Skimpy OOP: Introduction to Object-Oriented Programming Using Java

Skimpy Contents

Skimpy Preface: An Excuse for the Inadequacies Within . . . 3
But why are we stuck reading this eBook? . . . 4
And why are we using Java instead of ***my*** favorite language? . . . 4
How to Use This Book . . . 5
Related Resources . . . 5
Source code files: . . . 5
0000 - Getting Started . . . 5
What's the Point? . . . 6
Install the Java Development Kit . . . 6
Install an IDE . . . 7
Check Yourself Before You Wreck Yourself (on the assignments) . . . 8
1. 0001 - Computers and Coding . . . 8
What's the Point? . . . 8
1.1. Programming Languages . . . 10
1.2. Programming Paradigms . . . 11
1.3. Software Development Process . . . 12
1.4. First Java Program . . . 13
1.5. Basic Output . . . 14
1.6. Code Comments . . . 15
Check Yourself Before You Wreck Yourself (on the assignments) . . . 17
2. 0010 - Variables . . . 17
What's the Point? . . . 17
2.1. Java Identifiers . . . 20
2.2. Numeric Data Types . . . 20
2.3. `boolean` Data Type (and Boolean Logic) . . . 22
2.4. `char` Data Type . . . 22
2.5. Strings . . . 23
2.6. Declaring Constants . . . 24
2.7. Outputting Variable (and Constant) Values . . . 24
2.8. Math Calculations . . . 25
2.9. Getting User Input . . . 27
Check Yourself Before You Wreck Yourself (on the assignments) . . . 28
3. 0011 - Methods . . . 28
What's the Point? . . . 29

3.1. Defining Our Own Methods . . . 30
3.2. Variable Scope . . . 31
3.3. Passing Data to Methods . . . 33
3.4. Returning Values . . . 35
3.5. Overloading a Method . . . 37
3.6. Solution Walkthrough . . . 39
Check Yourself Before You Wreck Yourself (on the assignments) . . . 40
4. 0100 - Classes and Objects . . . 40
What's the Point? . . . 40
4.1. OOP Basics . . . 41
4.2. Encapsulation . . . 43
4.3. Defining and Using a Class . . . 43
4.4. Constructors . . . 51
4.5. `static` Constants and Methods . . . 54
Check Yourself Before You Wreck Yourself (on the assignments) . . . 55
5. 0101 - Decisions . . . 55
What's the Point? . . . 55
5.1. Boolean Expressions . . . 56
5.2. `if` Statements . . . 58
5.3. Adding an `else` Block . . . 58
5.4. The `if-else if` Structure . . . 59
5.5. Nested `if-else` Statements . . . 61
5.6. Using Logical Operators . . . 62
5.7. `switch` Statements . . . 64
5.8. Solution Walkthrough . . . 65
Check Yourself Before You Wreck Yourself (on the assignments) . . . 66
6. 0110 - Loops . . . 66
What's the Point? . . . 66
6.1. `while` Loops . . . 67
6.2. `do-while` Loops . . . 69
6.3. Input Validation with Loops . . . 70
6.4. `for` Loops . . . 71
6.5. OPTIONAL: `break` and `continue` Statements . . . 71
6.6. A Word About Nested Loops . . . 72
6.7. JUST FOR FUN: Recursion . . . 72
6.8. Solution Walkthroughs . . . 72
Check Yourself Before You Wreck Yourself (on the assignments) . . . 73
7. 0111 - Debugging and Generative AI . . . 73
What's the Point? . . . 74
7.1. Debugging . . . 74
7.2. Types of Errors . . . 74

7.3. Debugging Tools in Our IDE . . . 75
7.4. Generative AI in Coding . . . 76
Check Yourself Before You Wreck Yourself (on the assignments) . . . 79
8. 1000 - Arrays . . . 79
What's the Point? . . . 79
8.1. Arrays and Indexes . . . 79
8.2. Defining and Using Arrays . . . 80
8.3. Traversing Arrays . . . 81
8.4. Putting Objects in Arrays . . . 82
8.5. "For-Each" Loops . . . 83
Check Yourself Before You Wreck Yourself (on the assignments) . . . 83
9. 1001 - Inheritance . . . 84
What's the Point? . . . 84
9.1. Inheritance Overview . . . 84
9.2. Implementing Inheritance . . . 85
9.3. Inheritance Hierarchies and Overriding . . . 87
9.4. Constructors & Inheritance . . . 90
9.5. Introduction to Polymorphism . . . 91
Check Yourself Before You Wreck Yourself (on the assignments) . . . 92
10. 1010 - Graphical User Interfaces with Swing . . . 92
What's the Point? . . . 92
10.1. Graphical User Interfaces . . . 92
10.2. The Swing Library . . . 93
10.3. Event-Driven Programming with Swing . . . 94
10.4. Processing User Input with Swing . . . 95
10.5. Widgets . . . 95
10.6. GUI Layouts . . . 97
Check Yourself Before You Wreck Yourself (on the assignments) . . . 98
11. Stuff That's Tacked on the End . . . 98
Sample answers to *Check Yourself Before You Wreck Yourself (on the assignments)* questions . . . 98

[Skimpy OOP Cover] | *Skimpy_OOP_Cover.png*

Image created with the assistance of DALL-E 2

Skimpy Preface: An Excuse for the Inadequacies Within

This isn't a real textbook. It's not *imaginary* or anything, it's just missing most of the stuff a real textbook includes. It's, well, skimpy.

This content is intended to be a companion to CIS150AB, which is an introduction to object-oriented

programming. Since you'll be attending (or viewing recordings of) lectures, and you'll have access to additional content in Canvas (our LMS), this text doesn't try to cover everything; it tries to give you the basic information you'll need in order to get the most out of that other content.

It's also skimpy because the purpose of this particular course, at least from my perspective, is not to give students a comprehensive look at a programming language. It's an introductory course that's taken by a variety of students. If you're a coding student—majoring in *Programming and Systems Analysis* or *Web Design and Development*, for example—you'll take language-specific courses that will get much more detailed, so our goal here is to give you a foundation for that. If you're majoring in something else, you might never write another line of code after the semester ends—I want you to learn about programming and software development so you can communicate with the coders in your workplace and have a general understanding of how an idea becomes a program or app.

It's also possible that I just don't know very much. You know, a skimpy brain.

But why are we stuck reading this eBook?

The only reason I wrote this is because real textbooks are expensive. As I started to record more and more videos to supplement the real textbook I was using, students had less and less reason to actually read it. But I can't cover everything we need in my videos, so I started writing content to fill in details I couldn't get to.

To be clear, this **isn't** skimpy just because students don't bother to read textbooks that are dense and detailed. That sometimes seems true, but you're different; I can tell just by looking at you! But I acknowledge that many students prefer video content and in-person instruction rather than reading, so this is my attempt to meet those students halfway.

This eBook might not be very good or very detailed, but it **is** very free.

And why are we using Java instead of *my* favorite language?

Looking at you, Python nerds…

Okay, there are a few reasons I use Java to teach OOP:

- It's cross-platform, so my students can use Windows, macOS, or Linux machines and I can grade the work without hassle on whatever machine I'm currently using.
- I like the way Java implements OOP, which I think is "stronger" than in Python, for example.
- Java is *strongly, statically typed*, meaning we have to declare data types when we make our variables and that type can't change. I believe this "stricter" approach to data types is better for beginners.
- Since it uses C syntax, Java skills transfer well to many other languages.
- At Estrella Mountain Community College, we have other CIS classes that teach C# (my favorite OOP language) and Python. I want students using something different for this course so that they get exposed to multiple languages before they graduate.

If I'm doing this right, what you're mostly learning here is a good foundation of OOP (and general programming) skills and concepts. We happen to be using Java to do that, but these skills transfer to pretty much any other language you want to learn.

How to Use This Book

Before starting a chapter, review the relevant information on the "overview" page in Canvas. Then, start reading the chapter in this eBook.

- When you see **Time to Watch**, just watch the embedded (or hyperlinked) YouTube video. Quizzes in Canvas may include questions about the video content.
- After the video, continue reading the chapter.
- Each chapter begins with **What's the Point?**, which gives you an idea of what you should be able to do after reading the chapter.
- At the end of each chapter, **Check Yourself Before You Wreck Yourself (on the assignments)** provides a few questions to help you review the material.
- You can find sample answers to these questions at the very end of the eBook, in the Stuff That's Tacked on the End section.
- A few sections of the book are identified as **optional**. These sections are not required for the course assignments or quizzes, but they might be interesting or helpful to you.

Related Resources

In general, this book includes resources within each chapter. The following links collect those resources so you can find them when necessary.

GitHub landing page: https://timmcmichael.github.io

YouTube channel: @ProfTimEMCC

Course YouTube playlist: Object-Oriented Progamming Fundamentals (CIS150AB)

Source code files:

The source code files shown in the eBook and videos are available on GitHub; a link at the beginning of each chapter will take you to the appropriate folder in the repository. If you have trouble finding something (or using the GitHub site), please contact me at tim.mcmichael@estrellamountain.edu or in Canvas.

