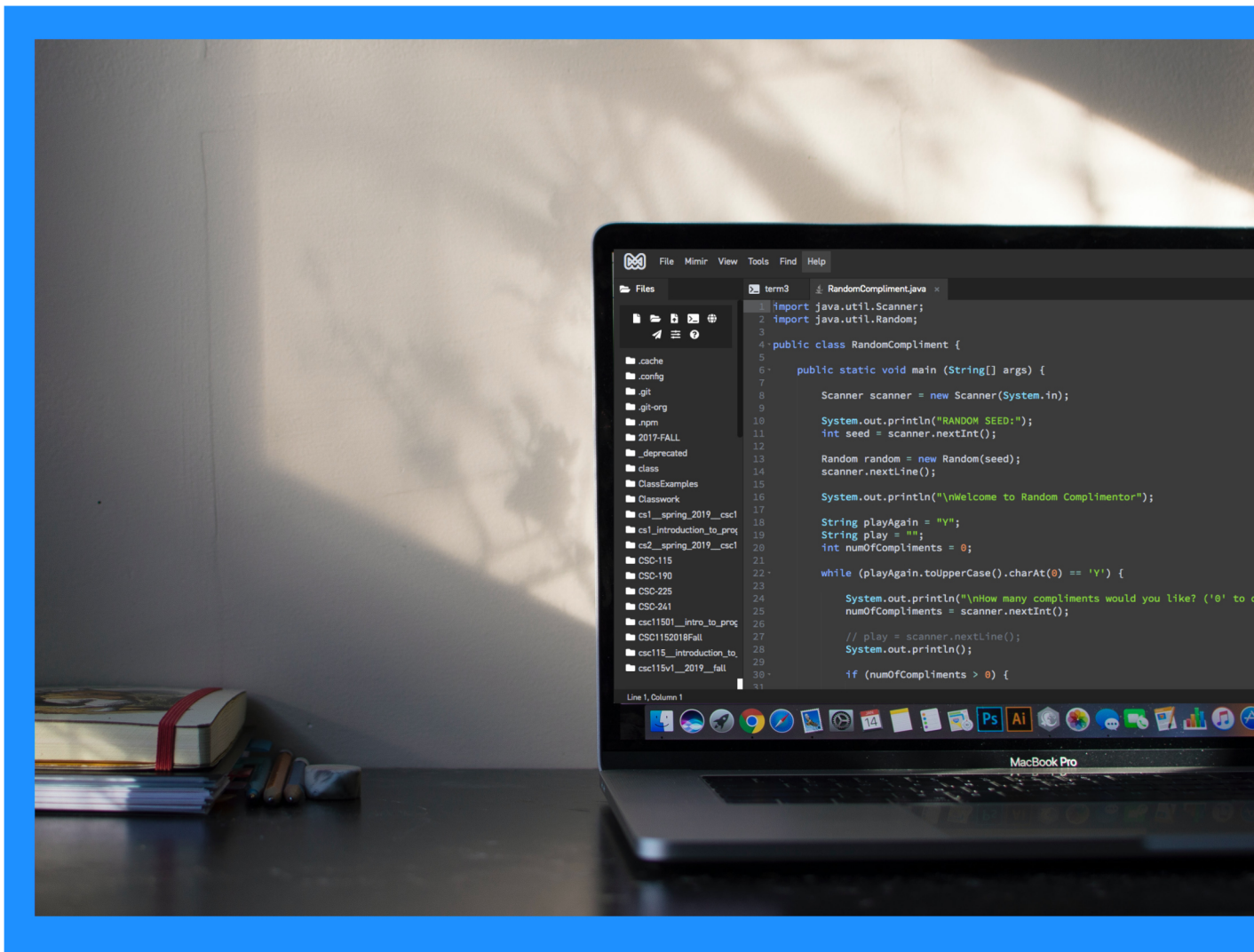


# Java Code: A Modular Introduction

v1.4.0



Java Code: A Modular Introduction

CC-BY

Printed in the United States of America

Unless otherwise noted, all parts of this book are licensed under the Creative Commons Attribution license. You are welcome to use some or all of the contents in this work for any purpose providing you credit the authors.

Dave Ghidiu  
Carrie Krueger  
Will McLaughlin  
Aaron Sullivan

ISBN-12: FLCCCSC115V5  
ISBN-19: FLCCCSC115V5-YOU-ROCK

# Finger Lakes Community College

Carrie Krueger  
Will McLaughlin  
Aaron Sullivan  
Dave Ghidiu

Version 1.4  
FLCCSC115V5

This is the fourth 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](mailto: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, *especially* Carrie Krueger, who added unquantifiable value to this text.

---

This book is also licensed under the [Creative Commons - Attribution 4.0](https://creativecommons.org/licenses/by-nc/4.0/) International (CC-BY-NC-4.0). We recognize that the NC license can be irksome for some institutions. If you are interested in using this in a commercial endeavor, please contact Dave ([dave.ghidiu@flcc.edu](mailto:dave.ghidiu@flcc.edu)).



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

- Many thanks to the students in CSC-115 at Finger Lakes Community College for providing feedback and editing.

#### 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! And, of course, the FLCC Book Nook for exuberantly supporting OER.

*We dedicate our efforts to April, whose leadership and expertise has had a profound impact on our students at FLCC. And not once did she ever say "no" when we had really, really bad ideas.*

# **Java Code:**

## A Modular Introduction

## 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- which is 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*

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.



Public Domain - [https://commons.wikimedia.org/wiki/File:Ada\\_Lovelace.jpg](https://commons.wikimedia.org/wiki/File:Ada_Lovelace.jpg)

Portrait of Ada by British painter Margaret Sarah Carpenter

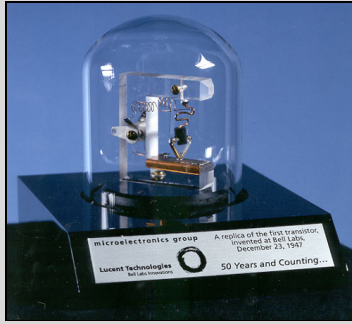
His friend and coworker, Ada Lovelace, envisioned that the analytical engine could potentially do more than mathematical calculations. Among extensive notes she wrote about it, she included what might be the first algorithm for a machine to follow. She may be the very first computer programmer and she 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. The gate 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.

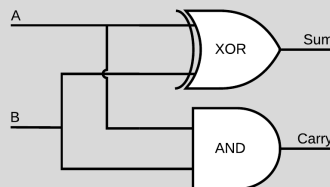
How could simple transistors process numbers? In order to enable computers to perform commands such as adding and comparing numbers, engineers used logical mathematics to arrange transistors. You may have studied Boolean algebra and seen truth tables with true or false outcomes from operators such as 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 (hexadecimal )	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

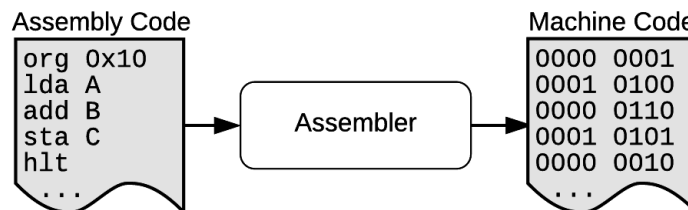
Writing and reading binary digits is perfect for computers but miserable business for most people. We could just slap some human-readable shortcuts over those numbers to represent the operation codes. Maybe then, we could also write a translating program to convert the words back into the equivalent 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**.

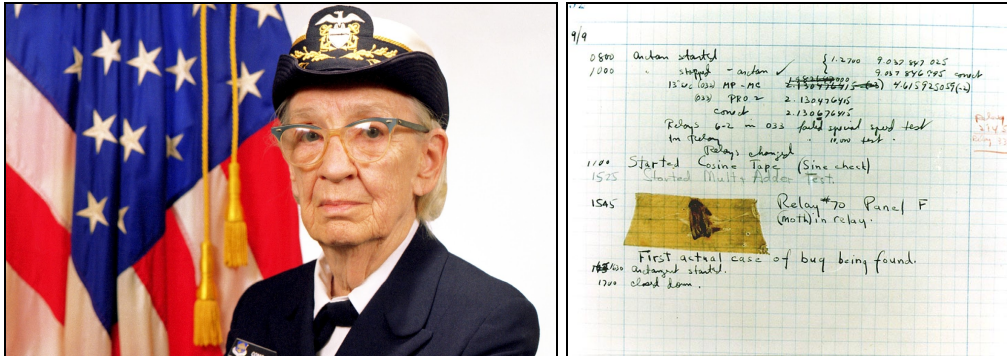
Besides the potential monotony of managing details in larger complex programs, code written in assembly can only work on the computer processor architecture it is targeting. When new advances in processors were introduced, they had different instruction sets. Programs coded in assembly would not work 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! When COVID crippled our computing infrastructure (that still relied on COBOL) because of the need to rapidly deploy relief funds to people filing for unemployment, an organization known as the [Cobol Cowboys](#) were able to [help mitigate the fallout](#).



Commodore Grace M. Hopper, USN (covered). (1984) and the infamous “bug in the machine”

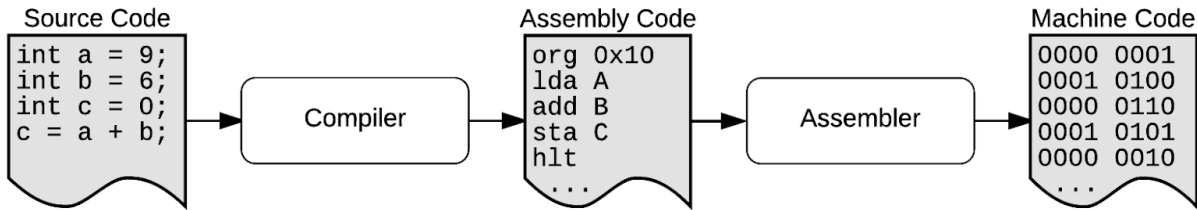
Other high-level languages like C, and eventually Java, added an extra layer of human-readability that brought relief to coders using assembly. Consider the following code that is functionally equivalent to the assembly and machine code listed above:

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

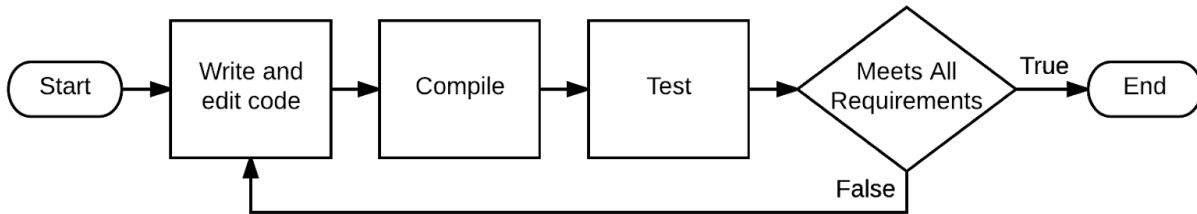
It's short and sweet compared to the others, right? Maybe even partly comprehensible for beginning programmers. Here is the same code with some **comments** added to help explain it.

```
//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
```

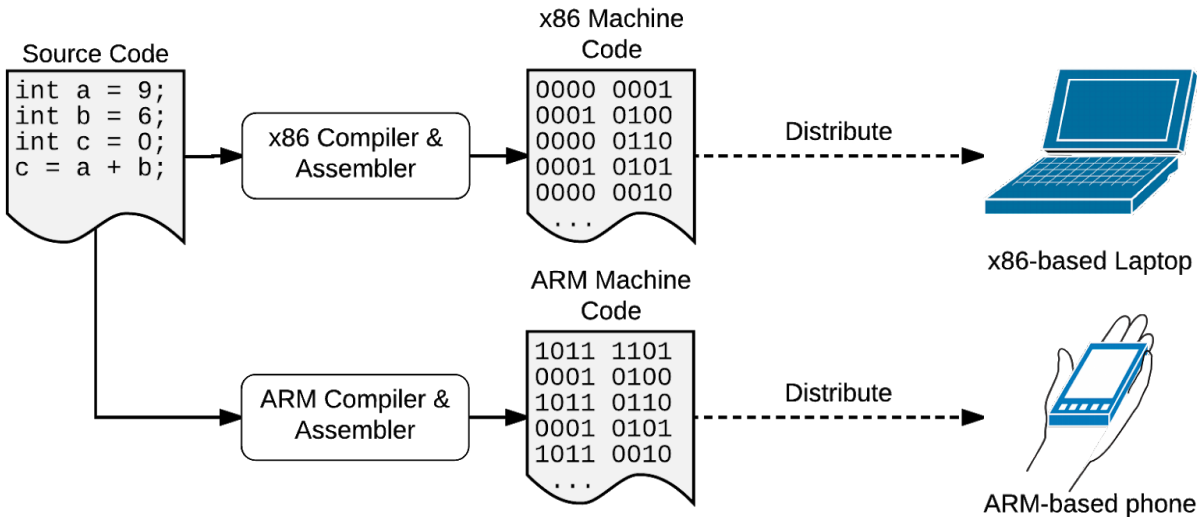
Compare this code example to the assembly and the machine code earlier in this chapter. Which would you rather write? Which would you rather read? There is an extra step needed when using a high-level language like C, because it is no longer so closely related to the hardware architecture. Single line 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.



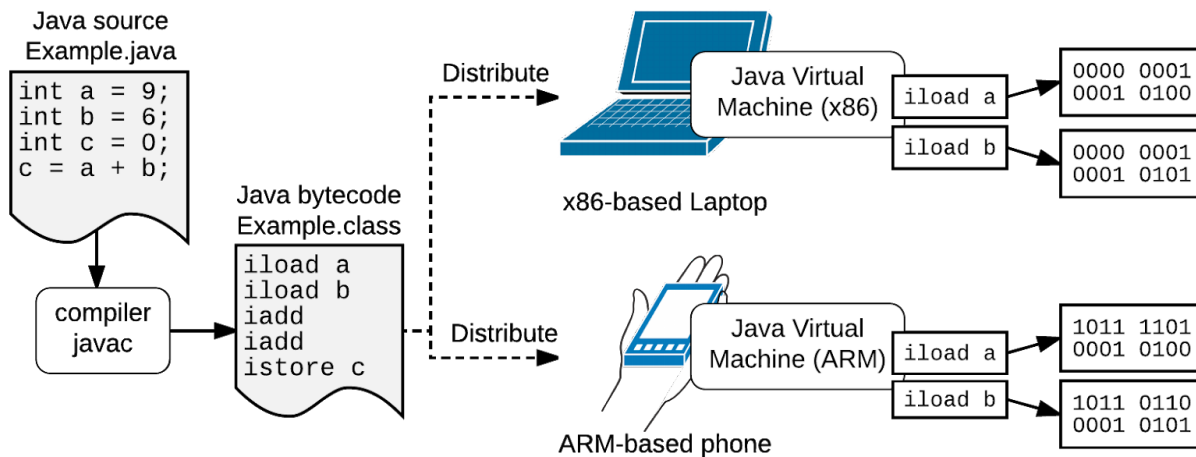
Because a compiler can target different computer architectures, it became possible to write **portable code** using high-level languages. In other words, if a new processor architecture came along, a compiler could be made for it. This way, programmers wouldn't have to rewrite their code. Instead, they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

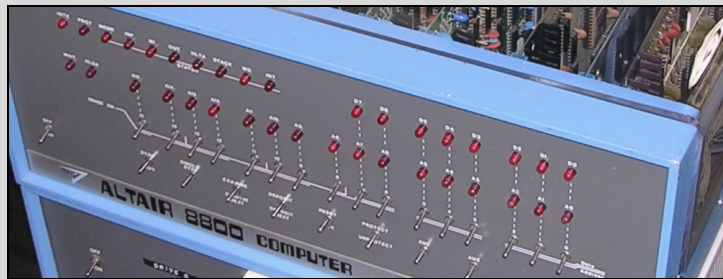
2. Why don't most modern programmers create programs using machine code? In other words, what are the downsides of writing machine code?
3. How does Java's middle step of using javac to compile to bytecode change the way programmers distribute their programs compared to previous high-level languages?

## §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 to 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. Reusable punch cards and other fast methods were developed but were still labor intensive compared to current methods.

