`Row`" is skipped, not matching "`row`" because of the capital `R`.

#6 `Happy Happy Birthday`
Though the substring method call selects and returns a `String` starting at index 6, it isn't stored in any new `String` or output, so it is wasted.

#7 (E) I, II, and III
The `equals` method checks for the characters in `s3` to be exactly the same as `s1`, which they are, so it is `true`. Using the equals relational operator to compare the `Strings` *also* comes out `true` for a very different reason. When `s2` is assigned `s1` and `s3` is assigned `s2`, the three `String` objects all reference the exact same place in memory.

# CHAPTER 4

## Section 4.1

#1 A. is equal to?
B. not
C. not equal to
D. less than
E. less than or equal to
F. greater than
G. greater than or equal to
H. and
I. or

#2 A. `false`
B. `true`
C. `true`
D. `true`
E. `false`
F. `true`
G. `true`

#3 Short-circuit evaluation is when a computer can stop evaluating a conditional prematurely because there is a foregone conclusion. For instance, when examining an AND statement, if the first component is `false`, there is no reason to examine the second component. This is great for optimization of code!

## Section 4.2

#1 A. NO
B. YES
C. YES
D. NO
E. YES

#2 A standalone `if` block may or may not execute. However, in an `if-else` paradigm, one of the two blocks is guaranteed to be executed.

#3 As an example:

```
public class CoinFlipper {
  public static void main (String[] args) {
    if (Math.random() < .5) {
      System.out.println("Heads");
    } else {
      System.out.println("Tails");
    }
  }
}
```

Note that there is no need to make a second comparison - we know that since there are only two options, if the random number isn't less than .5, it must be greater than or equal to .5 and will be reported as "Tails".

# CHAPTER 5

## Section 5.1

#1 The main difference is that the code in any one `if` statement may never execute; in an `if-else` situation, *exactly one* of the branches will fire.

#2 A. `retire`
B. `over the hill`
C. `lotto` - Trick question! The `if` statement for 24 was not `>=`
D. `run for president`
E. `run for president`
F. `rent a car`
G. no output
H. no output

#3 A. The issue is here:

```
if (num > 15); {
```

There should not be a semi-colon after the conditional. This code segment will still execute, but there will be erroneous results because "Bigger than 15" will always be output.

B. The issue is here:

```
if (choice == "The Rock") {
```

The code should read:

```
if (choice.equals("The Rock")) {
```

Your mileage will vary - some compilers may execute the code without incident, others will operate properly and compare the actual memory addresses, not values, and always return `false`.

C. There are two errors, and they reside on these lines:

```
}; else if (gpa > 2.99) {

}; else {
```

There cannot be a semi-colon after the curly braces - this will cause a syntax error when compiled.

**Section 5.2**

#1 Who lives in a pineapple under the sea?

#2 This is a classic toggle - it will always reverse the state of the `boolean` variable

#3 A. `!a || !b`
B. `!c && !d`
C. `e || !f`
D. `!g && h`
E. `i && j`

**Section 5.3**

#1 Falling through is a situation where there are not `break` statements in every case in a `switch` statement. This means that multiple outcomes can occur. Sometimes this is desirable, though. A `break` statement in every case is like an `if-else` in that only one segment will be executed. A `switch` statement without `break`s is like a number of standalone `if` statements - there could be more than one execution.

**Section 5.4**

#1 Scope is the visibility of a variable. It determines where it can be accessed. So a variable that is created before an `if` statement or a loop can be accessed in the `if` statement or loop. But if it is created inside of an `if` statement or loop, it cannot be used outside the `if` statement or loop.

#2 Compilation error! Although variables can live and die inside of curly braces, in this case we can't create a variable that is already alive!

#3 Nothing remarkable. The first loop will execute and print the numbers 1 through 9, and then the second loop will execute and print the even numbers 2 through 18. Since `i` is local to each loop, there will be no issues!

# CHAPTER 6

## Section 6.1

#1 The main method is the first to be called when a program is run.

#2 Method stubs

#3 True: using verbs in method names causes their names to feel more natural when called.

## Section 6.2

#1 When you copy code, you copy the errors along with it. If you want to update that code in the future, you will have to find it across many areas and possibly files. You may end up with some fixed parts and some that you missed.

#2 If you take the code you were planning to copy and paste into several areas and write it into the declaration of a method, you would only have one central bit of code to maintain over time.

#3 Because `secret` was declared inside the body of the `whisper` method, it has a scope limited to that method. The `main` method never even sees the `secret` variable. In fact, it is only a temporarily used variable while whisper is executing. As soon as `whisper()` is done, `secret` is inaccessible.

## Section 6.3

#1 A return type that is *not* `void` because the call to `getStuff()` is being used to assign a value to `numberOfThings`. The return type must match the data type of `numberOfThings`, so `int` is the best choice. Additionally, the header must have no parameters in the parameter list, and it must have `getStuff` as its identifier (name).

#2 Because it is a method call, the 55 and `userNumber` variable listed in the parentheses are called arguments.

#3 Because it is a method declaration, `count` and `num` are considered parameters in the method's parameter list. Notice that data types are required in a parameter list and not when passing arguments.

### Section 6.4

#1
```
eatVeggies(5);
// any integer value would be fine inside the parentheses
```

#2
```
System.out.println(getPortions(12.2));
// any double value in the parentheses will work
```

#3
```
public static void rotateAndDivide( double a, double b);
// names of parameters can be anything, but data types have to be
// double and double.
```

#4
```
public static int getLiquid(int a, double b);
// names of parameters can be anything, but data types have to be
// int and then double.
```

# CHAPTER 8

### Section 8.1

#1 There are thousands of examples, but some of the more popular examples of arrays include:

- Rosters for a sports team
- Phonebooks
- Inventories in video games
- Cataloging systems in libraries

#2 The diagram below is a sample - you're values in each box may be different, but the structure (name, size) should be the same as in the diagram.

### Section 8.2

#1

```
String[] arrMatey = new String[8];
```

Alternatively, it is permissible to have the code read:

```
String arrMatey[] = new String[8];
```

#2
```
double[] radioStations = {88.5, 91.3, 95.1, 103.5, 105.7};
```

## Section 8.3

#1 The value of `element` is `1`. Don't forget that arrays are "zero-indexed", so the number in the fourth "box" of an array actually has an index of 3!

#2 The answer is `14` because `arr[1]` is `4` and `arr[4]` is `10`.

#3 This one seems tricky, but it's not that bad. The code is:

```
int[] arr = {1, 3, 5, 9, 11};
System.out.println(arr[arr[1]]);
```

So let's look at the call to `arr[1]` first. The value of `arr[1]` is `3`. So the last line of code is really asking us to output `arr[3]`, which is `9`.

#4 Let's look at the code again:

```
int[] arr = {10, 20, 30, 40, 50};
System.out.println(arr[5]);
```

If we were to diagram this, the array would look like this:

This causes an error because in there *is no `arr[5]`!* In the biz, we call this an ArrayIndexOutOfBoundsException. But don't worry - we'll talk about it soon. And you'll most likely encounter this a bunch of times in your coding career!

## Section 8.4

#1 5

#2 4

#3 5

#4 The loop will iterate while `i` is less than or equal to the length of the array, and `i` starts at `0`. This means the loop will iterate one more time than there are elements in the array, and an error will be thrown.