- GitHub Repository: https://github.com/timmcmichael/EMCCTimFiles

0000 - Getting Started

Help Make These Materials Better!

I am actively working to complete and revise this eBook and the accompanying videos. Please consider using the following link to provide feedback and notify me of typos, mistakes, and suggestions for either the eBook or videos:

CIS150AB Course Materials Feedback (Google Form)

What's the Point?

- Install the tools you need to create Java programs

Our goal is to learn the fundamentals of **object-oriented programming** using the **Java** programming language. You might not even know what those things mean yet, but to get started you'll need a couple of things installed on your computer.

Java Development Kit (JDK)

A collection of software, used to develop Java programs. Most importantly, it includes a **compiler**.

Integrated Development Environment (IDE)

An application used to write and test source code.

These two tools can be used on pretty much any laptop or desktop computer, even if it's not very powerful.

Heads Up!

Java development on a Chromebook or iPad is tricky and not recommended for beginners. Use a different computer if at all possible.

The Canvas course includes some options for students who don't have a computer capable of running the necessary software.

Install the Java Development Kit

I recommend installing the JDK available at https://adoptium.net, and the Oracle JDK is another great option. The linked documents below provide detailed installation instructions—but if they are outdated at the moment, there are about a gazillion web pages and Youtube videos that will show you the process. I recommend installing the JDK before installing your IDE.

- Install JDK on Windows
- Install JDK on macOS

Install an IDE

A computer program is created by writing **source code**, which is just text and can be created using any text editor. However, developing programs with something like Notepad or TextEdit—which come with Windows and macOS, respectively—means going through some potentially confusing (or at least intimidating) steps each time you want to run your program. Though IDEs do much more, the most important thing they do for beginners is give you a "run" button you can press whenever you want to test your code.

You can definitely write Java programs without using an IDE, but it's kinda like getting engaged to someone you've only ever met online: just because it turns out okay for some people doesn't mean it's a good idea for you. The only people who should write code without an IDE already know too much about programming to be wasting their time reading this book.

There are many great IDEs out there, including many free options, and ultimately it doesn't matter which one you use: they all produce the same files, and a person looking at the files can't tell what IDE you used. It's like choosing between Word and Google Docs; each has strengths and weaknesses, but you can write romantic poems about your online fiancé (or fiancée) in either one.

I primarily use Visual Studio Code ("VS Code"), so that's what you'll see in most of my videos. It's completely free and is available for Windows, macOS, and Linux. You can download it at https://code.visualstudio.com, and the videos below walk through installing VS Code and the tools it needs for Java development. I recommend installing a JDK (see previous section) prior to installing an IDE.

Heads Up!

Visual Studio **Code** is not the same as Visual Studio! There's a free version of Visual Studio (called Visual Studio Community) but it's not intended to be used with Java. You're going to want Visual Studio Code (aka "VS Code") for your Java coding.

Once you've installed the JDK above, install the IDE you are going to use to learn Java. There's nothing wrong with trying out multiple IDEs to find out which one you prefer. Other popular options include eclipse, IntelliJ IDEA, and Apache NetBeans, and they all have free versions that are great for learning Java.

But my recommendation is to install VS Code.

Time To Watch!

Installing VS Code on Windows for Java

▶ https://www.youtube.com/watch?v=Pkj6n3UVXEI *(YouTube video)*

Note: If you are on a school-issued laptop, or are having trouble with the Windows installer, you can also install from the Windows app store: Installing VS Code on Windows from Microsoft Store

Installing VS Code on macOS for Java

▶ https://www.youtube.com/watch?v=DrV5vcvIyR0 *(YouTube video)*

Once you've installed the JDK and VS Code (along with the Java extensions), you're ready to create your first program.

Check Yourself Before You Wreck Yourself (on the assignments)

Can you answer these questions?

1. What does "JDK" stand for?
2. What is an example of an Integrated Development Environment?

Sample answers provided in Stuff That's Tacked On The End.

1. 0001 - Computers and Coding

Help Make These Materials Better!

I am actively working to complete and revise this eBook and the accompanying videos. Please consider using the following link to provide feedback and notify me of typos, mistakes, and suggestions for either the eBook or videos:

CIS150AB Course Materials Feedback (Google Form)

What's the Point?

- Understand the role of programming languages
- Get to know a little about Object-Oriented Programming (OOP)
- Use `print()` and `println()` to produce text output

Source code examples from this chapter and associated videos are available on GitHub.

A computer is basically a device that executes a set of commands—and does so very quickly. Because the guts of a computer (a CPU) are kinda like a bunch of light switches—and I mean a **BUNCH** of light switches—it can only deal with zeros and ones: a switch that is off is "zero," and a switch that's on is "one."

Never before has the concept of a modern computer been so recklessly simplified. I feel shame.

All of the information a computer handles ultimately has to be represented by some combination of 1s and 0s, which we call a **binary** (or "base-two") number system. The chapters of this eBook are numbered using binary to show basic examples of the numbering system: 0001 is, well, 1 in decimal. And 0100 is 4 in decimal. We won't get into binary numbers, but it's not very complicated and is kind of interesting if you're a nerd.

Learn More

The amazing code.org project has a playlist of great, short videos on how computers work—including binary numbers—if you want to know a little more without going too crazy:

How Computers Work

Imagine that your BFF is in your object-oriented programming class, but that cruel professor won't let you sit together. When the professor turns his back, you can use your fingers to send a quick message to your BFF. Holding up your index finger might mean, "meet me in the library after class;" two fingers could mean, "send me a copy of your homework." As long as you agree on what each number means, you could pass along commands or information—it would be like your own little language.

If you're a sports fan, I have a better analogy. A quarterback and a coach could have a language based completely on numbers. The coach tells the QB "22" and the QB knows which play to run, because they've agreed on what instruction (play) the number 22 means—and there are football teams that do exactly this.

Sending that number to the quarterback as a binary number would be a lot less efficient because the coach would have to say "10110" (which is the binary representation of twenty two) and the QB would have to know binary numbers,or have them written on his wrist-band thing as a cheatsheet.

Want to take the analogy one step further? A player standing on the sideline with their helmet on could represent a 1, and a player with their helmet off could represent a 0. Now the coach could just line up five players and tell the 2nd and 5th ones to take their helmets off. The QB could look over, see the pattern of helmets (10110) and run play 22.

Okay, that would be the nerdiest football team in history. And how is that relevant? Let's see…

Remember that a computer can execute commands super fast. Each command, or **instruction** (like a football play), is represented by a number—a binary number, since that's the number system the computer uses. The collection of instructions the computer understands is called **machine language**.

Interesting!

Just like people in different cultures use different languages, different CPU types use different

languages. There are lots of different machine languages out there.

You could give the computer a set of instructions (aka, a program) if you just looked up the binary number for each instruction you wanted to use. Of course, even a simple program requires a lot of instructions, so you're going to be looking up a **lot** of stuff.

Few people actually *want* to do that, so the rest of us use programming languages instead.

1.1. Programming Languages

A programming language is something that's easier for humans to use than machine language, but is capable of being accurately translated to machine language.

There are many, many programming languages. Seriously, there are a lot.

The instructions you write using a programming language are called **source code**. Translating source code file to a machine language file that can be executed is called **compiling**, and is done by an application called a compiler. When the computer runs the program, it's using the machine language translation created by the compiler. Clicking on the icon to open Microsoft Word runs a file that's been compiled from source code.

Time To Watch!

Coding and Compiling

▶ https://www.youtube.com/watch?v=yR939VDXPaM *(YouTube video)*

Some programming languages don't get translated ahead of time—they get translated "on the fly," as the program is running. That's called **interpreting** instead of compiling, but it's conceptually the same thing.

The C programming language is an example of a compiled language, and Python is an example of an interpreted language.

Java is an interesting case, because it's both compiled **and** interpreted. The JDK compiles our source code into an intermediate language called **bytecode**. To execute the program, you use the Java Runtime Environment to interpret that bytecode file as it executes.

Bytecode is very close to machine language. In fact, it's basically a "virtual machine language"--a machine language for a CPU that doesn't actually exist. This approach allows Java to be cross-platform (runnable on many different machines). A compiled machine language file only works on a computer that "speaks" the same machine language, but if we compile our program to a bytecode file, it'll run on any computer that has the Java Runtime Environment installed—and that will interpret the bytecode as it executes. This "write once, run anywhere" ability contributes greatly to Java's popularity.

1.2. Programming Paradigms

A paradigm is a model or example, or maybe better described as a way of seeing things—which is helpful to think about, since we can all have different ways of seeing how a problem could be solved.

Imagine that your crazy uncle wants to pay you to go through the junk in his garage and organize his giant collection of vinyl records. Or imagine it's a collection of books, if that's what you prefer. How would you organize the collection? You could group them by artist and then arrange them alphabetically, from ABBA to Warren Zevon. Or you could group them by genres (keeping disco separate from rock \'n' roll), or just put them in the order they were released.