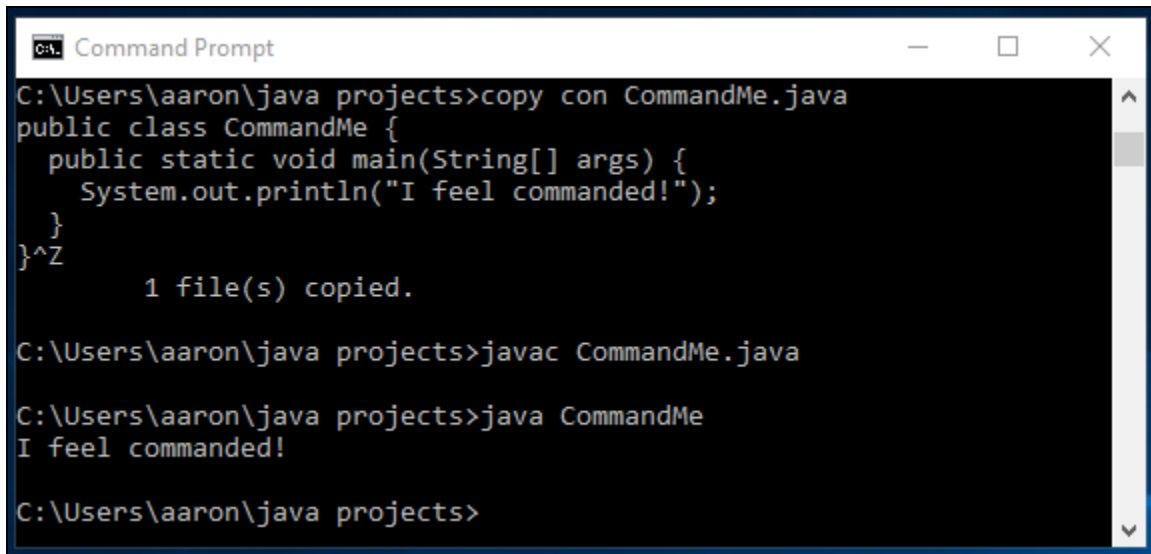


*User interface for the first personal computer, the Altair 8800*

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 using 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\aaaron\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.

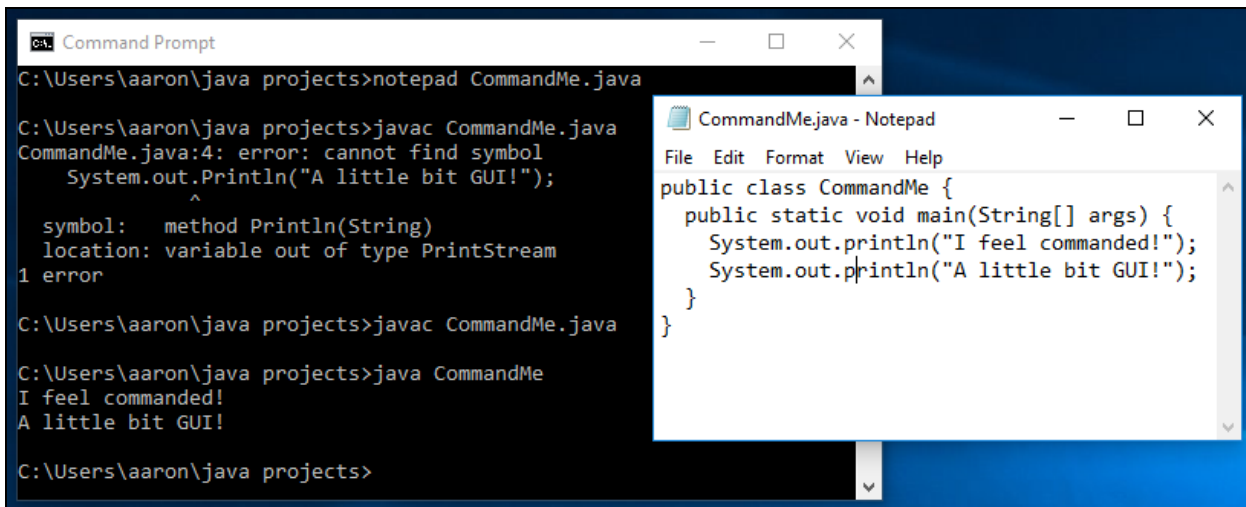
C:\Users\aaaron\java projects>javac CommandMe.java

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

C:\Users\aaaron\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\aaaron\java projects>notepad CommandMe.java
C:\Users\aaaron\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\aaaron\java projects>javac CommandMe.java

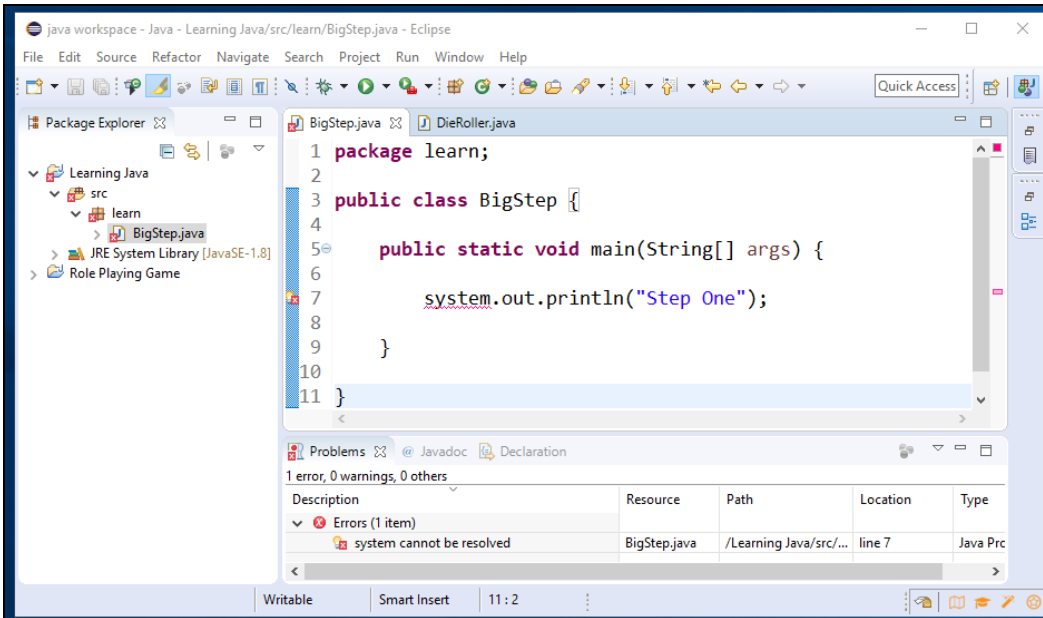
C:\Users\aaaron\java projects>java CommandMe
I feel commanded!
A little bit GUI!

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

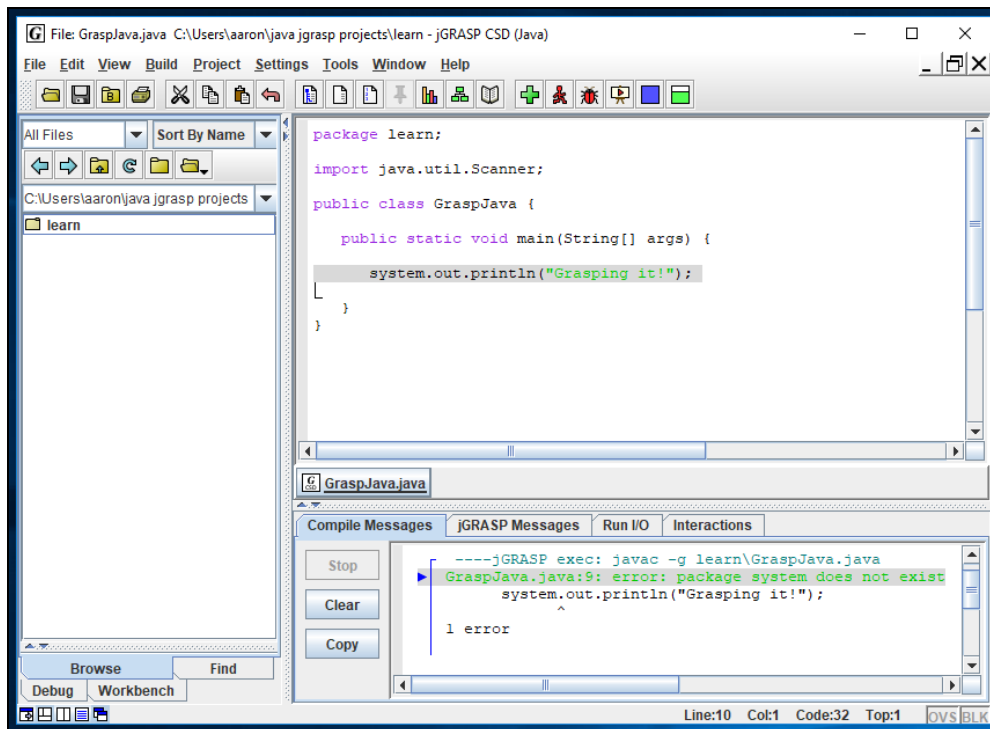
*Command Prompt and Notepad (Can you follow how an error was fixed?)*

## 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, and even suggesting shortcuts to help type parts of code for you!

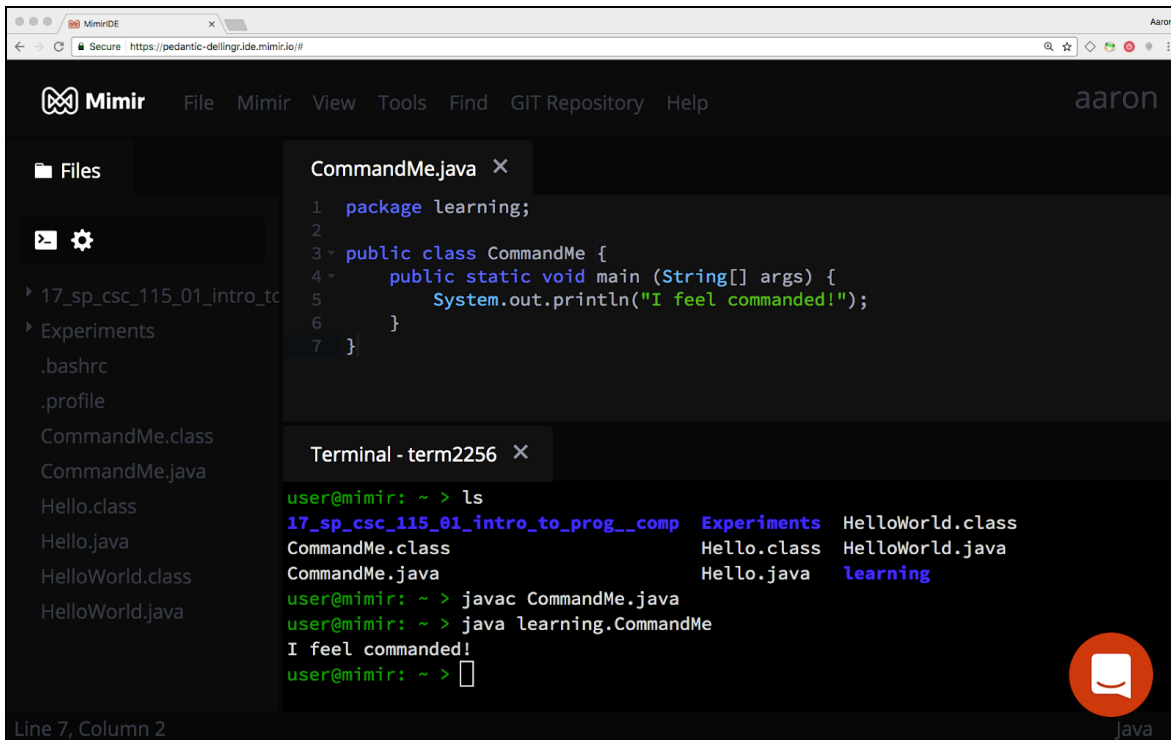


*Eclipse IDE underlining a syntax error with a red, jagged line*



*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 website, a learner can read an assignment, code a solution, fix errors, submit the solution, and even receive feedback.



*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 the IDE 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 (code folding)**.
- indicate syntax errors and other compile-time errors as they occur using a built-in **debugger**.

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 compile from the command line, 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. **Answer true or false:** You need an integrated development environment (IDE) to write, compile, and debug code.
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 the file.)

## 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;
```

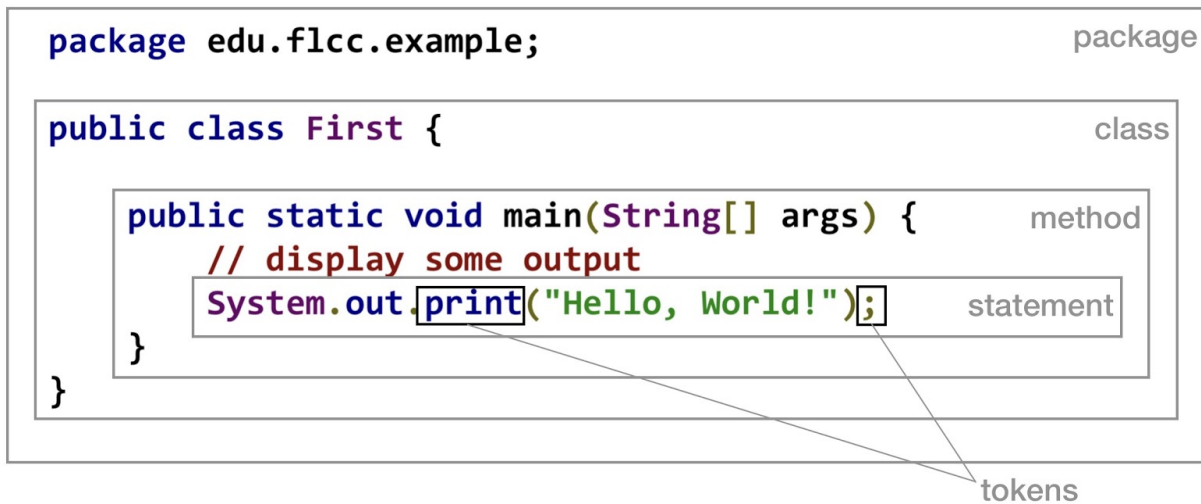
The current steward of the Java language, Oracle, has extensive documentation for when you need further details, including this tutorial about packages:

<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:



Packages are collections of classes.

↑ Classes contain method definitions.

↑ Methods contain statements.

↑ Statements are made of expressions and may also include tokens.

↑ Expressions are 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 each one just by comparing them. This is possible not only because there are many ways to logically find solutions to problems but also because of the flexibility in how code is written.

Look at this code, though:

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

Even early in your learning, that should look terrifying. You might be surprised to know 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 three 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 the 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 to the above sections of related code could also help make the code more readable. When learning, extensive commenting can help you review what you've done before. Professionals use comments too- code should be relatively easy to understand by anyone who is looking at it, whether they authored the code or not. Writing code that is so clear on its own that it does not need extensive commenting to be understood is a good goal to strive for (once you become a more experienced programmer!).

For now, however, adding comments like in the example 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.

```
//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!");
    }
}
```

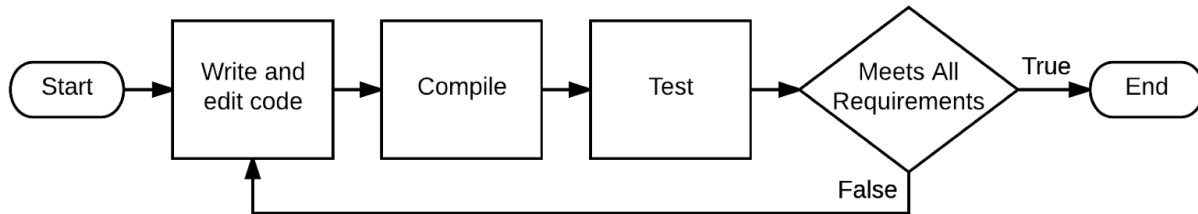
In later chapters, some coding conventions may be highlighted directly, while others will be implied when new concepts are introduced. Either way, be sure to pay attention to coding conventions, so you can help make a complex field more clear to programmers around you.



# §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 is not 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 we want to 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 before reading further. 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.

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");  
    }  
}
```

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.

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");  
    }  
}
```

```
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 & VARIABLES

*"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.*

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 a 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:

```
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. 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, though 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. **Stub code** is code that doesn't really do anything to help the program, but is typically used to output to the screen a message that things are working well.

```
public boolean computePrimeness() {
    /*
    TODO:
    - Design a sweet, sweet algorithm
    - Debug it!
    */

    // Stub code
    System.out.println("computePrimeness has run");
    return false;
}
```

Comments can be inserted wherever there is a need for elucidation - whether that is in the main method of a program or in a different method. Consider the next two examples. The first example does not really need comments because it is a straightforward program. But the second one includes comments that might help someone new to programming understand what is going on:

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

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

        // Example of calling methods from the main program
        printHello(); // Call a different method to do a task

    }

    // This method will output "Hello world!" to the screen!
    public static void printHello() {

        System.out.println("Hello world!");

    }
}
```

Commenting plays a critical role in debugging too, which is discussed later in this chapter.

---

## Check Yourself

---

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

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

3. Identify 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. In the table below, highlight the strengths and weaknesses of each.

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

STRENGTHS	WEAKNESSES

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

STRENGTHS	WEAKNESSES

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

STRENGTHS	WEAKNESSES

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

STRENGTHS	WEAKNESSES

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

STRENGTHS	WEAKNESSES

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

STRENGTHS	WEAKNESSES

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.)