There are tons of different ways to do it. None of them are really right or wrong, it just depends on what you prefer—or maybe on how you plan to find things later. Coding is similar, in that we all have different ways we imagine organizing a program or solving a problem. That's kinda the idea of a programming paradigm: how do you think about the program you're creating? How do you picture it being organized?

Just like programming languages, paradigms are numerous. In fact, some languages were created specifically to work well with a certain paradigm, and that's one reason there are so many languages. But when someone is going to learn to code, there are often two paradigms we consider:

- Procedural Programming
- Object-Oriented Programming (OOP)

They have a lot in common, and in fact, OOP is actually a type of procedural programming. Programmers love to argue why one approach is better than the other, in the same way some of my friends might argue about Ford trucks vs. Chevy trucks. And just like Ford vs. Chevy, there's nothing inherently better about one paradigm or the other—sorry friends, but Ford and Chevy trucks are basically the same. Growing up, my own first coding experiences were exclusively with procedural programming, and I stayed in that world until I began teaching Java—now I very much prefer OOP, both for my own programming and for teaching beginning coders. But that really is a personal preference, and I wouldn't argue that OOP is *better* than procedural programming.

Well, I wouldn't argue *much*.

1.2.1. Comparing Procedural Programming and OOP

In procedural programming, we break up a program into the tasks we need to complete. Each individual task is handled by a set of statements that we call a **procedure**. If we need to calculate a student's overall course grade, for example, we write a procedure to do it.

The data for our program—the stuff we're keeping track of—is stored somewhere else. If a procedure needs something to complete its task, that data is sent to the procedure, which then sends back a result. In short, information is kept separate from the code that uses it and is passed back and forth as necessary.

In Figure 1, the *main procedure* passes the values 5 and 4 to the *adding procedure*, which uses that information to calculate the sum 9 and return it to the *main procedure*. The *subtracting procedure*

also requires two numbers. `10` and `3` are passed to that procedure, which returns `7` to the *main procedure*.

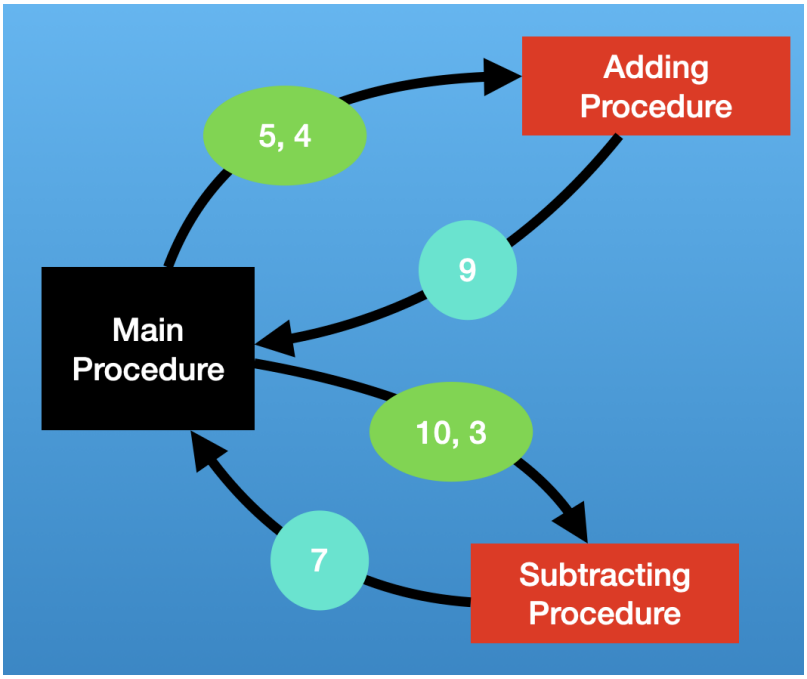


Figure 1. Passing data in procedural programming

Many programming languages (including Python) refer to procedures as **functions**, and the term **subroutines** is also used in some cases; if you've done some programming in any language, you've almost certainly used procedures in some form.

In OOP, the focus shifts from procedures to **objects**, which are programming elements that bundle data with the procedures that use it, instead of keeping them separate. We'll explore the nature of objects in great detail going forward, so we won't worry about describing it too much for now.

I have my own biases about the topic. I tend to think OOP lends itself especially well to things like graphical user interfaces (GUIs) and games, as well as large projects developed by teams of programmers.

The bottom line is that, when faced with a programming task or project, my brain immediately starts thinking in terms of the *objects* the program will need. But that's probably just a matter of habit; as I mentioned, I spent much of my life (including all of the time I spent as a professional programmer) seeing programs as a bunch of procedures.

It's important that you're aware of procedural programming, and the code we'll write in the next couple of chapters is really written from a procedural perspective, but our overall focus in this book and course is really on OOP.

1.3. Software Development Process

Learning to write code means creating a lot of programs—mostly small, straightforward programs at first. Remember those awful word problems where a train leaves Chicago traveling 40 mph, and another train leaves Denver at 35 mph? That kind of stuff; but in my course, we don't get too caught up in the math part of it. But we care **a lot** about understanding the requirements of a program and implementing it successfully.

As our programs become bigger and more complex, we'll need to work within a deliberate design and implementation process in order to keep ourselves organized and focused. Even the smaller programs we'll develop while learning the basics will benefit from a thoughtful approach beyond just opening a new file and starting to type. It ensures that we use our time efficiently. And when we are faced with solving a programming problem that really intimidates us, the process will help make the task more approachable.

For big or small projects, a good general approach to software development is:

Analysis

Identify the goals and scope of the program. As a rule, keep it small and focused—we can always add features later. **Ask yourself, *What does this program need to do?***

Testing Plan

Determine how the final program will be tested. The testing plan will be useful, but most importantly, taking the time to establish a specific testing plan ensures that you thoroughly understand the program before you begin writing code. If you don't know how the program will work, you're not yet ready to begin coding. **Ask yourself, *How will I make sure the program works correctly?***

Implementation

Write and test the code. We say that this is an *iterative* (or "repeating") process, meaning you'll write and test one small piece, staying with it until you know it's good. Then you'll move on to the next piece and repeat. **Ask yourself, *What code do I need in order to get the next part of the program working?***

Revise or maintain

If our needs or program requirements change, we'll need to go back to the first step and begin planning the next version. If not, we'll need to monitor that the program continues to perform as expected over time. **Ask yourself, *What's next for this program?***

We'll flesh out this process as we go—and as our programs become more advanced.

1.4. First Java Program

Enough of that, let's write some code!

One of the (valid) criticisms of Java as a choice for beginners is that it's a little complicated to create our first program. In Python, we just open a file and write our first command; in recent versions, C# has added that ability as well. But Java puts OOP front and center, and we can't start writing statements until we first define a class.

Time To Watch!

Java File Structure and First Program

▶ https://www.youtube.com/watch?v=zYDdJzs24rs *(YouTube video)*

File from video:

- Completed code: `HelloWorld.java`

Take a look at the code for a basic "Hello World" program; we'll learn what all of these pieces are as we go, but we should at least identify them now.

`HelloWorld.java` - *Hello World program in Java*

```
public class HelloWorld { ①

    public static void main(String[] args) { ②

        System.out.println("Hello World!"); ③

    } ④

} ⑤
```

Here are the parts of the program:

① Class declaration and start of a code block. This is a publicly accessible class called `HelloWorld`.

② `main()` method declaration and start of a code block. `main()` is where a Java program starts running.

③ `println()` statement to output the message.

④ End of the `main()` method code block, as indicated by the indenting and closing curly brace.

⑤ End of the class code block.

We'll learn about all of these components as we go. But for now, we're off and running!

1.5. Basic Output

The first programs we create in Java are *console* programs—they are text-based programs that can't really display any graphics. To start with, we'll use two basic ways to output text to the console: `System.out.print()` and `System.out.println()` statements. `print()` outputs whatever text is in the parentheses, and we'll need to put that text in quotation marks:

```
System.out.print("Mick Jagger");
```

This line of code outputs **Mick Jagger** to the console window. After `print()` outputs the text in parentheses, the cursor remains at the end of the output. This is just like if we type something in a word processor but don't hit enter; the next time we start typing, the characters resume on the same line. In the same way, the next output statement will continue on the same line in the console.

A `println()` statement works exactly the same way, but it advances the cursor to the next line when it's finished. Basically, it hits *enter*, and the next output statement will begin on a new line.

Interesting!

println() works by outputting the text inside the parentheses and then outputting a special character called a *newline*. The newline isn't visible, but it moves the cursor to the next line.

To understand the difference between `print()` and `println()`, consider this program.

`OutputExample.java` - *Simple console output in Java*

```
public class OutputExample {

    public static void main(String[] args) {
        System.out.print("As the Rolling Stones might say,");
        System.out.println("you can't always");
        System.out.println("get what you want.");
    }

}
```

The program produces the following output:

```
As the Rolling Stones might say,you can't always
get what you want
```

After the `print()` statement executes, the cursor is still sitting right after the comma following **say**, so when the next line of code outputs ***you can't always***, that output just gets jammed onto the end. Notice that it doesn't even add a space; if we want a space there, we have to include it within our quotation marks.

Because ***you can't always*** is in a `println()` statement, the cursor advances and ***get what you want*** is on a new line.

We'll use `print()` and `println()` in every Java program we write for quite a while, so it's important to take time to experiment with them on our own to make sure we understand how they work.

1.6. Code Comments

The Java compiler goes through our source code file line by line, translating all of the code into something that we can execute (unless it finds something it doesn't understand, which causes it to stop and output an error message). If there's something in our source code we don't want the compiler to process, we can identify it as a *code comment* and the compiler will ignore it. Code comments are generally used to provide information for any humans who might be looking at the code. And since it's ignored by the compiler, it can be written however we want; so our code comments should be written in plain human language (English, if we're submitting it to me). To indicate a comment, use two slashes:

```
// This is a comment!
```

Once the compiler sees two slashes, it just ignores the rest of the line. We can add a comment onto the end of a line of code:

Inline comment placed at the end of a line of code

```
System.out.println("Hello World"); // this outputs text to the console
```

The `println()` statement still gets processed and will execute when we run our program, but everything after the slashes gets ignored.

To make a comment that takes up multiple lines, start the comment block with `/*` (that's a slash and an asterisk) and end it with `*/` (asterisk and a slash). When the compiler sees `/*` it will ignore everything until it finds `*/`, and then it will resume processing as usual.

Multi-line comment block

```
/*
This program shows the difference between print() and println().
It is referring to an old Rolling Stones song.

Everything in this comment block will be ignored.
*/
public class OutputExample {

    public static void main(String[] args) {
        System.out.print("As the Rolling Stones might say,");
        System.out.println("you can't always");
        System.out.println("get what you want.");
    }

}
```

In general, code comments are used to explain or provide context for our code. Programming often involves going back to old code to make updates or corrections. Maybe it's been a long time and we might not remember what the code is supposed to do, or maybe it's someone different looking at the code and trying to figure it out. So code comments should be descriptive, especially when code might be confusing.

Code comments don't cost anything, so use lots of them!

We often add a multiline comment block at the top of a file to provide information about the overall program or class.

1.6.1. "Commenting Out" Code

Coding is all about trial and error, and programmers spend a lot of time writing code in different ways until they get it working the way they want. In a process like that, it's not unusual to delete something only to regret it and wish we could have that old code back.

Code commenting gives us a life hack to help prevent that regret. Instead of deleting code that's not working the way we want, we can just mark it as a comment. As far as the compiler is concerned, we've deleted the code. But if we want to see or use the code again down the road, it's still there.

Almost all IDEs have a keyboard shortcut for commenting out code—and in most, it's the same shortcut. Highlight a section of code and press `Alt` + `/` on Windows or `⌘` + `/` on macOS, and many IDEs will add `//` at the start of each line. Highlighting a comment and pressing the shortcut again will usually "uncomment" it and remove the slashes. If that doesn't work, check the IDE's documentation to see if there's a different shortcut.

Check Yourself Before You Wreck Yourself (on the assignments)

Can you answer these questions?

1. What is the primary role of a programming language in the context of computer programming?
2. Explain the difference between compiled and interpreted programming languages. Provide an example of each.
3. Describe the basic structure of a simple Java program, such as the "Hello World" example provided in the chapter.
4. What are the key steps in the software development process as outlined in the chapter? Why is it important to follow these steps?

Sample answers provided in Stuff That's Tacked On The End.

2. 0010 - Variables

Help Make These Materials Better!

I am actively working to complete and revise this eBook and the accompanying videos. Please consider using the following link to provide feedback and notify me of typos, mistakes, and suggestions for either the eBook or videos:

CIS150AB Course Materials Feedback (Google Form)

What's the Point?

- Know how to select a data type

- Write code to declare variables and assign values
- Do some basic math

Source code examples from this chapter and associated videos are available on GitHub.

It's fun to create programs that display stuff on the screen, but that's pretty limited. To start doing things that are more interesting—and useful—we'll need to keep track of some information. In computer programming, a piece of information is called **data**.

In broad terms, there are two kinds of data:

- **constant:** A piece of data that can't change during program execution.
- **variable:** A piece of data that may change (or "vary") during program execution.

In our `Hello World` code, the letters and characters we output are *constant* data.

`HelloWorld.java`

```
public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello World!"); ①

    }

}
```

① The string literal "Hello World"

The only way to change what's output is by editing that constant—in the quotation marks—in our source code, before we run the program.

The string literal in our `Hello World` code is just one kind of constant. We'll learn about others later.

In order for our programs to be flexible and responsive, we'll need **variables**. I encourage beginning students to think of a variable like a container, such as a cup.

Figure 2. A photograph of the **actual** *mug used in lectures.*

The container can hold one thing at a time. We can put coffee in it, or we can put water in it—but we can't put both. If we start with coffee and decide we want to put water in there, the coffee gets dumped out and is gone forever. That's fine, but we better make sure we don't want the coffee anymore before we put something else in the mug.

A variable works the same way. It can hold one value, and that value can vary—remember, that's where the name comes from—but once it's changed, that old value is lost.

I guess we could mix coffee and water, but then that's really a new substance and we can't get back our plain coffee (or our water). If we want to store coffee **and** water, we can do that by using two different containers.

From a programming perspective—rather than a coffee perspective—a variable is a location in the computer's memory where we can store a value. In order to use that value, we have to know where to find it, so we give the memory location a name, technically called an **identifier**.

And the computer will need to know how much memory to set aside for that variable, which depends on the kinds of values we want to store. Storing the number 7 doesn't take very much space; storing a picture of John Elway takes a lot more space (but it's worth it). The **type** of data we can store in a variable is called its

…*drumroll*, please…

data type.

Reserving a memory location by specifying a data type and identifier is known as **declaring** a variable. Placing data in that variable is referred to as **assigning** a value to a variable.

If you have programming experience in Python or JavaScript, the idea of a data type might be new to you. In those languages, the interpreter determines the data type for you—so you don't have to specify it. It's one reason I prefer to teach beginners using Java or C#, which teaches you how data types work.

Time To Watch!

Using Variables in Java

▶ https://www.youtube.com/watch?v=At0vquefCuo *(YouTube video)*

Files from video:

- Completed code: `DeclaringVariables.java`
- Completed code: `DeclaringVariablesFinished.java`

2.1. Java Identifiers

Variable names (more properly called variable *identifiers*) are the character sequences that identify a variable. There are a few rules that determine whether or not an identifier is valid:

- Can contain letters (upper or lower case), digits (0 - 9), underscores (_), and/or dollar signs ($)
- The first character **can't** be a digit.
- Can't contain whitespaces (tab, space, etc.) or special characters (like #, @, !, etc.) except the underscore and dollar sign.
- Can't be a Java keyword or reserved word (words that are used in the Java language, like `double` or `public`)

In addition to those technical rules about identifier names, the convention in Java is to use descriptive names in "camel case" format, as described in the preceding video. Single-letter variable names, like `x` for a number, are fine in your math class, but are generally **not** okay when you're coding. The expectation in my course is that your code will align with those conventions—because that's what industry people expect—and you will lose points if you don't.

2.2. Numeric Data Types

Variables for storing numbers come in two flavors: **integers** and **floating-point** numbers. As you may remember from your math class, an integer is a whole number; that is, a number that doesn't include any decimal places or fractional values. *5* is an integer, *-824* is an integer, while *3.14* and *7 1/2* are not.

A floating-point number includes decimals, so 3.14 can be stored as a floating-point number. 7 1/2 can also be stored as a floating-point number, but only as a decimal (i.e., 7.5).

The most common numeric data types in Java are `int` for integers and `double` for floating-point numbers. You can get pretty far in programming just using those two, and in courses I teach you

won't need to use any other numeric data types—but others do exist.

Other data types for integers are `byte`, `short`, and `long`. These different types exist because they use different amounts of memory. `byte` and `short` are smaller than `int`, while `long` is larger. The impact of these different memory sizes (or memory **allocations**) is that the types can store values of different sizes. For example, a `byte` uses 8 bits of memory and can store a number between -128 and 127 (inclusive), while an `int` takes 32 bits and can store a value between -2,147,483,648 and 2,147,483,647 (inclusive).

Interesting!

This all goes back to binary numbers. An 8-bit number has 8 digits, and we use the first digit to specify whether the number is positive or negative. That leaves 7 digits, and we can make 128 different combinations of 1s and 0s in 7 digits: `0000000`, `0000001`, `0000002`, and so on, all the way up to `1111111`. Those 128 possibilities give us the `byte` range of -128 to 127 (keeping in mind that we do need one of those combinations to represent zero).