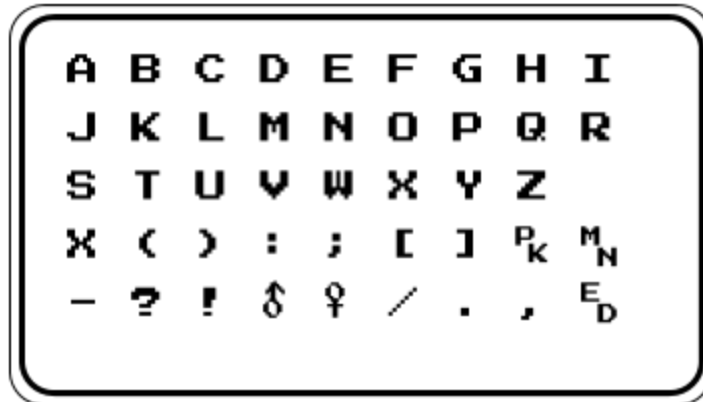
```
/*
 * Author: Will McLaughlin
 * Date: 6.29.17
 *
 * This program will print the alphabet to the standard console output.
 */

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
         * that 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 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:

YOUR NAME? JOHNCENA



lower case

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, Random, 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 **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 of these data types will be used in this chapter, or even in the text. However, for completeness, all eight data types are listed here. Note that you do not need to know the size (bits and bytes) for each type.



Primitive Data Type	Contains	Default value	Size (Bytes)	Size (Bits)	Range
boolean	true or false	false	NA	1	true or false
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 9,223,372,036,854,775,807 ( $-2^{63}$ to $2^{63}$ )
float	Floating point	0.0	4	32	+1.4E-45 to +3.4028235E+38 (around 7 significant digits)
double	Floating point	0.0	8	64	+4.9E-324 to +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. `ints` can store values roughly between negative 2 billion and positive 2 billion. For instance, the following are all valid values of `int`. An `int` 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 not 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` (1 byte) is like a puny `short`. 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 (especially if you know what a teraflop is from benchmarking hardware). 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).

One last thing - every char has an integer representation. So the letter 'A' is really the number 65, 'B' is 66, and 'd' is 100.

'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. That is roughly 321 million different ages that need 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 most of your programs, use the int data type for integers, and the double data type for real numbers.

### Strongly typed languages?

Java, like many languages, is 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 you are trying to assign x is not a whole number, so it 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**.

In contrast, a language 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.

## The String Class

But wait! So far we have ways to represent whole numbers, decimals, and characters. But previously in this book we've talked about text (a String). So why isn't that in the section for primitive data types?

It turns out that a String (think of it as text) is not a primitive data type. It's a **class**. Java thrives on the notion of classes, and we'll be talking about them sporadically through this book. You'll find out in later Java courses that classes are a really big deal. Chapter 4 is devoted to the String class and a few other classes we will be using in this course.

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");
```

---

## 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

## §2.3 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 of wizards may store their potions. You only need them at certain times, so you store them in bottles with labels on them. **Variables** are the bottles with labels. Data is stored in bottles (**memory**), and is labeled (variable names) for access later on.

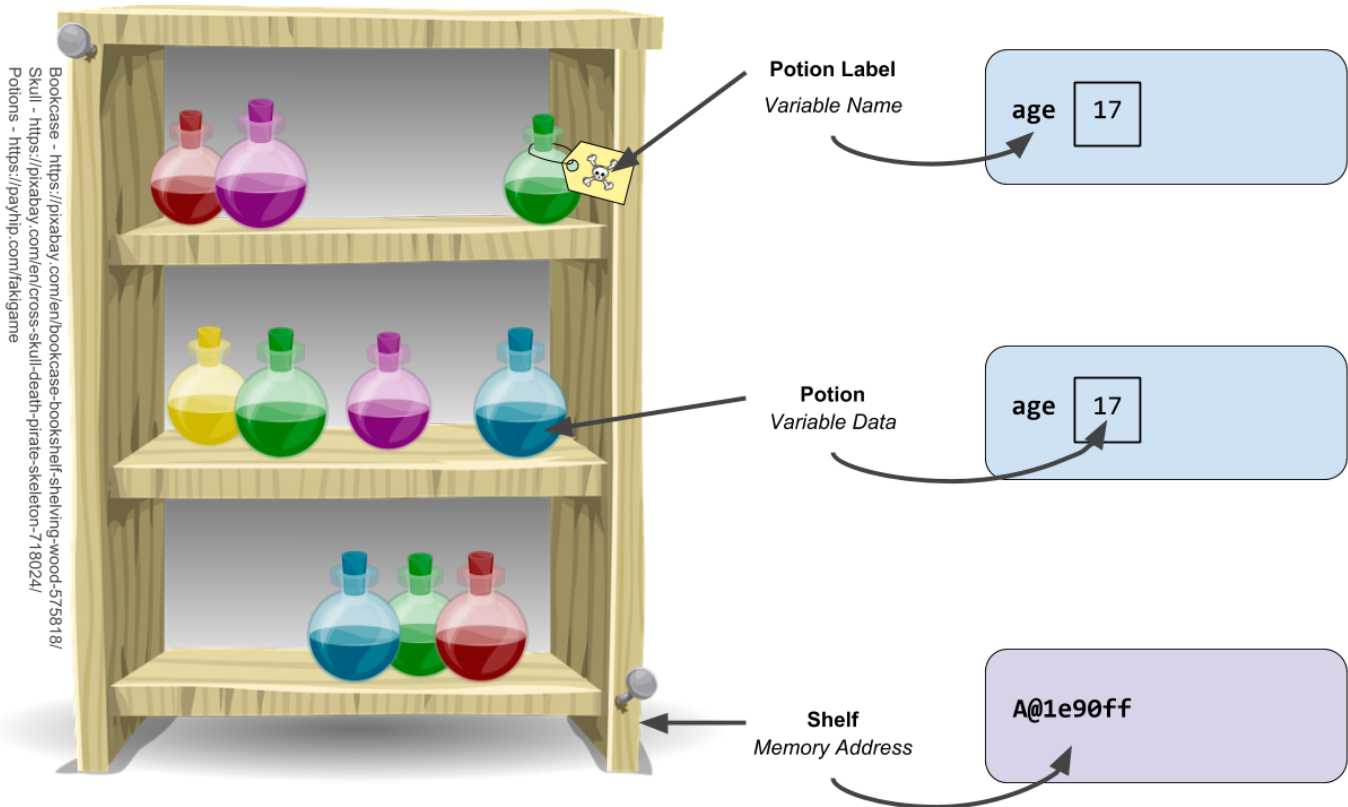


Figure 2.2a: Examples of data storage and terminology

### Declaring a Variable for Use

To **declare** a variable is to tell 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 data of that type is stored in that memory location. The name of the variable is used to associate the spot in memory where the value is. This is like the potions mentioned earlier in the book. A variable that has been declared but not yet assigned a value would be like a potion jar with no potion in it - there is room for the potion to go once it is mixed.

The item (data type) being stored dictates how large the bottle (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 declare a number of variables at once (assuming they are all of the same type):

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

## Naming Variables

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

1. Start with a letter, dollar sign, or underscore (a, b, c, ..., z, A, B, C, ..., Z, \$, \_).
2. Valid variable names can contain only letters, numbers, the dollar sign (\$), and an underscore(\_).
3. Variable names 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_but_not_preferred  
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 code 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.

```
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, which 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 assigned
// 'weight' has been declared and has been assigned a value of 160
// 'height' has been declared, but not assigned
```

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

```
int age = 17;
weight = 160; // BAD THINGS WILL HAPPEN
```

You can even reassign them multiple times in a program - it's like recycling! [@CaptainPlanet](#). To update the variable in memory, simply reassign the variable:

```
int age = 17;
age = age + 1; // statement to add one to age, and update the variable
```

There are a couple of concepts to point out from this example. On the right hand side (this is called the **RHS** in the biz, and you can probably deduce what **LHS** stands for) of the assignment operator, you have an expression. The expression is evaluated before the assignment happens. To evaluate it, it needs the current state of the variable `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.



## Keep It Consistent ...

Like many other programming languages, there are options and some flexibility in programming styles to complete tasks in Java. Declaring 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.

## camelCase 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. Some lend themselves better to SNAKE\_CASE, while others to camelCase. Java uses the camelCase naming convention for variables and SNAKE\_CASE for constants.

---

## Check Yourself

---

1. What are three of the predominant data types that Java programmers use?
  
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 of a lunch bill
  - d. Number of students in a college
  - e. Average number of siblings for college students
  
3. Label each line as 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;`
  
4. Declare a variable to store the number of classes a student is taking.
  
  
5. Declare variables to store data for each of the following:

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

6. 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;
```

7. What is wrong with the following code?

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

8. What is wrong with the following code?

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

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

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

## §2.4 Basic Output

A great program has an intuitive and smooth user interface. User interface design is an 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 parentheses (...). The argument is typically text, a 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 of 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
```

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

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

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

System.out.println(total + total * 0.08);
// The variable 'total' has a value of 44.99
// Display total with 8% tax added to it

System.out.println("Greetings " + name + ", ");
// Example that combines Strings
```

```
25
48.59
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 when 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:

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

```
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 forward to the next line. The method `print` does just that. The following would be much better:

```
String user = "Bill Gates";  
  
//print greeting to user  
System.out.print("Hello " );  
System.out.print(user);  
System.out.print("!");
```

```
Hello Bill Gates!
```

## Escape Sequences

So the `print` and `println` functions will send the value of the parameters output to the standard console, 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 will 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 - called an **escape character** - to flag the compiler that the next character is special. The character chosen to flag the compiler is the backslash (`\`). This escape character 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 it needs to be used within a String using double quotes. Common escape sequences are:

- `\"` 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.

## Output with Escape Sequences

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

```
/* 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");
    }
}
```

```
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 use only addition, so all computations are computed left to right, unless there is a set of parentheses. The expressions within the parentheses 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 outputting the value.

When adding String and numbers together, it gets 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 = 3" + 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 operations, 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");`

## §2.5 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.

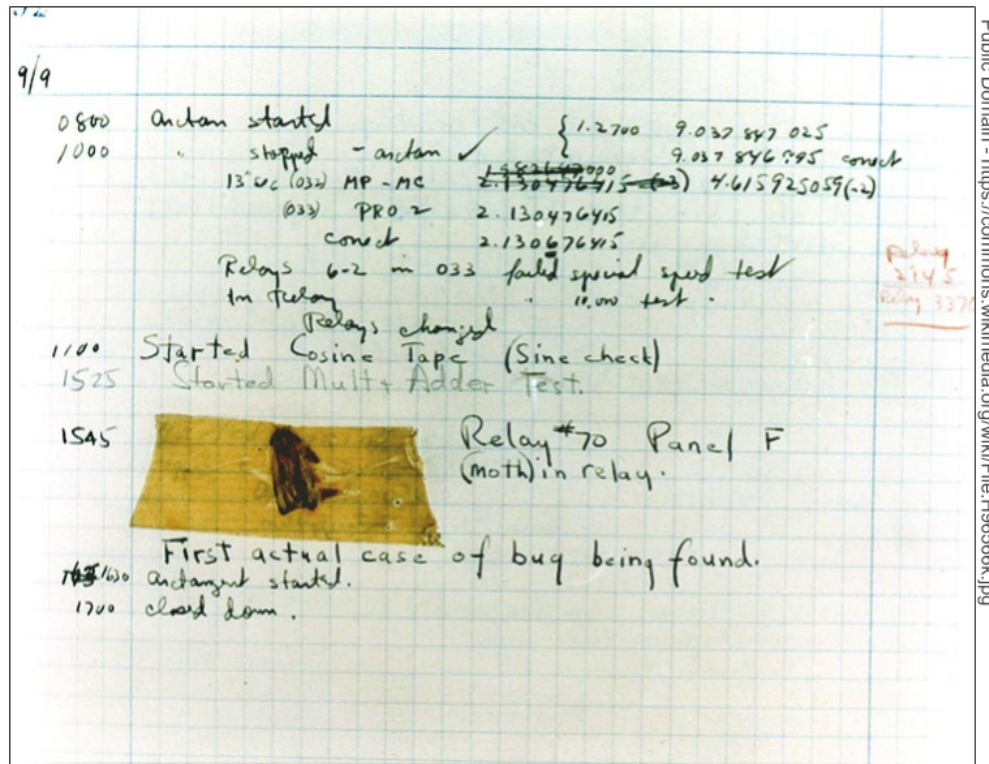


Figure 2.4a: First documented computer bug, Grace Hopper - September 9th, 1947

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, many editors will underline the error in red, much like a spell checker in word processors (Mimir and other command line editors do not have this feature, so you won't find out about syntax errors until you try to compile). Syntax errors will prevent the software from running. Usually, it's an easy fix (for instance, a semicolon has been omitted), although every now and then the syntax errors can be harder to understand. The following code has a syntax error:

```
System.out.println("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 report errors on a line that is actually caused by an 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 a while. 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 its tracks. In some editors (such as Eclipse) you’ll get the “Red Text of Death.” In Mimir, you’ll see an error message in the terminal.

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 really 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 the next program, the computer asks the user for a number. In the first input, 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 they 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. They won't stop the

program from running, and they often won't crash the program (which is why it's easy to miss them - you may never see any evidence they exist!). At least runtime errors and syntax errors have the decency to make you aware of their presence.

Logic errors happen because the computer does exactly what you told it to do- it's just that you 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. However, 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 num1 = 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 (if you don't, that's okay)?

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!).

## Famous Errors

Check out [Appendix D](#) for a list of famous errors that you will encounter along your programming journey. Come back and reference it often!

## Debugging

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 one more line (while keeping the other eighteen lines commented out). Run it. If no error occurs, uncomment another line. By introducing code one line at a time, it is possible to discover the error methodically.

Let's imagine that I want to write a program to create three 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.

```
public class RandomNumbers {
    public static void main(String[] args){
        randomNumber();
        randomNumber();
        randomNumber();
    }

    public static void randomNumber() {
        int randomNumber = (int)Math.random();
        System.out.println(randomNumber);
    }
}
```

```
0
0
0
```

*Well that's weird... What are the chances that all three random numbers are zero? Plus zero shouldn't even be an option for the random number. It's very likely I screwed up somewhere. Maybe I used `Math.random()` improperly? Maybe I'm not using methods the right way? I'm not sure. So I'm going to do two things. First, I'll comment almost everything out and see if it runs. I'm also going to add stub code to the program. This will make it easy to see if I'm on the right track- if the method is called, I'll be able to see evidence of it in the **console**. If the program works here, I'll try adding some of the commented lines back into the program...*

```
public class RandomNumbers {
    public static void main(String[] args){
        randomNumber();
        randomNumber();
        randomNumber();
    }

    public static void randomNumber() {
        // int randomNumber = (int)Math.random();
        // System.out.println(randomNumber);
        System.out.println("This is a test of the method");
    }
}
```

```
This is a test of the method
This is a test of the method
This is a test of the method
```

*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, and I see that the output came out as expected. 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. At this point, I'm confident that I'm calling the method right, so the issue is either in how I create the random number or how I am outputting it...:*

```
public class RandomNumbers {
    public static void main(String[] args){
        randomNumber();
        randomNumber();
        randomNumber();
    }

    public static void randomNumber() {
        int randomNumber = (int)Math.random();
        System.out.println("The random number is: " + randomNumber);
        System.out.println("This is a test of the method");
    }
}
```

```
The random number is: 0
This is a test of the method
The random number is: 0
This is a test of the method
The random number is: 0
This is a test of the method
```

*Hmmm.... Two things. First, I don't need that second output ("This is a test of the method") any more, so I'll be sure to comment it out before I run it the next time. The second thing is that 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.*

```
The random number is: 0
This is a test of the method
The random number is: 0
This is a test of the method
The random number is: 0
This is a test of the method
```

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

```
The random number is: 0
This is a test of the method
The random number is: 0
This is a test of the method
The random number is: 0
This is a test of the method
```

*I'm calling it! It looks like the error is still there, and it's probably 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.*

```
public class RandomNumbers {
    public static void main(String[] args){
        randomNumber();
        randomNumber();
        randomNumber();
    }

    public static void randomNumber() {
        int randomNumber = (int)(Math.random()*100);
        System.out.println("The random number is: " + randomNumber);
        // System.out.println("This is a test of the method");
    }
}
```

*The error, by the way, is that `Math.random()` generates a number between `0.0` and `0.9999999999` - don't worry, we'll cover that in a later chapter. I should have multiplied it by `100` before I turned it into an integer. Easy mistake, I do it all the time. You will too! There's actually another logic error here too- the way this is written, the numbers that will be generated are `0` through `99`, not `1` through `100` as hoped. That's also an easy fix for that that we'll talk about later.*



---

## Check Yourself

---

1. What are the three classifications of errors generated in Java?
  
2. Identify the kind of error that is generated by each line of code:  
(If needed, program the statements as they are written below, and then compile and/or run the program to see the errors generated.)
  - a. `System.out.println("Hello World!")`
  
  - b. `double average = (82+90+87) / 4;`
  
  - c. `int age = 18.5;`
  
  - d. `System.out.println("Hello Again!");`
  
  - e. `int area = length + width;`
  
  - f. `System.out.println("Sum of 1 and 3 = " (1+3) );`

## MATH

*"Do not worry about your difficulties in Mathematics.  
I can assure you mine are still greater."*

- Albert Einstein -

After covering the first two chapters, you should now know how to write a simple Java application, declare variables, and perform basic mathematical operations. This chapter will get deeper into how Java calculates expressions, and look at other operators and tools available to Java programmers.

You will see that numerical calculations are separated into two different categories: *Integral Operations*, and *Floating-Point Operations*. This is because they are represented, stored, and written in memory in two completely different formats. Integral values are stored as binary numbers, or simply a series of 1's and 0's. Floating-point values are stored as a product of two parts: a mantissa and a power on a base of two. This concept is above the scope of what is needed here, but feel free to research on your own. The point is that there are two completely different kinds of numbers in computers and in turn, programming. Integers and Floating Point values are each handled differently.

## §3.1 Integral Operations

### Integral Operations

Integral operations are operations that work on integer values. The primitive data types in Java that we will work with that are stored as integer values are short, int, long, and char.

All operations are done on values, not variables!

You can think of the following example not as multiplying the variables, but the values of the variables.