The other floating-point data type in Java is called `float`. It's called "single" in some programming languages, which helps understand where the name `double` comes from: `double` uses twice as much memory (64 bits) as a `float` or "single" (32 bits)--and therefore its range of values is twice as big. Be aware that to make a `float` number in Java, you have to add the letter F (in capital or lowercase form) at the end of the number.

Examples of numeric variable declarations

```
int myAge = 21;
float myGPA = 3.75f; ①
double myFriendsGPA = 3.54;
```

① The `f` suffix denotes that the value 3.75 is a `float` rather than a `double`.

2.2.1. Who Cares About Variable Size?

The general rule in programming is to be as efficient with your resources (memory, storage, processing speed, network bandwidth, etc.) as possible. If you're storing a person's age, you don't really need an `int`; nobody is going to be two million years old! A `byte` has plenty of room (up to 127) to store even the oldest person's age, and it uses a fraction of the memory—8 bits instead of 32. So I should be telling you to use a `byte` in this case.

But I don't worry about that with beginning programmers for two reasons. First, it's hard enough for a beginner to write programs that work—so instead of asking you to deal with *all* of the numeric types, I just have you use `int` whenever you need a whole number, and `double` when you need something with a decimal. Second, these days even the wimpiest computer has waaaaaay more RAM than is needed by even the most complex program a beginner will write, so we don't need to worry about it.

But understand that this attitude is only a teaching and learning aid. It's like saying we shouldn't worry about the price of groceries because we have plenty of money. That might be true, but it's

good to be careful with your money—and it's irresponsible of me to tell you to just waste your money.

As you get more comfortable with programming, use your memory resources efficiently. While you're learning, just worry about getting your code to work.

There's another bad data type practice that I use with beginners. Floating-point numbers aren't 100% precise, for reasons that are too nerdy even for us right now. Because of that precision problem, we should never use `float` or `double` for something like money/currency. Instead, Java has something called `BigDecimal`. But this is another place where I value simplicity for beginners, so we just use double for stuff like prices and account balances in our code. Just know that you'll get fired if you do that at your job with the bank.

2.3. `boolean` Data Type (and Boolean Logic)

A `boolean` variable has only two possible values: `true` and `false`. It's useful for tracking information that is only one thing or the other. *Am I passing this class?* The answer to that is either `true` or `false` --there is no other possibility. *Does Julia own a car?* Again, only two possible answers to that question; she either owns one or she doesn't.

The best practice is to name a `boolean` variable in a way that expresses this either/or state. That is to say, we usually name our `boolean` variables using words like *is*, *has*, *can*, and so on.

Examples of Boolean variable declarations

```
boolean isPassing = true;
boolean hasCar = false;
```

`boolean` variables go hand-in-hand with *Boolean expressions* which are statements that evaluate to be either `true` or `false`, like those questions above. We'll look at this "Boolean logic" in the chapter on decisions.

2.4. `char` Data Type

If we want to store a single character, like a student's letter grade or their first initial, we can use the `char` data type. Char literals are created by putting a character in single quotes, and that character can be a letter, number, punctuation mark or symbol—or some other weird stuff, too.

Examples of character variable declarations

```
char myLetterGrade = 'A';
char firstInitial = 'T';
```

Some people pronounce the `char` data type like the word "chart" without the "t". And some people pronounce it like the word "care". Either is okay.

Also, I pronounce it "char" like "chart" without the "t" and firmly believe that only a sociopath would say it like the word "care".

A `char` is really just an integer, but the number it holds conforms to a standard list of character values called **ASCII** (with the fun pronounciation "Ass key"). In this standard, the number 65 is a capital 'A', 66 is 'B', and so on. Lowercase letters are considered different characters, so 97 is 'a' and 98 is 'b'. Check out the complete ASCII table if you're curious.

2.5. Strings

A `char` is pretty limited since it can only hold a single character. If we want to put a collection of characters together to make words and sentences, we need multiple `char`s grouped into a single variable. That data type is called `String`, because it strings together a bunch of `char`s, like a string of holiday lights.

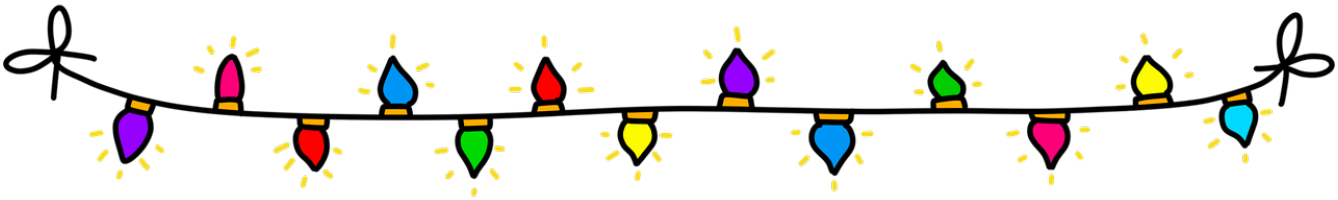


Figure 3. A `String` variable is multiple `char`s strung together like a set of lights.

A `String` is different from the other data types we've looked at so far. The types we've seen so far are **primitive data types**, and `String` is what's called a **reference data type** (though some folks might prefer the more generic "non-primitive data type"). Primitives are stored differently in memory, and they are not *objects*--which we'll learn more about later. For now, a really important thing to notice and remember is that primitive types start with a lowercase letter (`int`, `double`, etc.) and reference types start with a capital letter: so you have to remember to type `String` with a capital S.

`String` literals are denoted with double quotation marks.

Examples of `String` variable declarations

```
String address = "3000 N. Dysart Road";
String bestClass = "CIS150AB";
```

Strings are really important and there's all kinds of fun and useful stuff we can do with them, but we'll need to save all of that for later while we stay focused on the basics.

Always remember that, in Java, single quotes mean a `char` and double quotes mean a `String`. It's easy to get them mixed up—especially if you use Python, where they are interchangeable—but your code won't compile if you mix them up.

2.6. Declaring Constants

A constant is similar to a variable, with two rules:

1. A value must be assigned when the constant is declared.
2. The assigned value can't change during program execution.

To create a constant, add the keyword `final` at the start of your statement, followed by the rest of a declaration and assignment statement you'd use for a variable. So that people looking at your code can easily tell it's a constant rather than a variable, it should be named with all capital letters, using the underscore character to separate words.

Examples of constant declarations

```
final double SALES_TAX_RATE = 8.7;
final int MINIMUM_AGE = 18;
final String FAVORITE_CLASS = "CIS150AB";
```

There are a few different reasons to use constants in your code. For now: * Constants improve readability—they identify the purpose of a value within your code. * Constants prevent writing code that accidentally changes a value that shouldn't change. * Constants make code easier to maintain/update. * In some situations, constants are more efficient than variables.

Interesting!

The naming convention used for Java constants is called **snake case**. More specifically, since it's all caps, people refer to it as **screaming snake case**. Snake case with lowercase letters is the standard convention for variables in Python, among other languages.

2.7. Outputting Variable (and Constant) Values

Assigning a value to a variable or constant does not produce any output. If we want to display the output of a variable—or a constant—we just put the identifier in a `print()` or `println()` statement without any quotation marks:

```
String artistName = "Sam Cooke";
int birthYear = 1931;

System.out.print(artistName);
System.out.print(" was born in ");
System.out.println(birthYear);
```

This code output `Sam Cooke was born in 1931`. We can combine output into one statement by creating a string with multiple pieces using the `+` symbol.

```
String artistName = "Sam Cooke";
int birthYear = 1931;

System.out.println(artistName + " was born in " + birthYear);
```

Creating a `String` using the `` ` `` symbol is called _concatenating_. Be thoughtful when concatenating, because the `` ` `` symbol is also used to do addition with numbers, as you'll see.

For our purposes, there's no difference between outputting using separate `print()` statements or concatenating everything in one statement; you can use whichever approach you prefer (and we'll learn other ways to output values, too).

2.8. Math Calculations

To start doing some calculations, we'll use *operators*. You can think of an operator as a symbol that performs a calculation or other action. You've been using an operator already: the *assignment operator*, which uses the `=` symbol. The action it completes is assigning the value on the right of the `=` symbol to the variable on the left. Arithmetic operations work in a similar way. In Java, there are five arithmetic operators:

Table 1. Java arithmetic operators

Operator	Description
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division (quotient)
`%`	Modulo (remainder)

The arithmetic operators work pretty much the way you'd expect, except maybe *modulo*--which might be a term you've never heard before. Each operator acts on the value to either side:

Examples of simple arithmetic operations

```
int sum = 5 + 7; ①
int difference = sum - 2; ②
```

① The value of `sum` will be 12

② The value of `difference` will be 10 (i.e., 10 - 2)

Time To Watch!

Arithmetic Operations in Java

▶ https://www.youtube.com/watch?v=kfVVgFMuR3A *(YouTube video)*

Files from video:

- `Operations.java`
- `OutputtingVariables.java`

2.8.1. Order of Operations

Early on in your math studies you learned about *order of operations* when an arithmetic expression has more than one calculation, and it works the same in Java. We call this *operator precedence*, and here are the guidlines:

1. Any operations enclosed in parentheses are evaluated first, following the rest of the rules here.
2. Multiplication, division, and modulus are evaluated next: the `*`, `/`, and `%` operators. If there are more than one of these operations in the expression, they are evaluated from left to right.
3. Addition and subtraction are evaluated last. As above, if the expression contains more than one `+` or `-` operator, they evaluate from left to right.

Consider the following examples:

Examples of operator precedence

```
int result1 = 17 - 4 * 6 / 3; ①
int result2 = 17 - 4 / 2 + 2; ②
int result3 = 17 - 4 / (2 + 2); ③
```

① `result1` is 8: `4 * 6` is 24, then `24 / 3` is 8, and then `17 - 8` is 9.

② `result2` is 17: `4 / 2` is 2, then `17 - 2` is 15, and then `15 + 2` is 17.

③ `result3` is 16: `(2 + 2)` is 4, then `4 / 4` is 1, and then `17 - 1` is 16.

2.8.2. More Arithmetic with Less Typing!

There's a pretty consistent rule of thumb in coding that says programmers want to type as little as possible, so programming languages often provide shorthand ways of writing code that's used frequently. *Compound assignment operators* (also called *shorthand operators*) simplify the syntax when you need to change a variable's value relative to it's existing value. For example, if we want to add 10 to a `weight` variable that already has the value `145`, we could use the following:

```
weight = weight + 10;
```

Java starts on the right side of the assignment expression and retrieves the current value of `weight`, which is 145, adds 10 to that value, and stores the result back in `weight`.

We can combine the addition operator (`) with the assignment operator (`=`) to make a compound addition operator: `=, which allows use to rewrite the above line of code as:

```
weight += 10;
```

You can use compound assignment operators for all of the arithmetic opeerations:

- `+=` adds the value on the right to variable value on the left.
- `-=` substracts the value on the right from the variable on the left.
- `*=` multiplies the value on the left by the value on the right.
- `\=` divides the variable value on the left by the value on the right.
- `%=` divides the variable value on the left by value on the right, then assigns the **remainder**.

An operation we might not use much now, but will start using a lot when we learn to write loops, is *incrementing* a value, or adding 1 to a value. The *increment operator* (two plus symbols) gives us a very easy way to do that. On somebody's birthday, for example, we could write:

```
age++;
```

The `++` simply adds 1 to the current value of `age`. The *decrement operator* is `--`, and it subtracts 1 from a variable's value. If we're counting down the number of days until our next birthday, we could execute this expression each morning:

```
daysRemaining--;
```

Increment and decrement only require one operand, so we call them *unary* operators.

There are two forms to the increment and decrement operators: **prefix** and **postfix**. These examples use the postfix form, putting the operator after the variable name, whereas a prefix form would have the operator before the variable name: `++age`. There's a subtle difference in how postfix and prefix operations work, but for now you can use them interchangeably. I mention it here only because you might see code examples online using the prefix form.

2.9. Getting User Input

Until now, our code hasn't been interactive—each execution of a program results in the exact same output, and the user never has the chance to input anything. To produce output, we've been using `System.out` to send text to the "standard output device"--your monitor. For input, we'll need to use the "standard input device" (your keyboard) by accessing `System.in`. We can access that input device using something called the `Scanner` class.

Time To Watch!

User Input in Java

▶ https://www.youtube.com/watch?v=8y430BQktYQ *(YouTube video)*

File from video:

- `InputDemo.java`

The `Scanner` class includes a variety methods for working with input "streams" (including input sources other than a keyboard), but the ones you'll need for our work are shown below.

Table 2. Common `Scanner` methods

Method	Description
`nextLine()`	Returns a `String` with everything from the keyboard up until a *line feed*. In other words, this returns a complete line: everything until the user hits enter/return.
`next()`	Returns a `String` with everything from the keyboard up until the next "delimiter," which by default is whitespace. In other words, it returns what the user typed up until the first space, tab, enter/return, etc. For our purposes, it returns the next word from the input.
`nextInt()`	returns an `int`. If the input can't be converted to an `int`, it will cause an error.
`nextDouble()`	returns a `double`. If the input can't be converted to a `double`, it will cause an error.

As a reminder, there's a weird quirk that happens when you get numeric input from a user and then ask for `String` input using `next()` or `nextLine()`. If your program seems to skip that `next()` or `nextLine()`, review that part of the video!

Check Yourself Before You Wreck Yourself (on the assignments)

Can you answer these questions?

1. **What is the difference between a constant and a variable in Java? Provide an example of each.**
2. **Explain the purpose of the `Scanner` class in Java and provide an example of how it is used to get user input.**
3. **Describe the difference between integer division and floating-point division in Java. Why is it important to be aware of this distinction?**
4. **What are compound assignment operators in Java? Provide examples of how they are used in arithmetic operations.**

Sample answers provided in Stuff That's Tacked On The End.

3. 0011 - Methods

Help Make These Materials Better!

I am actively working to complete and revise this eBook and the accompanying videos. Please consider using the following link to provide feedback and notify me of typos, mistakes, and suggestions for either the eBook or videos:

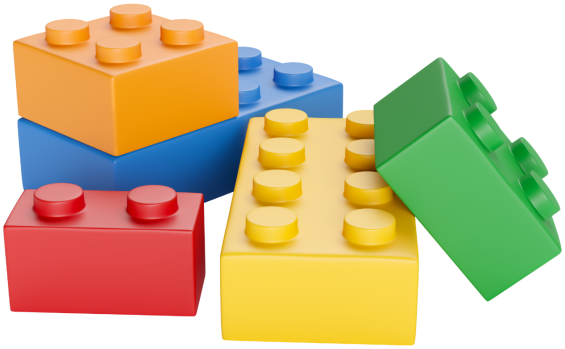
CIS150AB Course Materials Feedback (Google Form)

What's the Point?

- Define and call methods
- Understand variable scope
- Pass data to methods
- Return values from methods
- Overload a method

Source code examples from this chapter and associated videos are available on GitHub.

As our programs begin to get larger and more complex, it will be important to keep our code organized. Each program we write is contained within a *class*, and classes are the basic building block of an OOP program; we'll explore them in more detail in the next chapter. Within a program, we can organize code in **methods**. A method is a collection of statements that work together to complete a single task. Consider the assembly instructions for building a LEGO set.



It might take dozens of small steps to complete the set, but taken together, the instructions execute a single task: building the set. A method is conceptually the same.

If you've worked with other programming languages, you may know methods by a different name. In Python, we use functions, and in other languages we use procedures or subroutines. All of those terms pretty much mean the same thing, but in Java they're called methods. And really, they're generally called methods whenever we're using object-oriented programming, so even in Python we sometimes call them methods.

There are a variety of reasons to break our code into methods, but an important advantage for now is **reusability**. Once we create a method, we can use it as often as we'd like. That means we don't have to type the same code over and over. In turn, that improves our code's **maintainability**. If we have to perform some calculation ten times in our program and have written out that calculation all ten times, a change to the calculation means updating it in all ten places. But if we put that calculation in a method and use that method ten times, we can just update the method and our changes will automatically be used all ten times. We'll see additional advantages to methods as we learn more about programming.

Once we define a method, we **invoke** the method each time we want it to execute. We also say that we *call* a method, which sounds a little cooler than *invoke* but means the same thing.

3.1. Defining Our Own Methods

We've been defining a method from the start. Each program we write includes a method called `main`, which the Java runtime invokes when we run the program.

We've also been calling (*invoking*) methods from the start. To output text, we've been calling the `println()` and `print()` methods.

In Java, parentheses **always** indicate a method, whether or not there is anything inside of the parentheses. Anytime you see parentheses in Java code, you're looking at either a method definition or a method call.

Consider this simple method definition:

```
public static void displayCopyright() {
    System.out.println("(C) 2025 by Tim McMichael. All rights reserved.");
}
```

- The `public` keyword is an *access modifier*. Defining the method as `public` makes it possible to call the method from anywhere. More on this later, but for now all of our methods will be `public`.
- The `static` keyword basically means that the method belongs to the class and not an object. That will make more sense in the next chapter, but for now our methods will be `static`.
- The `void` keyword indicates that this method will not return anything. This is called the method's *return type*, and we'll learn about this shortly.
- `displayCopyright` is the identifier for the method, in the same way variables have identifiers. Like variable identifiers, use camelCase to make the name readable—and make the name descriptive. The convention is that the name of a method should describe what it does; usually, that means the identifier is (or includes) a verb.
- The parentheses indicate to the compiler that this is a method, as opposed to a variable or some other fancy thing. You'll see what we can put in the parentheses shortly.

Time To Watch!