```
int length = 4;
int width = 7;

//calculate the area
// does NOT multiply 'length' times 'width'...
int area = length * width; //multiplies 4 times 7
```

This is semantics, but it's helpful to understand this early. You will see many examples in future programming where this concept applies (using values and not variables). The following example will perhaps make it clearer as to why this concept is important. Recall from Chapter 2 (primitive data types) that characters are represented as an integer. The character 'B' is stored in memory as the value 66. (Note: there is more on adding characters later in the chapter.)

```
int offset = 5;
char ch = 'B'; //ordinal value of 'B' is 66

int cipher = ch + offset; //Adds the integral values 66 and 5
System.out.println(cipher); //will print the value 71
```

You know the common mathematical operators of addition, subtraction, multiplication, and division. The first three of the four work as you expect. Division, however, is a little different. You will likely understand the following examples without much effort, as they are written in the high-level programming language of Java. This code is intuitive, and does not use division.

```
int perimeter = 2*length + 2*width; //order of operations applies

int group1Size = 4;
int group2Size = 3;
```

```

int group3Size = 5;
int totalGroupNumber = group1Size + group2Size + group3Size;
// Sum of 4,3,and 5 is 12. The value of 12 is assigned to totalGroupNumber.

totalGroupNumber = totalGroupNumber - 2;
//remove two people from the group
//12-2 is 10, and 10 is the value assigned to totalGroupNumber.

```

It is worth noting here that the right-hand side (RHS) is evaluated first, and then assigned to the variable on the left-hand side (LHS). Also note that the values are substituted in for the variables before calculating the expression using the standard order of operations.

## Division

Division in Java is not what most early programmers expect. For example, what is  $5 / 2$ ? Most intelligent people would say 2.5, or  $2 \frac{1}{2}$ . However, this is incorrect! The answer is 2. Why is that? As noted earlier, integral values are represented as a binary number. These are integer numbers only! An integer is a positive or negative whole number- no decimals or fractions are allowed. When dealing with whole numbers in Java, you will only get whole number answers, and no decimals. So, the decimal, or fraction, part of a quotient (division problem) is never calculated. The following expressions should give you an idea of how division is calculated in Java:

Expression	Calculation	Expression	Calculation
$4 / 2$	2	$25 / 10$	2
$5 / 2$	2	$9999 / 10000$	0
$-6 / 2$	-3	$10001 / 10000$	1
$-7 / 2$	-3	$1234 / 10$	123

## Modulus

Modular arithmetic is an area of mathematics that revolves around a set of whole numbers created from division. More specifically: remainders. An initial read of modular arithmetic is daunting for many, however, the concept should not be too foreign. This system of mathematics for integers has many applications in number theory and other areas. The focus here will be to gain a simple understanding of the modulus operator. In Java the percent symbol (%) is used for this operator.

This may be the first time you have read about the modulo operator, abbreviated as mod, but you have done this mathematics before. Some have called this clock math, or remainder math, because this speaks to the heart of what modular arithmetic really is. On a clock, there are only 12 hours. Even though many hours go by, the largest hour value will always be 12. Let us take a look at several clock examples to understand the concept.

If a clock reads 12:00, and 15 hours go by, it will read 3:00. This is calculated by removing all of the 12 hour blocks and using the remainder of 3 to find the new time. We say 15 modulo 12 is 3, or  $15 \text{ mod } 12$  is 3. In Java we would see  $15 \% 12$  (with the result being 3).

If we started at 12:00, and 28 hours go by, it would then be 4:00. Note that 28 divided by 12 is 2, with a remainder of 4. Therefore,  $28 \bmod 12$  is 4.

One more clock example: start at 12:00, and have 58 hours go by. It would then be 10:00. Note that 58 divided by 12 is 4, with a remainder of 10. Therefore,  $58 \bmod 12$  is 10.

This remainder calculation has many uses, and you have likely used in your everyday life without even realizing it! For example, if you have ever set up teams for a game, you have done modular arithmetic. Say you want to set up teams for basketball, or any game with teams of 5. You want to know how many players will not be on a team and can be a sub. If you have 14 players, you will have 2 teams with 4 players not on a team ( $14 \bmod 5$  is 4). If you have 23 players, 3 will not be on a team ( $23 \bmod 5$  is 3). If you have 35 players, there will be no subs ( $35 \bmod 5$  is 0). Modulus goes hand-in-hand with integer division. In the three previous examples, we could use division to calculate the number of teams that could be formed. The following table may help clarify:

Number of players	Division expression	Number of teams (answer to integer division)	Modulus expression	Number of players not on a team (answer to modulus expression)
14	$14 / 5$	2	$14 \% 5$	4
23	$23 / 5$	4	$23 \% 5$	3
35	$35 / 5$	7	$35 \% 5$	0
4	$4 / 5$	0	$4 \% 5$	4
72	$72 / 5$	14	$72 \% 5$	2

Hopefully you can see why this is sometimes referred to as remainder math, as the answer to a modulus expression is the remainder. So, if the modulus is 5, as in the examples above, the smallest answer would be 0, and the largest answer would be 4. Clearly, the smallest number of players that can be a sub is 0, and the largest is 4. If there were more than 4 players not on a team, you would simply create another team.

## Overflow Error

If you put a gallon of water in an eight ounce glass, you will have a mess as all the water will overflow. Just like that, you can create an integral overflow error in Java. Recall that the `int` primitive data type has a maximum value of 2,147,483,647. If you try to add another value to it, the computer will do it, but the results will be inaccurate. The overflow error is a runtime error, meaning that the compiler will not prevent the program from running but the output will be inaccurate.

```
System.out.println( 1111 * 1111); //1234321
System.out.println( 11111 * 11111); //123454321
System.out.println( 111111 * 111111); //-539247567 - Overflow error
```

---

## Check Yourself

---

1. What would the value of the answer variable be after each of the following calculations?
  - a. `int answer = 5 + 2 / 4 ;`
  - b. `int answer = 12 % 8;`
  - c. `int answer = 3 + 12 % 10;`
  - d. `int answer = (3 + 12) % 10;`
  - e. `int answer = 3 + 2 * -4 + 1;`
  - f. `int answer = (3 + 2) * (-4 + 1);`
  
2. There are 23 members of a marching band. The director wants to put them in rows of 6.
  - a. Explain how integer division and modulus can be used to calculate the number of rows that will be in the band, and also to find how many band members will not be in a complete row.
  - b. Write the expression to calculate the number of complete rows formed.
  - c. Write the expression to calculate the number of band members that are not in a complete row.
  - d. What would the expressions be if there were 44 band members instead of 23?
  - e. If there was an `int` variable used, named `numBandMembers`, to represent the number of band members, what would the expressions be using the `numBandMembers` variable?
  - f. Write the expressions using the `numBandMembers` variable and a `numOfPlayersPerRow` variable, where `numOfPlayersPerRow` is an `int` variable representing the number of band members per row.
  
3. Think about or find two other examples where the modulus and division operators can be used to find the solutions to a problem.

## §3.2 Floating Point Operations

Now that you have read about integral operations, floating-point operations will be brief. When the literal values are decimals and/or the data type variables used are doubles (or floats), floating point calculations are used. In other words, when dealing with decimals, you will have decimal answers. The same operations are available: addition, subtraction, multiplication, division, and modulus. The final value will be a **double** data type. Modulus arithmetic can be done with floating point values, but it is typically only used with integral values.

### Round-Off Error

As mentioned earlier, floating point values are not stored like integer values. Computers do well with base 2 numbers, but not as well with base 10. There are values that are difficult for computers to store the exact value for using primitive data types. You likely can add 0.7 and 0.1 in your head, but with the way a computer works, the calculation becomes difficult. The computer can come up with a really close approximation, but it will be unable to keep track of the exact value.

```
System.out.println( 0.7 + 0.1); //output: 0.7999999999999999
```

So, the take-away: **whenever you use a floating point value in your calculations, assume the value is not accurate.** This may sound silly, but it will help you to assume that the answer calculated is never 100% correct. Realize that the calculated values will be super close, but may not be the exact (correct) value. This type of error is a runtime error and will not be detected by the compiler.

### Loss Of Precision Error

Primitive data types that work with real numbers (floating-point values) do well with keeping track of really big numbers, like the mass of the earth ( $5.9722 \text{kg} \times 10^{24}$ ). They are also good at holding really precise fractional values, like the approximate value of pi (3.141592653589793). Due to the way floating point values are stored, they can only hold a limited number of significant values. So when working with relatively large numbers and very accurate values, you may lose some precision when adding. For example, if you add the value of pi to the mass of the earth, the value will be unchanged. This type of error is also a runtime error and will not be detected by the compiler.

---

## Check Yourself

---

1. Why should you always assume that all real number calculations in Java are only close approximations?



## §3.3 Mixed-Mode

We now know that computers handle integral calculations differently than floating-point calculations. What happens when we use different data types in the same calculation? For example, we know  $5 / 2$  is 2 and  $5.0 / 2.0$  is 2.5. But what is  $5 / 2.0$ ?

Java allows for mixed-mode calculations, and will seamlessly change everything to floating-point values during the process of calculating. Whenever an operation has a floating-point value in it, a floating-point calculation is made. So in the example of  $5 / 2.0$ , the computer is going to use floating-point calculations. To do this, the integral value of 5 is changed to the decimal value 5.0, and the expression is then evaluated to 2.5.

Keep in mind that the calculations are done with only two values at a time, so each calculation is handled separately. For example, the following code calculation does not work as you may expect. The end result will be a `double`, but an integer calculation is used.

```
int x = 7;
double y = x / 2; //both are integral values so an integral calculation is used
System.out.println(y); //output: 3.0
```

The following example should also support the concept that each calculation is done separately, with either an integral calculation or a floating-point calculation. For example,  $5 / 2 * 4.0$  is  $8.0$ . The calculation is done as follows (note that order of operations is discussed in section 3.6):

$5 / 2 * 4.0$	order of operations applies; first calculation is $5 / 2$ - an integral calculation
$2 * 4.0$	mix-mode operation, so need to deal with real numbers only (convert)
$2.0 * 4.0$	floating-point calculation is made
$8.0$	final value

This modified example looks very similar, but has different results. When calculating  $5.0 / 2 * 4.0$  to be 10.0, the calculation is done as follows:

$5.0 / 2 * 4.0$	first calculation is $5.0 / 2$ - mix -mode operation, so change 2 to 2.0
$5.0 / 2.0 * 4.0$	now calculate $5.0 / 2.0$
$2.5 * 4.0$	floating-point calculation is made
$10.0$	final value

The final floating point value is a `double` data type.

One last example:  $1.0 + 3 / 2 + 3.0 * 4$  is 14.0. The calculation:

$1.0 + 3 / 2 + 3.0 * 4$	first calculation is $3 / 2$ - integral calculation
$1.0 + 1 + 3.0 * 4$	now calculate $3.0 * 4$ - mix-mode, so change 4 to 4.0
$1.0 + 1 + 3.0 * 4.0$	calculate $3.0 * 4.0$
$1.0 + 1 + 12.0$	order of operations, calculate $1.0 + 1$ - mix-mode, so change 1 to 1.0
$1.0 + 1.0 + 12.0$	calculate $1.0 + 1.0$
$2.0 + 12.0$	calculate $2.0 + 12.0$
$14.0$	final value

## Type Mismatch Error

When dealing with mix-mode operations, it is common to encounter a type mismatch exception. This is a compilation error, so the program will not finish compiling and will not be able to run. This error occurs when you attempt to assign a data type an incorrect type of value. You cannot assign a floating-point value to an `int` data type. The following will cause a type mismatch exception during compiling:

```
int x = sum / 2.0; //mix-mode calculation; a double value is calculated
```

For this example and others like it, there are several solutions:

1. Declare the variable as a double.
  - a. Easiest solution to bypass the error.
  - b. Likely the worst solution - if a variable is declared as an `int`, it is likely because it is intended to store an integer value. If the variable was intended to store fractional values, it would have originally have been declared to be a `double`.
2. Type cast the expression. (This is described in Casting - a future section in the text.)
3. Round the value. (This is done with the predefined Java Math class - in a future chapter.)

---

## Check Yourself

---

1. What is the final result of each of the following calculations?
  - a.  $10 * 2.0 / 4$
  - b.  $7 / 3.0$
  - c.  $5 + 4.0 * 2$
  - d.  $(4 + 11.0) / 5$
2. The following line of code causes a compilation error. What type of error will be caused and why?

```
int answer = 3.0 + 4;
```

## §3.4 Constants

### Constants

A **constant** is like a variable in that a name and type are identified. Unlike a variable, however, the value of a constant is hard-coded into the software. The constant (variable) is assigned and can never be changed. The naming convention is that all of the letters are capitalized and words are glued together with an underscore, called snake case (or snake\_case). 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? Or there is the possibility someone clicks on the URL and it doesn't function properly.). Another good reason for using a constant is because (and this URL example is perfect) if it needs to be changed, we only have to change the value of it once (in one location in the code). 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 void main(String args[]) {
        final String HELP_PAGE_URL = "http://www.stackoverflow.com";

        System.out.println("For help, go to: " + HELP_PAGE_URL);
    }
}
```

---

### Check Yourself

---

1. Which of the variables listed below are properly declared constants?

- a. `double PI = 3.14;`
- b. `final int NUM_OF_PEOPLE = 5;`
- c. `int answer_to_everything = 42;`
- d. `final double TAX_RATE = 0.08;`
- e. `int WATER_DEPTH = 314;`

2. Explain why an error will be generated from the following code:

```
final double MIN_SALARY = 45678.90;  
MIN_SALARY = 48000;
```

3. Give one reason why you might use a constant variable in your code.

## §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 point and anything after the decimal point). 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, recall 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 also cast `num2` to be a double. Of course, we *could* have cast both `num1` and `num2` to be double, but we didn'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 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()` (the `Math` class is discussed in Chapter 4).

```
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 they are encountered left to right), then addition and subtraction (as they are 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!*



---

## Check Yourself

---

1. What is the final result of each of the following calculations?
  - a.  $10 - 6.0 * 2$
  - b.  $15 + 10 / 2$
  - c.  $19 / (12 / 5) + 33.3 - 4 \% 3$
  - d.  $6 / 5 + 3.0 + 4 \% 2$
  
2. Identify the operation that will be performed first in each calculation:
  - a.  $8 / (5 + 3.0) + 4 \% 2$
  - b.  $64 * 44 + (\text{double})22$
  - c.  $56 / 5 / 3 + 32 - 5 \% 3$

## §3.7 Increment, Decrement, and Compound Assignment

Up to this point in the chapter, you have been reading about the basic arithmetic operators for working with numbers, and how to use them with variables and simple expressions. This section gets into some of the “tricks of the trade” and common tools. These are not unique to Java, but are used by many different programming languages. You could go the rest of your programming career without ever using the tools highlighted in this section. However, you will definitely see these tools in sample code found in this text and online.

### Increment Operator

There are many times that programmers need to add 1 to a variable. This is very common, and a special unary operator has been defined for this purpose. A **unary operator** is an operator that only has one operand (value). To contrast, addition is a **binary operator** that requires two operands (values). The increment operator, ++, simply adds one to the current value of the variable and updates the variable by assigning it that value. To use the operator, you simply place it directly before or after the variable. Each of the following three lines of code have the same end results: the value of the age variable is increased by 1.

```
age = age + 1;
age++;
++age;
```

The difference in writing the increment operator before or after the variable is subtle, but can play an important role in how the program works when the increment operator is used within an expression. Many programmers use the increment operator as a stand-alone statement, like in the examples above. However, this operator may be used within any expression as in the following examples. Note the code on the left performs essentially the same as the code on the right.

```
System.out.println(age);
age++;
```

```
int a = 4;
++a;
int b = 3 + a*2;
```

```
System.out.println(age++);
```

```
int a = 4;
int b = 3 + ++a*2;
```

When you write the operand before the variable (e.g. ++a), it is a pre-increment. This indicates that the variable increments by one BEFORE it is used. When the operand is written after the variable (e.g. age++), it is a post-increment. With a post-increment the variable increments by one AFTER the value of the variable is used in the expression. Using the increment operator within an expression can be very confusing, so it is recommended against doing so as it does make the code less readable.

## Decrement Operator

The decrement operator, --, simply subtracts one from the value of the variable and updates the variable with that value. This operator behaves the same exact way as the increment operator. Each of the following three lines of code have the same end results: the value of the age variable is decreased by 1.