Defining and Calling Methods in Java

▶ https://www.youtube.com/watch?v=hAxUD7xV7h8 *(YouTube video)*

Files from video:

- Starter code: `SimpleMethod.java`
- Completed code: `SimpleMethodFinished.java`

3.2. Variable Scope

As we begin organizing our code into different methods and (when we learn object-oriented programming basics) classes, we'll need to understand how data is compartmentalized within a program. Whenever we create a class or method—and other structures we'll learn as we go—we use curly braces to define the boundaries and indenting to help make those boundaries clear. These braces form *code blocks*.

The outermost code block is our class. Although there are a few things that can go outside the class, like `package` and `import` statements, the class code block contains all of the components of our program.

In some cases we can place one block inside another, as such as putting a method inside a class. This is called *nesting* blocks, and a nested block must completely enclosed; in other words, a method can't be partly in a class and partly out of it.

And some kinds of blocks can't be nested. A method can be nested inside a class, but a method cannot be nested inside another method. Many IDEs, including Visual Studio Code use color coding to make code blocks more clear.

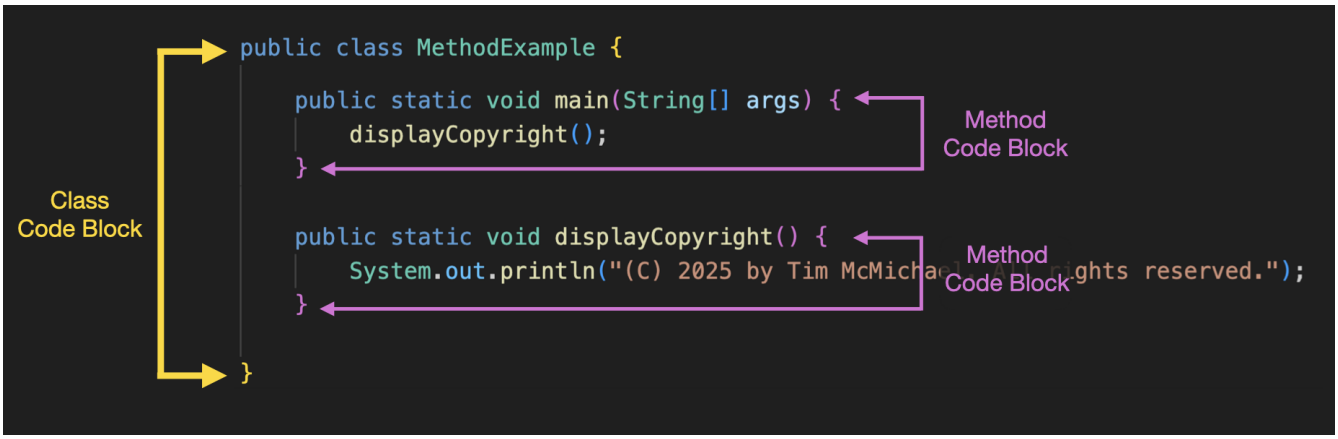


Figure 4. An example of nested blocks in Visual Studio Code.

A variable can only be used or accessed inside the block in which it was declared; that block is the variable's `scope`. When we refer to a variable, the compiler checks within that code block, or scope, to see if the variable has been declared. If it doesn't find a variable with that identifier within the current scope, it will stop compiling. Basically, referring to a variable that is declared in a different scope is the same as referring to a variable we never declared at all. Trying to use a variable in a different code block is referred to as an *out of scope* reference.

`ScopeExample.java`. *An example of code with an out-of-scope variable reference.*

```
public class ScopeExample {

    public static void main(String[] args) {
        int favoriteNumber = 7;
        System.out.println(favoriteNumber); ①

        outputNumber();
    }
```

```
    public static void outputNumber() {
        System.out.println(favoriteNumber); ②
    }
}
```

① This is a valid, or *in scope*, reference because `favoriteNumber` is declared within `main()`.

② This is an invalid *out of scope* reference because `favoriteNumber` can only be accessed within `main()`.

3.2.1. Variable Shadowing

When we first started using variables, we learned that we can't make two variables with the same name, but it's a little more nuanced than that. We can't make two variables with the same name *and scope*. Java **will** allow us to declare a variable with the same name in a different scope, which is called *variable shadowing*. Shadowing is a **very** bad practice, because it often leads to confusion about which variable is in scope.

The example below can be confusing to beginners and to people who are reading the code quickly. When `outputNumber()` is called, another variable named `favoriteNumber` is created and assigned the value `18`. After that is output, an assignment statement changes that value to `10`. Then, program execution returns to `main()`, where a `println()` statement outputs `favoriteNumber` again. However, *this* `favoriteNumber` wasn't changed to 10—the other one was.

`ShadowingExample.java`. *An example of variable shadowing, which we should avoid.*

```
public class ShadowingExample {

    public static void main(String[] args) {
        int favoriteNumber = 7;
        System.out.println(favoriteNumber); ①

        outputNumber();

        System.out.println(favoriteNumber); ②
    }

    public static void outputNumber() {
        int favoriteNumber = 18;
        System.out.println(favoriteNumber); ③
        favoriteNumber = 10;
    }
}
```

① This outputs `7`

② This outputs `18`, because it refers to the variable declared in `outputNumber()`

③ This still outputs `7` because the change to `10` is made to the `favoriteNumber` within the `outputNumber()` method.

3.2.2. Global Variables

As we can see, variable scope has a big impact on how our code runs. Beginning programmers sometimes try to avoid scope issues by declaring their variables within the class code block, which makes them accessible to any block nested within the class. This kind of class-level variable is sometimes called a *global variable*, and the use of global variables is generally discouraged.

`GlobalVariableExample.java`. *An example of a global variable, which we should not use.*

```
public class GlobalVariableExample {
    static int favoriteNumber = 7; ①

    public static void main(String[] args) {
        System.out.println(favoriteNumber);

        outputNumber();

        System.out.println(favoriteNumber);
    }

    public static void outputNumber() {
        System.out.println(favoriteNumber);
        favoriteNumber = 18; ②
    }
}
```

① Declaration at the class level. Note that global variables must be `static`.

② This changes the value of `favoriteNumber` to 18 for all methods in the program.

Instead, we'll declare all of our variables within our methods; these are called *local variables*.

The use of global or class-level variables in code that you turn in for an assignment in my class is very heavily penalized. As much as possible, I try to reinforce best practices—and that means minimizing the use of global variables.

Of course, this presents a problem. What if we need access to a variable in another method? The best practice is to pass that variable value to the method as needed, and for the method to pass back a value when necessary.

In the next chapter, we will start using variables that look a lot like the global variables I just said we shouldn't use. To be clear, those *instance variables* behave differently and serve a different purpose. They are *global variables* as described here.

3.3. Passing Data to Methods

Sometimes a method needs some information in order to carry out its purpose. For example, the `print()` method needs to know what it's supposed to print. To provide information to a method, we

pass the information in as **arguments**. So, the `String` we want to output is passed to the `print()` method as an argument, and arguments are always placed inside the parentheses:

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

In this example, "Hello World" is an argument.

We establish what information a method needs as part of the method definition. Within the method we're defining, those pieces of information are called **parameters**. A parameter is a variable that exists in the method and receives the argument, and it's declared inside the parentheses in our method definition. The methods we've defined so far didn't need any information, so we haven't been putting anything in the parentheses—but now let's see an example with a parameter.

ParameterExample.java - Defining a parameter and passing in an argument

```
public class ParameterExample {

    public static void main(String[] args) {
        outputGreeting("Tim"); ①
    }

    public static void outputGreeting(String name) { ②
        System.out.println("Hello, " + name + "!");
    }

}
```

① "Tim" is the argument.

② `name` is the parameter.

In the above example, "Tim" is passed to the `outputGreeting()` method as an argument. Within that method, the parameter `name` stores the argument, so when this code runs, `name` is equal to "Tim".

The actual value passed in when we call a method is referred to as an *argument*. The variable that receives that value within the method is referred to as a *parameter*.

Time To Watch!

Passing Data to a Method in Java

▶ https://www.youtube.com/watch?v=DNJjyKykPvE *(YouTube video)*

File from video:

- Starter code: `AreaOfCircle.java`
- Starter code: `AreaOfOval.java`
- Completed code: `AreaOfCircleFinished.java`

- Completed code: `AreaOfOval.java`

3.4. Returning Values

The methods we've seen to this point are basically commands—they simply perform a task, and then when they're done, program execution just goes back to the method that called it and resumes there. But we can also create methods that are like questions—they perform a chunk of code, but when they are finished they give back an answer.

Consider this method:

```
public static void printFavNum() { ①
    int favNum = 10 - 3;
    System.out.print(favNum);
}

public static int getFavNum() { ②
    int favNum = 10 - 3;
    return favNum;
}
```

① This specifies a return type of `void`

② This specifies a return type of `int`

The first method is a command to print out a favorite number, so it does not return anything. The `void` in the method header is the return type, and *void* basically means nothing; this method returns nothing. The second method is asking to get a favorite number, so it is going to give back an `int`. The return type is specified as `int`. The `return` statement sends the `favNum` value back to the method that called `getFavNum()`.