```
age = age - 1;
age--; //post-decrement (subtract 1 AFTER using the value of the variable)
--age; //pre-decrement (subtract 1 BEFORE using value of the variable)
```

## Compound Assignment Operators

There are five compound assignment operators, also known as shorthand operators, that can be used to more efficiently program. These assignment operators are simply shortcuts to programming. The following expressions are equivalent:

```
value = value * 3; // triples the value
value *= 3; // triples the value
```

When an expression involving the shorthand operator is calculated, the right hand side of the equation is evaluated, and then the operation is done with the variable on the left hand side before updating the variable with the computed value. The main advantages of compound assignments is less typing, and some would argue that they make the code more readable. The following code provides some examples:

```
int a = 4;
a *= 3; //triple the value of 'a'
System.out.println(a); //output: 12

a /= 2; //divide the value of 'a' in half
System.out.println(a); //output: 6

a += 4 - 3 * 2; //add -2 to 'a' (RHS is evaluated first)
System.out.println(a); //output: 4

a = 123;
a %= 10; //mod 'a' by 10 and store
System.out.println(a); //output: 3

a -= 1; //decrease a by 1
System.out.println(a); //output: 2
```

---

## Check Yourself

---

1. Explain the similarities and differences between the following 2 lines of code:

```
x = x + 4;  
x += 4;
```

2. What is the final value of a after the following lines of code are executed?

```
int a = 2;  
a++;  
a += 3;  
a--;  
a *= 2;
```

## STANDARD JAVA CLASSES

*“There are three classes of people: those who see, those who see when they are shown, those who do not see.”*

- Leonardo da Vinci -

## §4.1 What is a Class?

### Difference Between Primitive Data Types and Classes

Up until now, we have been talking about variables as primitive data types; that is, as small pieces of singular data that we do things to. We can add an `int` to another `int` or we can divide two `doubles`. We can perform basic arithmetic operations with primitive data types.

But as far as variables are concerned, there is a whole new world out there. As pointed out earlier, there is the notion of a class. The technical definition of a class is fairly abstract, but for now it is probably best to think of a class as a collection of data and objects that almost-kinda-sorta behaves as a small, self-contained program. You'll find out later in life that this is actually a pretty terrible definition, but based on what we know about Java so far, it's not the worst working definition.

It might be easier to think of a class as one variable, but unlike primitive data types, classes can actually do things on their own. As a bonus, they get to hold more than one value too. In fact, a class can store multiple primitive data types all at once.

### Instantiation

Classes come in all sorts of shapes and sizes. There are (at least in Java 11) roughly 6,003 different classes. They each do specific things. Most programmers do not know all 6,003 classes. In fact, Oracle (the custodians of Java) are aware of this, and they offer an **API** (Application Programmer Interface) to help programmers figure out how to use these classes. We'll look at the API a little later in this chapter.

Classes have to (typically) be **instantiated** before we can unleash their power in a program (though there are some notable exceptions). To be exact, we have to *instantiate* an **object** of that class and use that object in our program. If that sounds confusing, it is. For a not-so-perfect-but-not-awful real life example, we can look at Jay Leno.

He has an automobile collection that is just shy of 300 vehicles. Of those 300 vehicles, 169 are cars and the rest are motorcycles. So when Jay says, "I'm going to take a car from the garage and drive in to town," he is being very generic. Listeners might conjure up their own image of a car in their head. One person might think it's a 1969 yellow Volkswagen Beetle that is worth \$16,000. Someone else might think of a 1981 silver DeLorean DMC-12 valued at \$45,000. But Jay might be thinking of his 2014 McLaren P1 that is worth \$1.5 million.

Each of those vehicles is an **instance** of the vehicle class. In our heads, we each have a blueprint of a vehicle (it has some number of wheels, it has a color, it has a make and a model), but the specific details are what make it unique.

Likewise, classes need to be instantiated before we can use them *unless* they have **static** methods that can be used. If that sounds familiar, it should - we've already made our own classes with static methods. And we didn't have to instantiate the class to call those methods. As you'll see below, there is one class in particular - the `Math` class - that we don't ever instantiate. There's also a class - `String` - that doesn't get instantiated the formal way. Every other class we'll use this semester will have to be instantiated the official way.

If you're really craving an example of instantiating an object from a class, peek down at the `Scanner` section and then come back up here.

### Parameters

Some methods also accept **parameters**. A parameter is information you give the method so it can use that

specific piece of information and do something with it. In a minute, we'll take a look at the Math class. But before we do, let's look at just one of the methods, round(), from the Math class. The parameter is a double, and the method will return to us the value of that double rounded up or down:

```
System.out.println(Math.round(4.6)); // 5
System.out.println(Math.round(4.3)); // 4
```

The parameter was 4.6 in the first example and 4.3 in the second. Though it was not the case in this example, there are some methods that can accept more than one parameter. You'll see that later in this chapter with Math.pow().

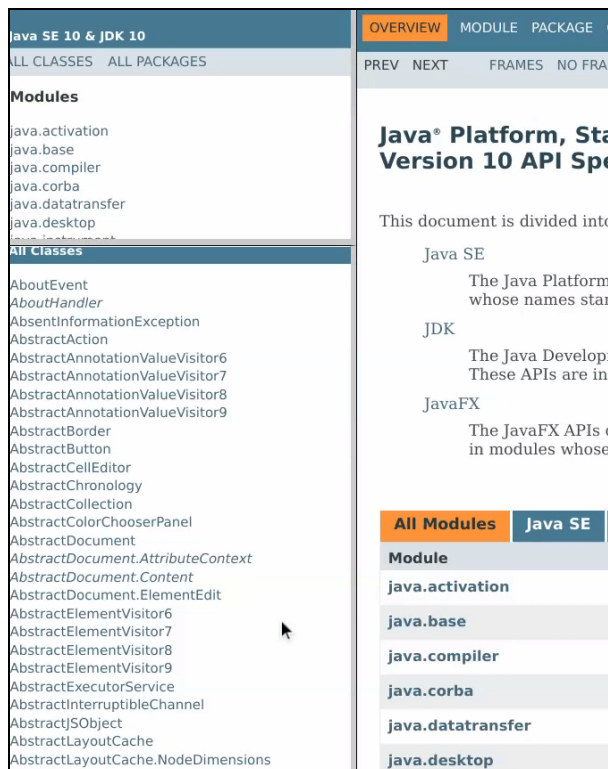
## Methods

Classes also have methods. We can think of a method as a mini-program that is contextually relevant to the class it is in. For example, if there was a class called ConnectFour, it may have a method called checkForFourInARow() or a method called clearGameBoard(). A class designed to track caloric burn might have a method called convertStepsToCaloriesBurned() or a method called calculateExpenditure().

This is important because we've already created methods unique to the classes we've authored (like when we shifted the code down to a method called printHello() in chapter 2). We will examine methods more thoroughly in a later chapter.

## API

The API (Application Program Interface) is like an instruction manual for every class. You can look at the API of a class and you should be able to see a list of all the variables and methods in that class. The API will also help you understand how to use the methods. You can look at the [API for Java 11 here](#), but check out this image that attempts to show just how many classes there are:



## §4.2 Math Class

### Constants in the Math Class

The Math class has two constants that are useful and a few methods that are helpful as well. The Math class can always be used in any Java program without using an **import** statement or instantiating it. It is so popular that you can just call on it whenever you'd like.

For instance, if you are trying to calculate the area of a circle with a radius of 10, you will need to use the value of pi. Most people know that pi is *roughly* 3.14, and if you are a bit nerdier you could go to 3.14159. Certainly that is a bit more accurate. But you could, instead, type `Math.PI` and the compiler would use the value of pi that extends to fifteen decimals. To find the value of the area, we need to know the formula  $A = \pi r^2$ . But since we don't know how to do exponents yet, let's just multiply the radius by itself:

```
System.out.println("The area is " + (Math.PI * 10 * 10));  
// 314.1592653589793
```

Likewise, you can also use `Math.E` which is used in natural logarithms.

```
System.out.println("The value of e is " + Math.E);  
// 2.718281828459045
```

### Methods in the Math Class

Turns out there are methods in the Math class you can use too. There are around 80 different methods in the Math class. If you're interested, you can check out the [Math Class API](#), but we only use a handful of them in this course. You can check out the [Relevant API](#) (in the Appendix) which highlights some of the more popular ones, but let's look at one or two so you can get a feel for them. Note that these are only a few of the methods we will be looking at. It's also important to note that some methods, like `Math.max()`, exist in multiple forms. One of them takes in integers, and one takes in decimals.

RETURNS	NAME & PARAMETERS	DESCRIPTION & EXAMPLE
double	<code>Math.max(double, double)</code>	Returns the biggest of the two numbers  <code>Math.max(4.5, 7.1); // returns 7.1</code> <code>Math.max(9, 2); // returns 9.0</code>
int	<code>Math.max(int, int)</code>	Returns the biggest of the two numbers  <code>Math.max(4, 2); // returns 4</code> <code>Math.max(-8, -9); // returns -8</code>
double	<code>Math.pow(double, double)</code>	Returns the result of the first number raised to the second number  <code>Math.pow(5, 2); // returns 25.0</code> <code>Math.pow(16, .5); // returns 4.0</code>



double	<code>Math.random()</code>	Returns a random number that is $0 \leq n < 1$  <code>Math.random(); // 0.46572967856131986</code> <code>Math.random(); // 0.864801958005641</code>
long	<code>Math.round(float)</code>	Returns the number rounded to the nearest whole number  Note: <code>Math.round()</code> can accept a variable of any type that can be promoted to <code>float</code> (for example, <code>double</code> or <code>int</code> ). It returns a <code>long</code> , but we often use casting to get an <code>int</code> result.  <code>Math.round(3.789); // returns 4 (as a long)</code> <code>(int)Math.round(8.213); // returns 8 (as an int)</code>

If you didn't look at that table carefully, you may have missed that there are two methods called `Math.max()` and they look almost identical. The difference is small - one of them takes in two `doubles` and one takes in two `ints`. Computers care about this type of thing, but we don't really. We just know that when we put in two numbers, `Math.max()` will return the bigger of the two.

## §4.3 String Class

Let's chat about the `String` class now. It's sort of an oddball class in that it behaves more like a primitive data type than an object. We can declare and assign a `String` variable in the same way we can a primitive:

```
String name = "Barney Stinson"; // Barney Stinson
```

We can also modify it as we would a primitive:

```
String name = "Barney Stinson"; // Barney Stinson
name = "NPH"; // NPH
```

*Technically, we aren't exactly modifying it. When you reassign a primitive, you really are changing the value. But when you reassign a `String`, you are actually creating a whole new `String` and replacing it.*

And in a kind of surprising case, we can add to it like we would a primitive:

```
String name = "Barney Stinson"; // Barney Stinson
name = "NPH"; // NPH
name += " (aka Doogie)"; // NPH (aka Doogie)
```

But this is where the similarity stops - `Strings` can do so much more. Much like the `Math` class has methods that can be used, `Strings` do too. There's actually a number of different things you could do with a `String`. You could check out the [official documentation for the `String`](#), but there is a lot to digest there (roughly 75 methods). You're better off looking at the [Relative API for `String`](#) in the Appendix. Just like we curated a list of popular `Math` methods, we've listed the popular `String` methods as well.

One thing you *should never do* is test for equality between `Strings` with the “`==`” operator. That's a big no-no. It has to do with how objects are stored (again, a lesson you'll learn in another Java class). Instead, use the `.equals()` method.

Here are a few of the methods you should keep your eye on. Keep in mind one thing: a `String` is **zero-indexed**, so the first character is in index number 0, the second character is in index number 1, and so on.

RETURNS	NAME & PARAMETERS	DESCRIPTION & EXAMPLE
char	<code>charAt(int)</code>	Returns the <code>char</code> in the <code>String</code> at the specified index  <pre>String name = "Skywalker"; char letter0 = name.charAt(0); // S char letter5 = name.charAt(5); // l char letter20 = name.charAt(20); // ERROR // StringIndexOutOfBoundsException</pre>
boolean	<code>equals(String)</code>	Returns true if the two <code>Strings</code> are lexicographically equivalent, false otherwise  <pre>String a = "Tony Stark"; String b = "Tony Stark"; String c = "Elon Musk";</pre>

		<pre>System.out.println(a.equals(b)); // true System.out.println(a.equals(c)); // false</pre>
int	indexOf(String)	<p>Returns the index of the first time the parameter String appears in the String calling it, or returns -1 if the String is not found</p> <pre>String a = "brand"; System.out.println(a.indexOf("an")); // 2 System.out.println(a.indexOf("zed")); // -1</pre>
int	indexOf(char)	<p>Returns the index of the first time the parameter char appears in the String calling it, or returns -1 if the char is not found</p> <pre>String a = "Star Wars"; System.out.println(a.indexOf('r')); // 3 System.out.println(a.indexOf('k')); // -1</pre>
int	length()	<p>Returns the length of the String (the actual number of characters in the String)</p> <pre>String a = "Tesla"; System.out.println(a.length()); // 5 System.out.println("Edison".length()); // 6</pre>
String	substring(int)	<p>Returns a String that starts at the position passed in and goes until the end of the String</p> <pre>String a = "Captain Marvel"; System.out.println(a.substring(8)); // Marvel System.out.println(a.substring(12)); // el</pre>
String	substring(int, int)	<p>Returns a String that starts at the position of the first int passed in and goes until one before the second int passed in</p> <pre>String a = "Avengers"; System.out.println(a.substring(1,4)); // ven System.out.println(a.substring(0,6)); // Avenge</pre>

---

## 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 o

utput 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 III

## §4.4 Scanner Class

A Scanner is needed to get input from the user. It is not entirely inaccurate to say that a Scanner will look at all the commands coming in (usually from the keyboard) and has the ability to do something with all that input. To use a Scanner in our software, we need to do two things - we need to **import** it and also instantiate it.

### import

Since there are around 6,000 classes in Java, not all of them are natively included in your program. In fact, most of them aren't. `String` and `Math` are always included, but in order to use many other classes - including `Scanner` - you have to import them. All this means is that before you start coding the main part of your program, you need to use the **reserved word** `import` and tell the compiler what package to use (it happens to be the case that `Scanner` resides in the `java.util` package).

In this example, notice that the `import` statement is right at the top - before the class starts.

```
import java.util.Scanner;

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

        runScannerDemonstration();

    }

    public static void runScannerDemonstration() {

        Scanner scanner = new Scanner(System.in);
        // This instantiates an object named "scanner" that is from the
        // "Scanner" class, and it will be getting input from "System.in"
        // which is pretty much the keyboard

        System.out.println("Enter a number:");
        int num1 = scanner.nextInt(); // 20

        System.out.println("Enter a number:");
        int num2 = scanner.nextInt(); // 15

        System.out.println("The sum is: " + (num1 + num2));
        scanner.close(); // Closes the scanner so it doesn't cause issues
    }
}
```

```
Enter a number:
20
Enter a number:
15
The sum is: 35
```

## Using a Scanner

So there are a lot of things to know about the Scanner class. For a more detailed description, check out the [Scanner entry in the Relevant API](#).

### Conventions for Instantiating

When you instantiate something (that is, create one specific instance of a class), you must follow a particular pattern. Similar to declaring and assigning a primitive variable, you'll need to specify the type of the variable and the name of the variable:

```
Scanner scanner;
```

The above code segment will create an object from the Scanner class named "scanner". We could have named it anything (fluffy, voteForPedro, theRock), but usually naming an object after the class it is derived from makes the code easy to follow.

As with primitive variables, we can both create declare the variable and assign it at the same time. The catch is that the assignment part is a bit trickier. There is a specific way to create an instance of a class - we must tell the compiler to create a new instance of the class and give it (if necessary) any information it needs to create the object. In this case, we will use `System.in` (which is the input from the keyboard - sorta). We could have instead put a file name or a channel or a stream in there, but we will almost always use `System.in` for the entirety of this class.

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

That's it! We've created a Scanner called "scanner" that we can use to capture input! Now we just need to know the methods available to us. Just like `Math` and `String`, there are a ton of methods we can use ([in the Java API](#)), but we'll just use a small subset in this class ([our Relevant API](#)).

### `.nextInt()`

Use this method to read in the value of an integer. You *can* use this method by itself, but it is almost always used as part of an assignment statement for a variable. For the next few examples, we will assume a Scanner has been instantiated and is called "scanner":

```
System.out.print("Enter your age:"); // Prompt the user
int age = scanner.nextInt(); // Takes the integer the user
                             //typed and stores it in 'age'
```

That's it! It's always a good idea to **prompt** the user with something on the screen so they can get a sense of what to type in.

Note that this program would have crashed if someone entered a decimal instead of an integer because the `nextInt()` method is expecting an `int` *and only* an `int`.

## `.nextDouble()`

Just like `nextInt()`, but works with a double. In this case, if a user types in an int, that's okay! Java will automatically promote an int to a double.

```
System.out.println("What is 5 divided by 2?"); // Prompt the user for answer
double theirAnswer = scanner.nextDouble(); // Store input in 'theirAnswer'

System.out.println("What is 6 divided by 2?"); // Prompt the user for answer
double anotherAnswer = scanner.nextDouble(); // Works even if an int is input
```

## `.nextLine()`

Use the `nextLine()` method with a Scanner if you want to take in a `String` instead of an `int` or `double`. Note that this will read all the input from wherever the scanner is to the end of the line (literally). This has implications for program flow, and this quirky behavior is discussed in the next section.

```
System.out.println("Enter your name:"); // Prompt the user
String name = scanner.nextLine(); // Angus MacGyver
```

## `.close()`

Always remember to close a Scanner when you are done with it. Usually this is done right before the method ends. You can do this by using the code

```
scanner.close();
```

## Quirky Behavior with a Scanner

So there is this really odd thing with Scanners, and you are almost certainly going to encounter it many times. The way a Scanner thinks of information is in lines. For instance, this paragraph has 7 lines in it. And a Scanner does not always go to the next line when it is done reading. In fact, a Scanner will *only* go to the next line if we tell it to by using the `nextLine()` method. That's a little confusing because `nextLine()` is also the method to use when taking in a `String` that the user typed in. Both of those are true! An invocation of the `nextLine()` method will read everything from where the Scanner is now until the end of the line and *then* go to the next line.

This is confusing because the Scanner will *not* automatically go to the next line when called by `nextInt()` or `nextDouble()`. And that is okay sometimes. The following walkthrough might help. We'll use a double arrow (`>>`) to symbolize where the Scanner actually is.

```
>>This is a test
of the Scanner
40
93.7
The Liam Neesons
```

The Scanner is at the beginning of the first line of text. When we call `nextLine()` on it, it will travel all the way to the end of the first line, capture all the text "This is a test", and then go to the next line.

```
String firstLine = scanner.nextLine();
```

```
This is a test
>>of the Scanner
40
93.7
```

Now the Scanner is on the second line of text. That's great - everything is going according to plan. So when we call `nextLine()` again, it will capture "of the Scanner" and go to the next line.



The Liam Neesons

```
String secondLine = scanner.nextLine();
```

```
This is a test  
of the Scanner  
>>40  
93.7  
The Liam Neesons
```

Looks like we should use a `nextInt()` here so we can capture the 40. But don't forget that a call to `nextInt()` will *not* put the Scanner on the next line - rather, it just stops moving after the int has been scanned in.

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

```
This is a test  
of the Scanner  
40>>  
93.7  
The Liam Neesons
```

Aha! Although this looks like it might be problematic, it really isn't all that worrisome. If we invoke `nextDouble()`, the Scanner will actually cruise along until it is done scanning in a double, *even if it has to go to the next line before a double is encountered*.

```
double grade = scanner.nextDouble();
```

```
This is a test  
of the Scanner  
40  
93.7>>  
The Liam Neesons
```

Okay, not a big deal. The Scanner is still sticking around after the 93.7 but this isn't awful. The problem is that most new programmers might think a call to `nextLine()` will scan in the text "The Liam Neesons". But it won't - `nextLine()` only scans in *what it encounters before going to the next line*, which in this case is nothing.

So to make this work right, we need to actually call `nextLine()` all by itself to send the scanner to the appropriate place, and then call `nextLine()` again to read in the text.

```
scanner.nextLine();
```

```
This is a test  
of the Scanner  
40  
93.7  
>>The Liam Neesons
```

Now we are poised to read in "The Liam Neesons" with `nextLine()`.

```
String keyPeele = scanner.nextLine();
```

```
This is a test  
of the Scanner  
40  
93.7  
The Liam Neesons>>
```

Awesome!

## §4.5 Random Class

### Importing

Just like the Scanner class, we import the Random class. Happily, it even lives in the same package (`java.util`). If we are writing a program that uses both Scanner and Random, we have two different options for importing. We can import each one explicitly:

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

We can also import *everything* in the `java.util` package by including an asterisk ( `*` ). That is referred to as a **wildcard**, and is used to represent *everything* in the package.

```
import java.util.*;
```

### Instantiating

There are two different ways to instantiate an object of the Random class. The usual way is to use no parameters:

```
Random random = new Random();
```

This will create an object that is as random as Java will let it be (it's really **pseudorandom**). So if we generate a random number with this object, there is no way we could really know what that number is.

Alternatively, we can create an object of the Random class with a **seed**. That will allow us to predict the random numbers. You might be saying to yourself, “Self, why would we want to predict random numbers? Doesn't that defeat the point of randomness?” And you'd be right. We typically use this method when we are in the process of writing software. It helps us see if errors exist in other parts of the code. After all, if we have no idea what the random numbers we are using are, how can we tell if the code is right?

```
Random random = new Random(42); // Creates a Random object with a seed of 42
```

This might help explain why you'd want to do this. In the following code, we'll create two different Random objects. Each of them will generate two different numbers between 0 and 99. We can look at the output from running this program a few times in a row.

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

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

        randomNumbersDemo();
    }

    public static void randomNumbersDemo() {
```

```

Random random = new Random();
Random randomSeeded = new Random(42);

System.out.println(random.nextInt(100));
System.out.println(random.nextInt(100));
System.out.println(random.nextInt(100));

System.out.println();

System.out.println(randomSeeded.nextInt(100));
System.out.println(randomSeeded.nextInt(100));
System.out.println(randomSeeded.nextInt(100));

}
}

```

TRIAL ONE	TRIAL TWO	TRIAL THREE
92	12	97
90	18	61
8	95	46
30	30	30
63	63	63
48	48	48

It is very likely that if you tried replicating this on your computer, you would always get 30, 63, and 48 as the three “random” numbers from the Random object seeded with 42.

## Methods

Just like Math and Scanner, there are a lot of methods we won’t use (but you can [scope out the API](#) if you’re interested). Instead, check out the [Relevant API](#) for the high value methods we will be using. Here are a few of them:

### nextInt()

This will generate a random number within the range of ints, which is roughly -2.5 billion to 2.5 billion. If you use this, expect a lot of big numbers.

```

Random random = new Random();
System.out.println(random.nextInt()); // Any valid int

```

### nextInt(int)

If you use nextInt() with a parameter of type int, you bound the values. This method will generate a random number between 0 and the parameter (but exclusive of the parameter).

```
Random random = new Random();  
System.out.println(random.nextInt(10)); // Between 0 and 9, inclusive
```

### **nextDouble()**

This will return a double between 0 and 1.0 (exclusive of 1.0). If this sounds similar to `Math.random()`, that's because it is. In fact, under the hood, `Math.random()` actually uses the `Random.nextDouble()` method!

In addition to all the classes that come with Java, there is the ability for the programmer to create their own class. It happens all the time. By default, any program you've written is a class, but it gets deeper than that - you'll be creating your own classes to use in other classes if you continue programming in Java (just not this semester). You should really check out the [Relevant API](#) for more methods and examples of classes.

## INTRODUCTION TO SELECTIONS

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

- Dave Ghidiu -

## §5.1 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. The computer makes decisions by using Boolean Logic (where the answers are `true` or `false`) in `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 two 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).

```
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 nerdland, 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  $\geq$  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
```

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

---

## Check Yourself

---

1. Describe what each of these symbols mean:
  - a. `==`
  - b. `!=`
  - c. `<`
  - d. `<=`
  - e. `>`
  - f. `>=`
  
2. What are the truth values of the following expressions?
  - a. `4 > 44`
  - b. `(3 + 4) <= 8`
  - c. `(7 * 5) % 2 == 1`
  - d. `8/2 != 4`

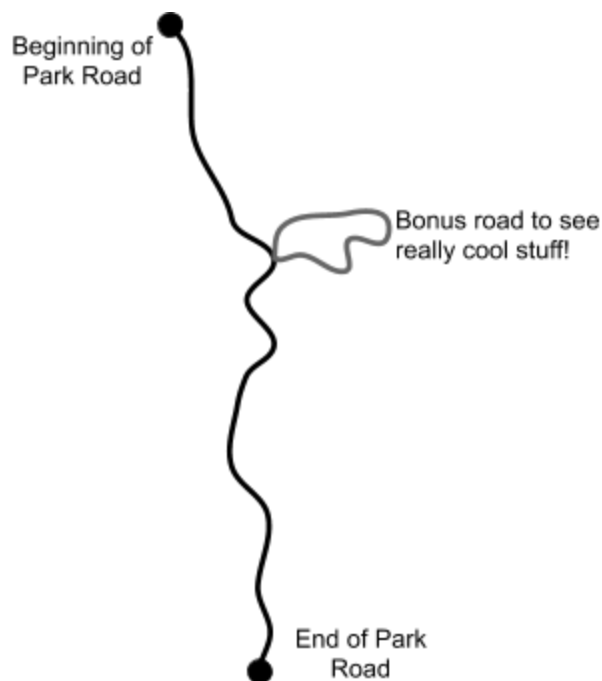
## §5.2 if Statements

### 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 the user punched in "3", then the code in the brackets would be ignored. In either case "Game over." would be output because it's just plain ol' 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`. So the following are valid examples of conditions:

```
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 (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
}
```

This could also be written as:

```
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 - you'll read more about methods later (you may even be using them already!).

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 me. By the way - the correct comparison (as we 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 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");
}
```

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). 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 -



## §6.1 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, and 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 assigned! 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 executed. 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. Describe the difference between multiple if-statements and an if-else structure by explaining what happens when the following code examples are executed:

Code A:

```
int x = 5;

if (x < 10) {
    System.out.println("The number is less than 10.");
}
if (x < 20) {
    System.out.println("The number is less than 20.");
}
if (x < 30) {
    System.out.println("The number is less than 30.");
}
```

Code B:

```
int x = 5;

if (x < 10) {
    System.out.println("The number is less than 10.");
} else if (x < 20) {
    System.out.println("The number is less than 20.");
} else if (x < 30) {
    System.out.println("The number is less than 30.");
} else {
    System.out.println("This is a large number!");
}
```

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

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 = new 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';
}
```

## §6.2 Compound Conditionals and Boolean Operators

A **compound conditional** is nothing more than an `if` statement that tests multiple conditions.

To create these conditionals, we use **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.

### Not

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

!

The “not” operator 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.



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.

Here's another example of the NOT operator in action:

```
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:

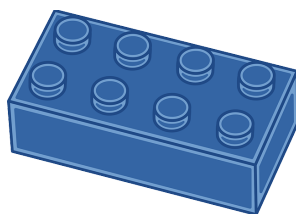
```
int i = 10;
int j = 150/15;

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

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

## And

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:



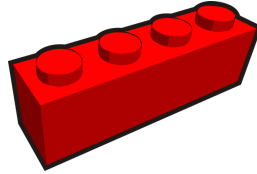
In this case, the Lego sorting algorithm may do a preliminary sort and want to consider any brick that is at least two rows wide and four columns long. The key concept is *and* - both conditions must be true. The code may look like this:

```
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 condition. 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  
}
```

When the conditional is evaluated in this case, it doesn't really matter what the value of brickLength is because the totality of the if statement will be false. In fact, the second condition could even include code that would cause a runtime error, but the software will not fault because the second condition doesn't even get considered:

```
int brickwidth = 1;  
if ((brickWidth >= 2) && (5/0 > 1)) {  
    // do something cool  
}
```

Even though 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, as a "pipe." The *or* statement requires that at least one of the conditions be true - but if they both are true, 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 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 parentheses 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 parentheses:

```
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.

## 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!

## 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](#)).



---

## 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.  $!(!i \ \&\& \ !j)$

4. Describe short-circuit evaluation, and explain why it is important.

## §6.3 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 let 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 are no `break` statements, 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.

## §6.4 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; 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 = "odd";
}
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. Technically speaking, there is no such thing as a global variable in Java, but it's pretty dang close. Note: the following code example includes a method with a return type other than void-methods are up next in Chapter 7.

```

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 methods and 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. The following code has a compilation error. Identify the error, and rewrite the code so that it will compile and run successfully.

```
int num = 17;

if (num > 0) {
    String result = "positive";
} else {
    String result = "non-positive";
}
System.out.println("The number is " + result);
```

3. In your own words, describe 'scope'.



## METHODS

*“Though this be madness, yet there is method in it.”*

- William Shakespeare -

## §7.1 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 many modules that each have 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 superheroes, then methods are the individual superheroes, 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.

You should note that `glitch()` has a **header** that looks very similar to `main()`, which is also a method. The way the header is defined gives it a unique **method signature**. Obviously the name of the methods `glitch()` and `main()` make their signatures different, but you will see that there are other elements in the header that can make them distinct.

```
public static void glitch() {  
    // the body of the method goes here ...  
}
```

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 special case, `glitch()` may break everything with a divide by zero runtime error!

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 should come back to `main()` and, when the end of `main()` is reached, the program would end.

Take another look at those method stubs above and pay special attention to the headers. Do you see some additional elements besides the names that make the method signatures unique? They are for passing data in and out of methods when they are called. Soon, you will be able to take advantage of those superpowers.

---

## *Check Yourself*

---

1. What makes the main method special?
2. What term is used to describe method declarations with enough detail to be called but bodies that are incomplete?
3. **Answer true or false:** Because they are used for actions, it is good practice to name methods using a verb.

## §7.2 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 those variables 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. 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 multiple 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;
    }

    public static void main(String[] args) {
        whisper();
        System.out.println(secret);
    }
}
```

## §7.3 Predefined Methods: Learn from Math

As covered earlier in this text, the Java language specification includes a set of standard libraries for programmers to take advantage of. So, let us take advantage.

It will be useful to keep the earlier chapter on predefined classes available for reference, but it is a good habit to check on the official documentation too. The Math class documentation is here:

<https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html>

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. To help us, we will 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—>

<code>static double</code>	<code>random()</code> 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 that 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 ints, 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 parentheses 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? What about 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 80 inclusive
```

Incidentally, while exploring Math.random() is a good academic exercise, you will want to research and use the Random class that is part of the java.util package. It is more flexible, faster, and potentially gives more evenly distributed random results. The Random class is covered in Chapter 4 in this text.

## 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 often 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 parentheses. 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 the method 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 `55.0` is returned, and then, our statement sets `flooredNumber` to `55.0`. It might not 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 yield a result of 55 from `floor()`, how can we get it to round?

When rounding to the nearest one, a value of 5 or above in the tenths digit 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 55.0. That same 0.5 added to 55.999, though, would give us 56.499. Send that to `floor()` and you'll get 56.0. That's what we want!

Okay, so let's round a number using `floor()`.

```
double roundedNumber = Math.floor(55.95 + 0.5); // 56.0!
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 only 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 if you 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. See section 4.2 for more details on `Math.round()`. Then try it out for yourself!

## Multiple Parameters

It is time to 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 `doubles`. 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! We can set up 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(Math.pow(10, 2)); //outputs 100
System.out.println(Math.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!

## **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...  
}
```

## §7.4 Defining Your Own Methods

When it comes to working with methods, there is maybe one experience more rewarding than finding a needed predefined method: writing your own awesome method!

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. However, 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

        scanner.close();
    }
}
```

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 there's a web page that shows a chart of what the star ratings mean. Perhaps they will also be shown in some database views that aren't in the original article. 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);
        scanner.close();
    }
}

```

## 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
//          identifier

```

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.

We placed `starRating` in the parentheses 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);
        scanner.close();
    }

    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 the `starRating` variable, 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 that summarizes 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 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. If there is a case where a return statement that can be reached is missing, 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 the method, 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."*

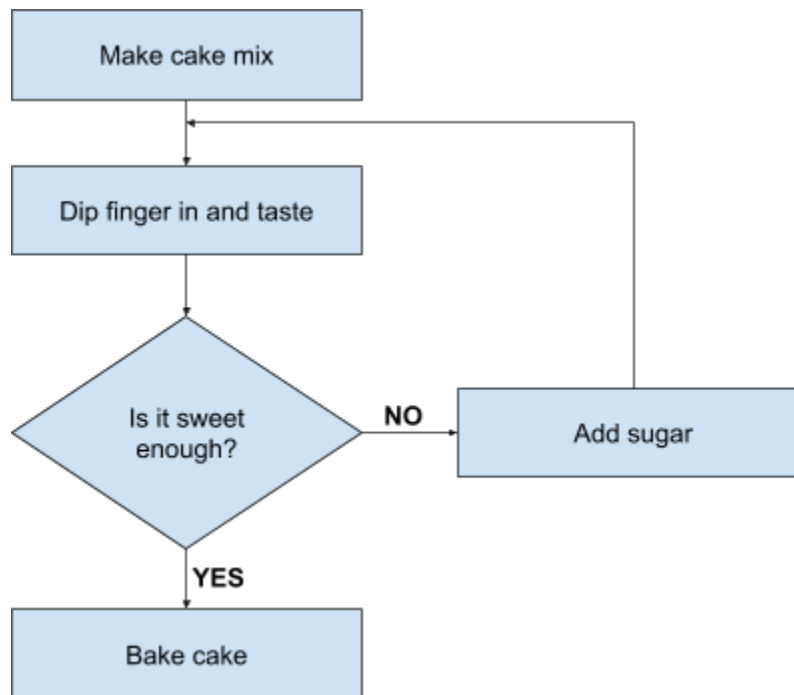
- Malcolm Kotok -

## §8.1 Control Structures - Loops

You know the `if` statement--a control structure that can have the program elect to either execute a block of code, or skip the code. 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. 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*

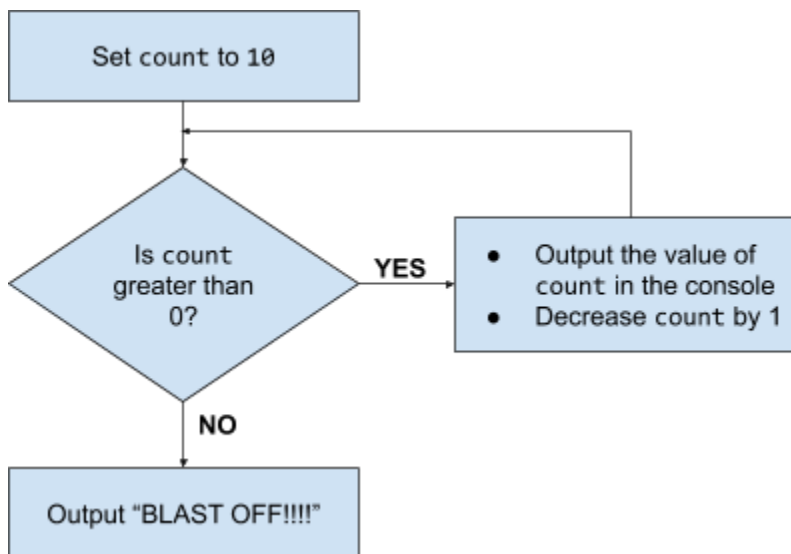


## §8.2 The while statement

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 conditional and tests it again. Depending on the result, the program will either 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!!!!");
```

```
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 countdown  
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 currently 10, so this expression evaluates to true. Because the boolean expression is true, the body of the loop will be executed (lines 4 and 5)
- The count (which is 10) 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 all the code in the body of the loop will run
- 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. Control then 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

---

## ***Check Yourself***

---

1. Other than a countdown program, where do you see loops being useful?
2. Do you think indenting is important when writing loops? Why or why not?
3. What would happen if the statement(s) in the while loop were not between brackets?
4. Consider the following code segment:

```
int a = 0;
while (a <=3) {
    System.out.println(a);
    a++;
}
System.out.println("Game over man!");
```

What is the output of the code segment?

5. Consider the following code segment:

```
int b = 10;
while (b > 0) {
    System.out.println(b);
}
```

There is a logic error with the code segment. What is it?

## §8.3 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:

```
//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!!!!");
```

```
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!!!!");
```

```
10!
10!
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 running program in your IDE. If you are in a Linux environment (command line), CONTROL + C will usually terminate the program.

---

## ***Check Yourself***

---

1. Why do you suppose an "infinite loop" gets the name?
  
  
  
  
  
  
  
  
  
  
2. What key combination usually makes a program stop running (in Java)?

## §8.4 Common while Loops

There are common algorithms or tasks done with the `while` statement:

1. Running instructions some predefined 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 the loop. 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);
    count++; // same as count = count + 1;
}
```

There are several variations 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 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 total (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

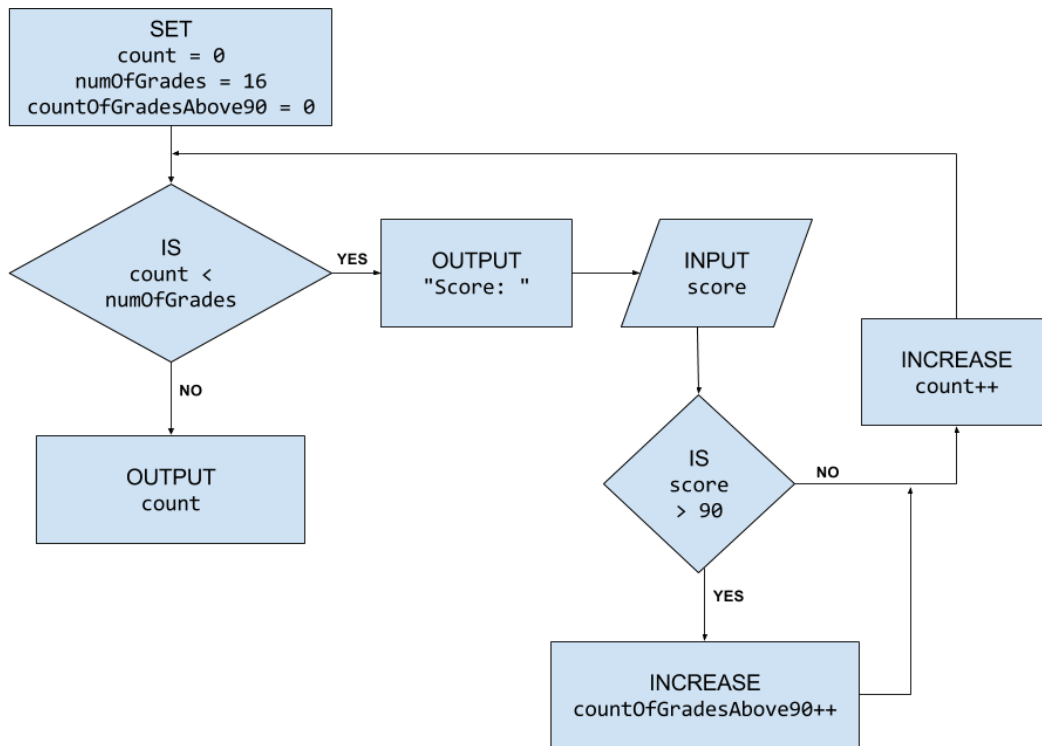
while (count < numOfScores) {
    System.out.print("Score: ");
    int score = input.nextInt();
    sum = sum + score; //could have used: sum += score;
    count++; //same as count = count + 1;
}

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 that 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 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!");
  
```



## Search

The sentinel controlled loop is a common loop where you loop while the loop control variable is not a particular value. Two examples below illustrate that it can be used as a type of search method, or as a way to provide the user to stop a loop.

```
Scanner in = new Scanner(System.in);
Random ran = new Random(12345); //set the seed
int x = ran.nextInt(20) + 1;
int y = ran.nextInt(20) + 1;
int guess = 0;

while (x + y != guess) {
    //prompt for guess
    System.out.print("What is " + x + " + " + y + ": ");
    guess = in.nextInt();

    if( guess != (x + y) ) {
        System.out.println("Incorrect, try again ...");
    }
}
System.out.println("\nNice work! " + x + " + " + y + " = " + (x + y) + "!");
```

```
What is 12 + 1: 14
Incorrect, try again ...
What is 12 + 1: 10
Incorrect, try again ...
What is 12 + 1: 13

Nice work! 12 + 1 = 13!
```

```
Scanner in = new Scanner(System.in);
final int STOP = -1;
int val = 0;
int sum = 0;
System.out.println("Enter positive integers to add, enter -1 when done.");
while(val != STOP) {
    sum += val;
    val = in.nextInt();
}

System.out.println("\nThe sum = " + sum);
```

OUTPUT:

```
Enter positive integers to add, enter -1 when done.  
5  
2  
8  
-1  
  
The sum = 15
```

---

## *Check Yourself*

---

1. Consider the following code segment:

```
int a = 0;  
int b = 10;  
while (a < b) {  
    System.out.println(a + " " + b);  
    a += 2;  
    b++;  
}
```

What is the output of the code segment?

2. Consider the following code segment:

```
int c = 0;  
int d = 10;  
while (c < d)  
    System.out.println(c + " " + d);  
    c++;  
    d--;
```

What will happen when this code segment is run?



## §8.5 The for Loop

Recall from section 8.3 that 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 and 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 = 1; i <= 10; i++) {  
    System.out.print(i + " ");  
}
```

1 2 3 4 5 6 7 8 9 10
----------------------

Note that although the last number to be output is 10, the actual last 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 2a: Output multiples of 5

```
// Print the first 6 multiples of 5  
for (int i = 0; i < 6; i = i+1) {  
    System.out.print( 5 * (i+1) + " ");  
}
```

```
5 10 15 20 25 30
```

Note that in this case, the conditional specifies that  $i < 6$ , not  $i \leq 6$ . That's because we started at 0 and wanted six iterations. If we had used  $i \leq 6$ , we would have output seven numbers.

## Example 2b: Output multiples of 5

```
// Print the first 6 multiples of 5
for (int multiple = 5; multiple <= 5*6; multiple = multiple + 5) {
    System.out.print( multiple + " ");
}
```

```
5 10 15 20 25 30
```

## Example 2c: Output multiples of 5

```
// Print the first 6 multiples of 5
int value = 5;
//loop 6 times
for (int count = 1; count <= 6; count += 1) {
    System.out.print(value + " ");
    value += 5;
}
```

```
5 10 15 20 25 30
```

### Example 3: Print a table for squares and cubes

```
//print a table of perfect squares and cubes
System.out.println("value\tquared\tcubed");
System.out.println("-----");

for(int i = 1; i <= 10; i++){
    System.out.println(i + "\t\t" + (i*i) + "\t\t" + (i*i*i));
}
```

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

---

### Check Yourself

---

1. What do you perceive as the main difference between for loops and while loops?

2. Consider the following code segment:

```
for (int i = 0; i <= 20; i++) {
    System.out.println(i);
}
```

What is the last number to be output on the screen? What is the last value of i?

## §8.6 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.

## §8.7 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 (though possibly more problematic) 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. 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; it is good practice to avoid break and continue unless you absolutely need to use them.

---

## Check Yourself

---

1. Why do you think you should use break sparingly? Can you imagine a scenario when you cannot write a program without using the break statement?

2. Consider the following code segment:

```
String password = "";
for (int i = 0; i < 3; i++) {
    System.out.println("Please enter the password:");
    password = scanner.nextLine();

    if (password.equals("p@ssp0rt") {
        System.out.println("Password accepted");
        break;
    }
}
```

Why is the break statement inside the if statement?

## ARRAYS

*Q: Why did the programmer quit their job?*

*A: Because they didn't get arrays!*

- Programmer Folklore -

## §9.1 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 options.

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 all them separately is just not manageable.

This is where **arrays** come in. An array is 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:



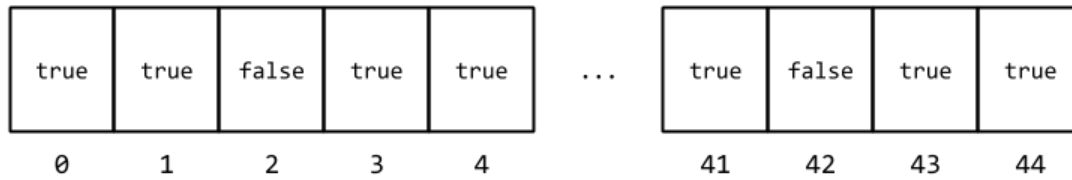
*Arrays in real life*

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. Let's assume 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.

boolean[] student01



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). The numbers on the bottom are the **index numbers** for the elements in the array.

Arrays are "zero-indexed," just like Strings, 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 that's really the 43<sup>rd</sup> 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.

## §9.2 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 in position 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
  - 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 (as shown earlier) 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 while the program is running, there are instances when an array is hardcoded with data right from the get-go. To do this, the declaration (the left hand side) 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

## §9.3 Accessing Information in Arrays

Knowing a bit about arrays is great in theory, but it takes some technical skills to handle arrays in a practical way.

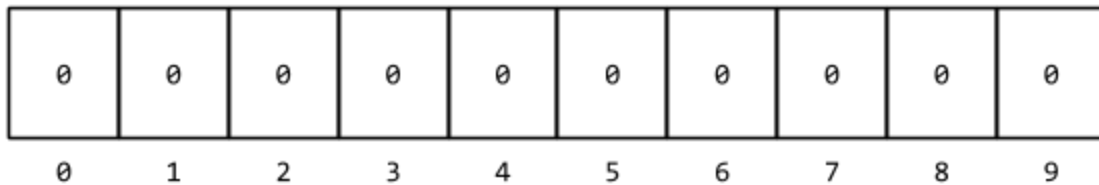
### Populating an Array

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:

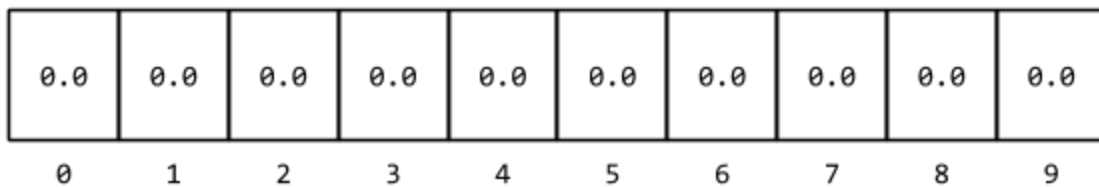
```
int[] arrOfInts
```



An array of doubles called `arrOfDoubles` looks like this:

```
double[] arrOfDoubles = new double[10];
```

```
double[] arrOfDoubles
```



An array of `boolean`s can be declared and assigned as follows:

```
boolean[] arrOfBooleans = new boolean[10];
```

It looks like this:

```
boolean[] arrOfBooleans
```

false	false	false	false	false	false	false	false	false	false
0	1	2	3	4	5	6	7	8	9

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];
```

```
String[] arrOfStrings
```

null	null	null	null	null	null	null	null	null	null
0	1	2	3	4	5	6	7	8	9

Knowing that an array is by default created with the default version of each type in every element, it is natural to then 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 say 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 hand code it:

```
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 isn't as clear-cut as you may think. For instance, in section 9.2 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.println` with an object as the parameter will always output the unsigned hexadecimal representation of the hash code of the object.

So, we have to output the values by referencing each element we want to output, one at a time. Of course, this would be super easy with a loop (see section 9.4 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`:

int[] arr

155	32	34	555	345	734	-76	0	-1	765
0	1	2	3	4	5	6	7	8	9

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 give the number of cells in the array. Likewise, a call to `arr[arr.length - 1]` will always reference the last element in the array (indexed at `arr.length - 1`).

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?

## §9.4 Iterating Through Arrays

With arrays comes the task of iterating through arrays. This means walking through the array- visiting every element and checking the value stored in each one.

### Examples of Iterating Through Arrays

For demonstrative purposes, let's look at four 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 these examples, this will make sense.

#### Outputting an array

Let's say 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 printed.

#### 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, which is 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 an 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 ints, and we are curious about if one specific int, say 33, is in the array.

Well, first we'd have to start at the first element. We'd 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 on to the second element (and then look to see if the item in there 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 in code:

```
// 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 element, and then peek into each element sequentially until you find what you are looking for or you get to the end and realize it's not there. 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 ints. 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 four 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 actually the iterator variable (in fact, all iteration is handled under the hood and we couldn't access that information even if we wanted to). The `i` is actually the value of the element in the array at the current index. This loop control expression is read as "For every element in `testArray`, let's consider `i` as the actual value in each spot."

---

## 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<sup>1</sup>

#### 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 ints that was size 5 and you used copyOf():

```
int[] arrayOne = {1, 3, 5, 3, 6};
int[] arrayTwo = 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()

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:

---

<sup>1</sup> 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>

```
// 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<sup>2</sup>

## 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.

## 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
```

<sup>2</sup> 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.E

Same thing as `Math.PI`, but 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 returns 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 two numbers - each a `double` - so if you pass in two variables of type `int`, the result will still be a `double`.

```
Math.pow(3,5); // 243.0
```

## **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 you can 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<sup>3</sup>

## 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 *pseudorandom* numbers to be as close as we can get to genuinely random.

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.

---

<sup>3</sup> 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>



## nextBoolean()

A call to this will return a boolean value that is either true or false. It's 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();
```

```
int randomInt;
randomInt = random.nextInt(50);
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. It's 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<sup>4</sup>

### 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.

### 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's 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) {
```

---

<sup>4</sup> 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>

```

    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 next token. 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 with 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):

38  
↑

```
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 its job.

Which it does. So the scanner is now after the 38.

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();
```

38  
↑  
Katie

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.

38  
Katie  


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();
```

38  
Katie  

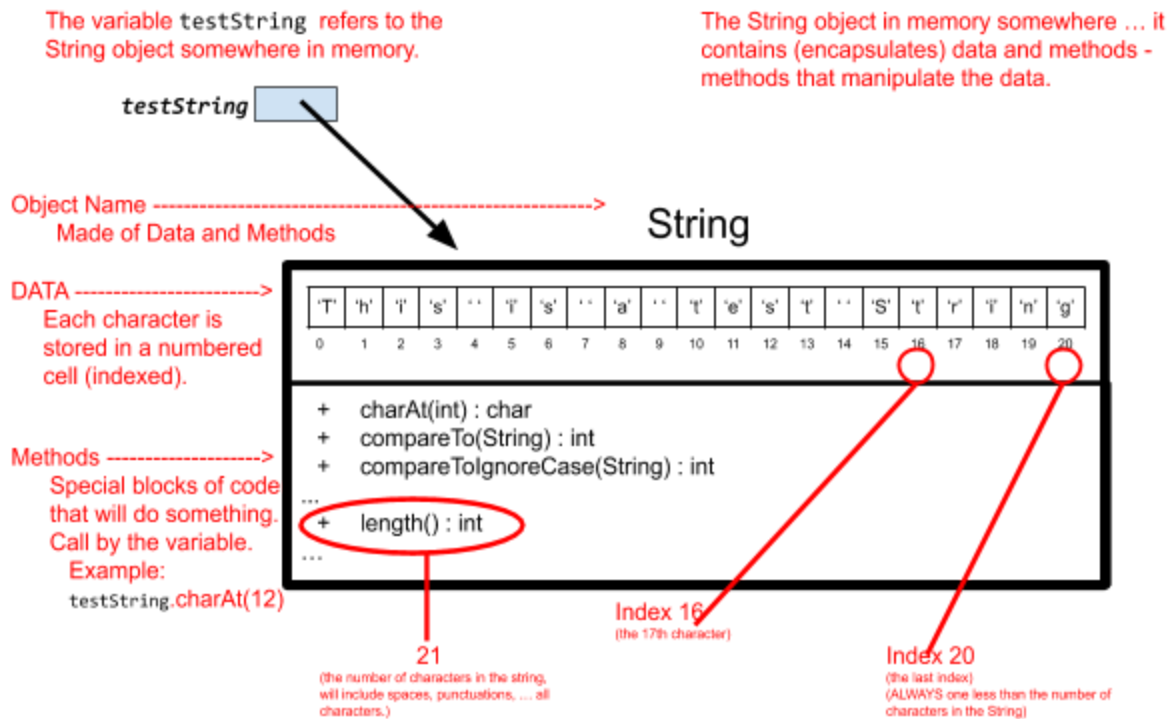

In this case, the input has now been read and stored. So all is good with the world. For the record, here is the complete code to scan in a number and then a `String`. Note that if you read in a `String`, there is no need to go to the next line before reading in a number - the call to read the `String` - `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<sup>5</sup>

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";
```



## 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 indexes start 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
```

<sup>5</sup> 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>



## 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 character 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 also 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.

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, it returns an `int` that is the index of the first occurrence of that `char`.

```
String testString = "In a galaxy far away...";
System.out.println(testString.indexOf('a')); // 3
System.out.println(testString.indexOf('q')); // -1
```

NOTE: this method actually does not take in a character - it takes in an integer. But most of the time, we pass in a character and the compiler will convert it to an integer.

### **indexOf(String)**

Returns the position (zero-indexed!) of the first occurrence of the String that is passed in. If the String that is passed in does not appear in the original String, a value of -1 is returned. Otherwise, it returns an int that is the index in the original String of the first character of the String that is passed in.

```
String testString = "In a galaxy far away...";
System.out.println(testString.indexOf("galaxy")); // 5
System.out.println(testString.indexOf("a")); // 3
```

## **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 of the String. 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.

<code>abstract</code>	<code>double</code>	<code>instanceof</code>	<code>strictfp</code>
<code>assert</code>	<code>else</code>	<code>int</code>	<code>super</code>
<code>boolean</code>	<code>enum</code>	<code>interface</code>	<code>switch</code>
<code>break</code>	<code>extends</code>	<code>long</code>	<code>synchronized</code>
<code>byte</code>	<code>false</code>	<code>native</code>	<code>this</code>
<code>case</code>	<code>final</code>	<code>new</code>	<code>throw</code>
<code>catch</code>	<code>finally</code>	<code>package</code>	<code>throws</code>
<code>char</code>	<code>float</code>	<code>private</code>	<code>transient</code>
<code>class</code>	<code>for</code>	<code>protected</code>	<code>true</code>
<code>const</code>	<code>goto</code>	<code>public</code>	<code>try</code>
<code>continue</code>	<code>if</code>	<code>return</code>	<code>void</code>
<code>default</code>	<code>implements</code>	<code>short</code>	<code>volatile</code>
<code>do</code>	<code>import</code>	<code>static</code>	<code>while</code>

# 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 underlines 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 it has 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*

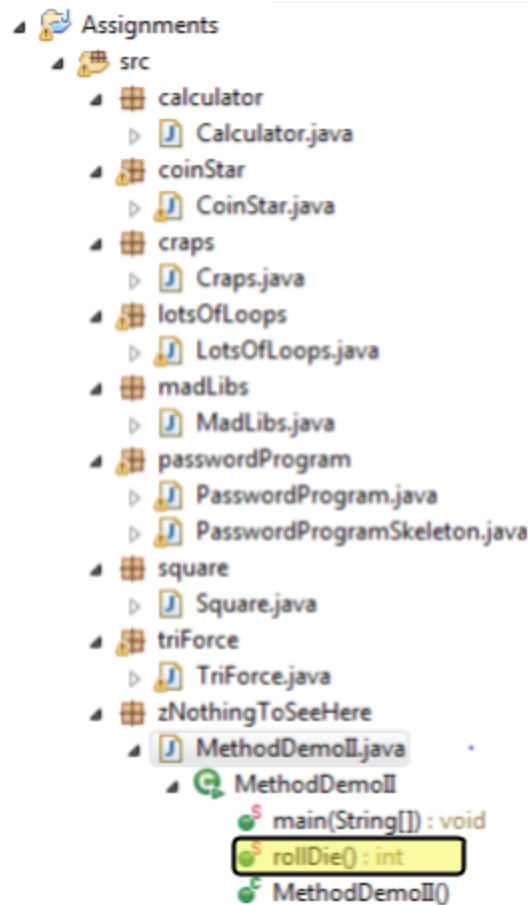
The Java Virtual Machine is what runs on your computer and allows you to execute Java programs. It takes in a `.class` file (bytecode) and converts it into 1s and 0s. The JVM is specific to the platform it is running on (MacOS, Windows, mobile devices).

## *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).

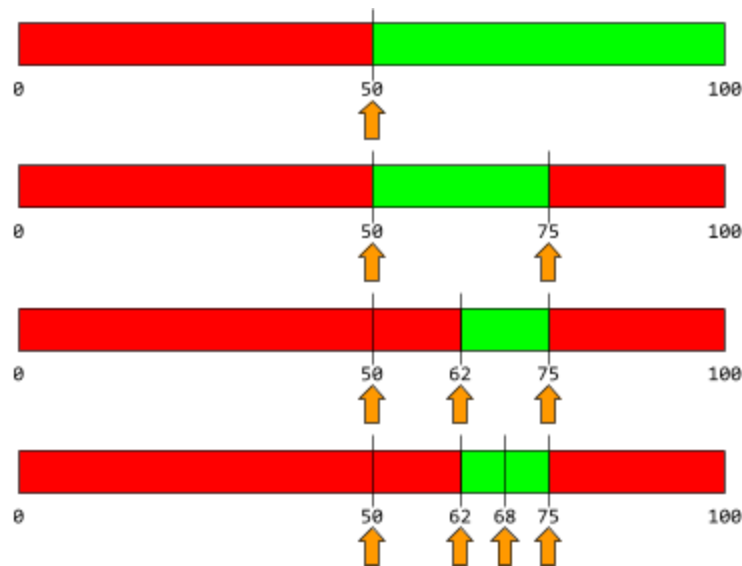
## Binary Search

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  $\log_2(n)$ . 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  $\log_2(n)$ ). 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)); // 2.0
System.out.println((double)8 / 3); // 2.6666666666666665
```

## Class

Often associated with an object, a class is a template for creating an 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 life into it - 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 programmers 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.println("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 when 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
BackWhack

System.out.println("I'm not \"the norm\" - Chris Farley");
System.out.println("This will\nbe a new line");
System.out.println("This is\t a 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 camelCase 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 the current statement right away. So interpreting translates the statements individually while executing 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, a method name, and parameters. Here's a great example:

```
public double area (int width, int height) {  
    // code to compute the 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 the 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 types (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, say 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 its own personal properties. In the Automobile example, the Automobile class specifies that every object of that class (that is, every automobile that drives around) will have its 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 in the `String` class. The `String` class 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:

<code>abstract</code>	<code>double</code>	<code>instanceof</code>	<code>strictfp</code>
<code>assert</code>	<code>else</code>	<code>int</code>	<code>super</code>
<code>boolean</code>	<code>enum</code>	<code>interface</code>	<code>switch</code>
<code>break</code>	<code>extends</code>	<code>long</code>	<code>synchronized</code>
<code>byte</code>	<code>false</code>	<code>native</code>	<code>this</code>
<code>case</code>	<code>final</code>	<code>new</code>	<code>throw</code>
<code>catch</code>	<code>finally</code>	<code>package</code>	<code>throws</code>
<code>char</code>	<code>float</code>	<code>private</code>	<code>transient</code>
<code>class</code>	<code>for</code>	<code>protected</code>	<code>true</code>
<code>const</code>	<code>goto</code>	<code>public</code>	<code>try</code>
<code>continue</code>	<code>if</code>	<code>return</code>	<code>void</code>
<code>default</code>	<code>implements</code>	<code>short</code>	<code>volatile</code>
<code>do</code>	<code>import</code>	<code>static</code>	<code>while</code>

## 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 notion 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 everything 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: FAMOUS ERRORS

## FAMOUS ERRORS

The following are some of the most popular errors that programmers experience. 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 but 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 Message:

- **ArithmeticException: / by zero**

Way to Fix/Avoid:

- You probably should just be diligent when pressing the / key while 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 hopefully won't experience it too often.

Code Snippet:

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

Example of Error Message:

- This is a logic error, so you won't get an error message.
- Most likely, you'll notice that the loop or `if`-statement doesn't function right (for example, the code in the loop will always execute exactly 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 Message:

- Because this is a logic error, you will not generate an error message.
- The output will not be what you expect - in this example, the programmer wanted to retrieve the first letter of the name, so 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;  
    int 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 §2.5 regarding RandomNumbers

Example of Error Message:

- No error message 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.
- This error usually happens when using if-statements.

Code Snippet:

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

Example of Error Message:

- 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(). 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 Message:

- 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 likely 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 depending on the scope).

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.
// 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. Therefore 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 Message:

- 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!

# APPENDIX E: 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 zeros) 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 way 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 compiler 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 Easy to read	A lot of <code>//</code> (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 <code>*</code> in each line, it could 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
//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
        //that 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 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 2.3

- #1 int, double, and String
- #2 A. int

- B. String
  - C. double
  - D. int
  - E. double
- #3
- A. Both (Initialization statement)
  - B. Assignment statement
  - C. Declaration statement
  - D. Assignment statement
  - E. Both (Initialization statement)
  - F. Declaration statement
- #4
- ```
int numClasses;
```
- #5
- A. int age;
  - B. String petName;
  - C. double total;
  - D. int numStudents;
  - E. double numSiblings;
- #6
- ```
first is 'y'
second is 'a'
last is 'y'
```
- #7
- 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!
- #8
- 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.
- #9
- `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 2.4

#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 2.5

- #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 (concatenate) 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 The values are:
- a. 5
  - b. 4
  - c. 5
  - d. 5
  - e. -4
  - f. -15
- #2 There are 23 members of a marching band. The director wants to put them in rows of 6.
- a. Use integer division (23/6) to get the number of rows; this is the number of *full rows* of marchers. Mod 23 by 6 and you'll get the number of band members that will NOT be in a row (they are left over!)
  - b. 23/6
  - c. 23%6
  - d. 44/6 and 44%6
  - e. 

```
numBandMembers / 6 // Number of rows  
numBandMembers % 6 // Number of leftover
```
  - f. 

```
numBandMembers / numOfPlayersPerRow // Number of rows  
numBandMembers % numOfPlayersPerRow // Number of leftover
```
- #3 Answers will vary

### Section 3.2

- #1 Because of the way floating-point values work, even small numbers may cause a lack of precision.

### Section 3.3

- #1 The answers are:
  - a. 5.0
  - b. 2.5
  - c. 13.0
  - d. 3.0
- #2 It will cause a Type Mismatch Exception, because the calculated answer (7.0) is a double value that is attempting to be saved to an int variable.

### Section 3.4

- #1 Correctly declared constants: b (NUM\_OF\_PEOPLE) and d (TAX\_RATE)
- #2 MIN\_SALARY is declared as a constant variable. Therefore its value can not be reassigned anywhere in the code. The second line of code will generate an error.
- #3 There are a few reasons why a programmer may want to use a constant variable in their code. Declaring a variable as a constant guarantees that it can't be accidentally reassigned later in the code. If a constant variable is declared and assigned, and used throughout the program, but then needs to have its value updated, the value only needs to be set once, when the variable is originally declared. All references to the constant variable will then use that value. Due to the ALL\_CAPS (snake\_case) naming convention, constant variables are easy to spot when looking at code.

### 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

#1 The answers are:

- a. -2.0
- b. 20
- c. 41.3
- d. 4.0

#2

- a. (5 + 3.0)
- b. (double)22
- c. 56 / 5

### Section 3.7

#1 The two lines of code accomplish the same task: add 4 to the current value of the variable x. The second line (with the += ) is a shorter/neater way to code it.

#2 The final value of a is 10.

## CHAPTER 4

### Section 4.3

#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. There are 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 can manage many characters in sequence.

#4 The first line of code forces 3 and 4 to be considered Strings, 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 5

### Section 5.1

- #1 A. is equal to (check, not assign)  
B. not equal to  
C. less than  
D. less than or equal to  
E. greater than  
F. greater than or equal to
- #2 A. false  
B. true  
C. true  
D. false

### Section 5.2

- #1 A. NO  
B. YES  
C. YES  
D. NO
- #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 6

### Section 6.1

- #1 The main difference is that code in multiple if-statements may execute (or perhaps nothing will be executed). In an if-else situation, *exactly one* of the branches will fire.

In Code A, every print statement will execute because each if-statement evaluates to true for the value of x.

In Code B, only the first print statement will run. Due to the elses, the other conditionals will not be evaluated (and the code in the final else will not run either).

- #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 semicolon 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 semicolon after the curly braces - this will cause a syntax error when compiled.

## Section 6.2

- #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`
  
- #4 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 6.3

- #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 6.4

- #1 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!
  
- #2 The code will have a compilation error at the `System.out.println` line because the variable `result` can not be found (it was declared inside of the `if-` statements). Here is one possible

solution:

```
int num = 17;
String result = "";
if (num > 0) {
    result = "positive";
} else {
    result = "non-positive";
}
System.out.println("The number is " + result);
```

- #3 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. Note: loops are covered in Chapter 8.

## CHAPTER 7

### Section 7.1

- #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 7.2

- #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 7.3

- #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 7.4

- #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. Since an integer value can be saved to a  
// double, the first parameter type could technically be double.
```

## CHAPTER 8

### Section 8.2

- #1 Loops are useful in any program that needs to repeat steps. For instance, asking the user to guess a number until they get it. Or a robot that is programmed to climb stairs until it gets to the top.
- #2 YES! Indenting is *always* important! While it may be true that the program will run without indents, the code is infuriating to read when not indented! There are other programming languages when indenting is necessary for proper compilation, so you should get in the habit of being fastidious with your indenting.
- #3 Interestingly, if you fail to use brackets, you *might* be okay (though only a monster would not use brackets). If there is only *one* line of code in the body of the loop, you do not use brackets (but you should). If there are multiple lines of code and you don't use brackets, the loop will only iterate the first line over and over and over and over and over. This leads to logic errors. So, just use the dang brackets.
- #4 

```
0  
1  
2
```

- #5 The variable `b` never changes! So it will always have the value of `10`, and will always be greater than `0`, and so the loop will run forever! This is called an *infinite loop*--you'll read more about them in this chapter.

### Section 8.3

- #1 Infinite loops derive their name from code that iterates forever. Usually this can be avoided by more care when programming.
- #2 Control + C

### Section 8.4

- #1 The code output is:

```
0 10
2 11
4 12
6 13
8 14
10 15
12 16
14 17
16 18
18 19
```

- #2 There is a logic error in the code! There are no brackets for the lines of code in the while loop. Consequently, the computer will consider *only* the line of code directly following the while statement. In this case, the code that will be repeated is:

```
System.out.println(c + " " + d);
```

Therefore, `c` and `d` will never alter, so this code will just output the values of `c` and `d` forever.

### Section 8.5

- #1 Functionally, there is no difference between a for loop and a while loop; they can both do the same thing (that's true of a do-while loop, too). The major difference is rationale. Typically a while loop is used for a process that does not have a defined number of iterations (think of asking a user to guess a number or run until a condition is met). A for loop is usually used when code will be executed a known number of times (a countdown from 10, adding up all the numbers between 1 and 100).
- #2 The last number to be output on the screen is 20, though the last value of `i` is 21. Don't forget that when `i` is 21, that is what causes the loop to terminate.

### Section 8.7

- #1 In a loop, use of the `break` statement is usually a lazy way to end a loop. You can always avoid a `break` statement with better conditionals. Other than `switch` statements, there is never a case when you will need a `break`.

- #2 The break statement is in the if statement because that's when the check happens (to see if the password was entered correctly). The break will kick out of the loop despite being in an if statement.

## CHAPTER 9

### Section 9.1

- #1 There are thousands of examples, but some of the more popular examples of arrays include:
- Rosters for sports teams
  - Phonebooks
  - Inventories in video games
  - Cataloging systems in libraries
- #2 The diagram below is a sample - your values in each box may be different, but the structure (name, size) should be the same as in the diagram.

double[] grades

87.4	76.8	98.1	62.0	83.9	92.2	59.4	71.3	99.7	84.6	68.2	87.9
0	1	2	3	4	5	6	7	8	9	10	11

### Section 9.2

#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 9.3

- #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:

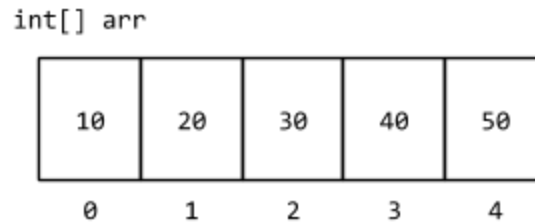
```
int[] arr = {1, 3, 5, 9, 11};
System.out.println(arr[arr[1]]);
```

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 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 9.4

#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.