If a method has a return type of anything other than `void`, it will only compile if it has a `return` statement followed by a value (literal or variable) of the specified type.

This means that methods themselves essentially have data types. `printFavNum()` has a data type of `void`. `getFavNum()` has a data type of `int`. Since methods have types, we can use them in statements just as we'd use a literal or variable of that type. For example, the following line of code is valid:

```
int answer = 18 + getFavNum();
```

This evaluated the same way as any other assignment statement. It starts on the right and finds that method call, so it will execute `getFavNum()` and plug the returned value into the operation, resulting in `18 + 7` and ultimately evaluating to `25`, which is then assigned to `answer`.

A `return` statement in a `void` method stops execution of the method and returns to the calling method.

```
public static void shortCircuit() {
   System.out.print("This runs...");
   return;
   System.out.print("This can never run!");
}
```

The second print() statement won't execute because the return statement ends the method. The compiler doesn't like these kinds of *unreachable statements*, though, so it will not compile.

`return` statements in `void` methods will have some uses for us later on.

A Java method can only return one piece of data, so it can only have one return type.

3.4.1. Returning vs. Outputting

Generally speaking, it's better to return values from a method rather than outputting values. There are a few reasons for this, but consider an obvious one. If we use a `print()` method, our code is limited to only working in a console application. That's fine for now, because it's the only kind of application we know how to make! But what if we want to use that same code in a web application, or a mobile app? That `print()` statement won't work—rather, the user will never see the result, because they won't have a console window.

Consider this code:

```
public class BadOutput {
    public static void main(String[] args) {
        kingOfSoul();
    }

    public static void kingOfSoul() {
        System.out.println("Otis Redding");
    }

}
```

If the `kingOfSoul()` method knows who the King of Soul is, how do we print that out if we can't perform the output inside the method? The solution is to return the String and perform the output in `main()`.

```
public class GoodOutput {
    public static void main(String[] args) {
        System.out.println(kingOfSoul());
    }

    public static String kingOfSoul() {
        return "Otis Redding";
```

```
    }

}
```

This is another example of something that seems annoying, like it's just extra work. When we're learning new things, we sometimes have to accept the wisdom of people who are experienced—and recognize that eventually we'll see the point. We're all about learning good habits and best practices around here, so we'll almost always return values rather than printing them.

There are times when we want a method whose sole purpose is to produce some output. In that case, be sure to give it an appropriate identifier. Notice that those kinds of methods in my examples have *print* or *output* in the name.

I rarely want students to create methods that produce output, and when I do I always make that explicit in my directions. When I refer to *returning* something, I mean just that. The rule of thumb is, **all input and output statements should be in the `main()` method** and data should be passed around as necessary.

I strongly penalize input and output statements outside of the `main()` method because I'm trying to build habits that will serve us well as we learn more about programming.

Time To Watch!

Returning Data from a Method in Java

▶ https://www.youtube.com/watch?v=JI0e0vVONmM *(YouTube video)*

File from video:

- Completed code: `AreaOfOval.java`

The Lab Assignments in Canvas can be completed using what we've covered to this point. You might choose to complete that work now, then move onto the next section—which you'll need for the Programming Project.

3.5. Overloading a Method

Sometimes the task, action, or calculation that a method produces has different ways of operating depending on the circumstances.

Consider a method that calculates the average age of two people:

```
public static double averageAge(int age1, int age2) {
       return (double) (age1 + age2) / 2;
   }
```

This is a pretty straightforward method. Notice that the return statement uses casting with `(double)` to ensure that the result is not truncated.

If we want to calculate an average age of 3 people, we could almost use the same method. We want a method that still calculates the average age, but takes three arguments and divides by 3, instead of 2.

To create another version of a method that operates a little differently, we can use *method overloading*. To overload a method, we write a new method with the same identifier, but with a different set of parameters. An overload for our `averageAge()` method could look like this:

```
public static double averageAge(int age1, int age2, int age3) {
        return (double) (age1 + age2 + age3) / 3;
    }
```

Note that the method identifier is exactly the same, but this version accepts three `int` arguments instead of two. That difference allows the compiler to easily determine which implementation of the method is being called: if there are two `int`s in the parenthesis, it calls the first implementation, and if there are three `int`s, it calls the second implementation. When we're using the method, we can call whichever best suits our needs at the time.

The compiler can also easily distinguish overloaded methods if the *types* of the parameters are different. An implementation that accepts `double`s is valid:

```
    public static double averageAge(double age1, double age2, double age3) {
        return (double) (age1 + age2 + age3) / 3;
    }
```

If the compiler sees a call to `averageAge()` with three `double` values, it will invoke this last version.

3.5.1. Incorrect Overloading

Overloaded methods **must** have differences in the number and/or types of the parameters. The *names* of those parameters doesn't differentiate them, so different names is not enough to make a valid overload.

`BadOverloading.java`. *An invalid example of overloading.*

```
public class BadOverloading {
    public static void main(String[] args) {
        System.out.println(area(15, 10)); ①
    }

    // Calculates area of a rectangle
    public static double area(double length, double width) {
        return length * width;
    }
```

```
    // Calculates area of an oval
    public static double area(double smallRadius,
            double bigRadius) {
        double area = 3.14 * smallRadius * bigRadius;
        return area;
    }

}
```

① The compiler can't tell if we want the area implementation of a rectangle of the implementation for an oval.

The term we use to describe a method's identifier and parameter list (the number, order and types of parameters) is a *method signature*. The method signature must be unique so the compiler can identify which method code to run.

`AverageAge.java`. *A valid example of method overloading.*

```
public class AverageAge {
    public static void main(String[] args) {
        System.out.println(averageAge(1.25, 1.5, .5)); ①
        System.out.println(averageAge(10, 20)); ②
        System.out.println(averageAge(10, 20, 25)); ③
    }

    public static double averageAge(int age1, int age2) {
        return (double) (age1 + age2) / 2;
    }

    public static double averageAge(int age1, int age2, int age3) {
        return (double) (age1 + age2 + age3) / 3;
    }

    public static double averageAge(double age1, double age2, double age3) {
        return (double) (age1 + age2 + age3) / 3;
    }

}
```

① The compiler sees three `double` values, so it calls the third implementation.

② The compiler sees two `int` values, so it calls the first implementation.

③ The compiler sees three `int` values, so it calls the second implementation.

3.6. Solution Walkthrough

In "solution walkthrough" videos, I give a problem/prompt that is similar to the kinds of work I assign, and then I record myself writing a solution. It's not absolutely mandatory to watch this video, but students report that these videos are particularly helpful.

Time To Watch!

Methods with Parameters and Returns

▶ https://www.youtube.com/watch?v=f08bKXVqxZk *(YouTube video)*

Solution file from video:

- Completed code: `Percentages.java`

Check Yourself Before You Wreck Yourself (on the assignments)

Can you answer these questions?

1. What is the main purpose of using methods in Java, and how do they contribute to code maintainability?
2. Explain the difference between a parameter and an argument in the context of Java methods. Provide an example to illustrate your explanation.
3. Why is it generally better to return values from methods rather than printing them directly within the method? How does this practice improve the modularity and reusability of code?

Sample answers provided in Stuff That's Tacked On The End.

4. 0100 - Classes and Objects

Help Make These Materials Better!

I am actively working to complete and revise this eBook and the accompanying videos. Please consider using the following link to provide feedback and notify me of typos, mistakes, and suggestions for either the eBook or videos:

CIS150AB Course Materials Feedback (Google Form)

What's the Point?

- Understand the basics of object-oriented programming (OOP).
- Use a class diagram to plan a program.
- Create a class in accordance with OOP principles.

- Create a driver class and use objects.
- Define and use constructors.
- Understand the role of `static` in Java.

Source code examples from this chapter and associated videos are available on GitHub.

There are multiple ways to organize a program, and the code we've written to this point could be described as *procedural programming*. In procedural programming, we organize our code into methods that perform specific tasks, and we think of a program as a series of tasks that need to be performed in a specific order. This course is all about *object-oriented programming* (OOP), but I want students to have a handful of basic coding skills before we dive into OOP. In this chapter, we'll start our transition to OOP by learning about classes and objects, which are the basic building blocks of OOP.

4.1. OOP Basics

Until now, we have been organizing our programs into methods, which is a procedural programming approach. We've been thinking of a program as a collection of tasks that need to be performed in a specific order. There's nothing wrong with this approach, but it can get cumbersome as a program grows in size and complexity—and it isn't always an intuitive way to plan our programs.

Now we're ready to start our transition to object-oriented programming (OOP), which lets us think about programs the way we think about the world around us. Instead of thinking about a program as a bunch of tasks, we can think of it as a bunch of objects that interact with each other. As an example, the old-school video game **Pac-Man** has the main character (an object) moving around a maze (another object), and avoiding four ghosts (yep, four more objects).

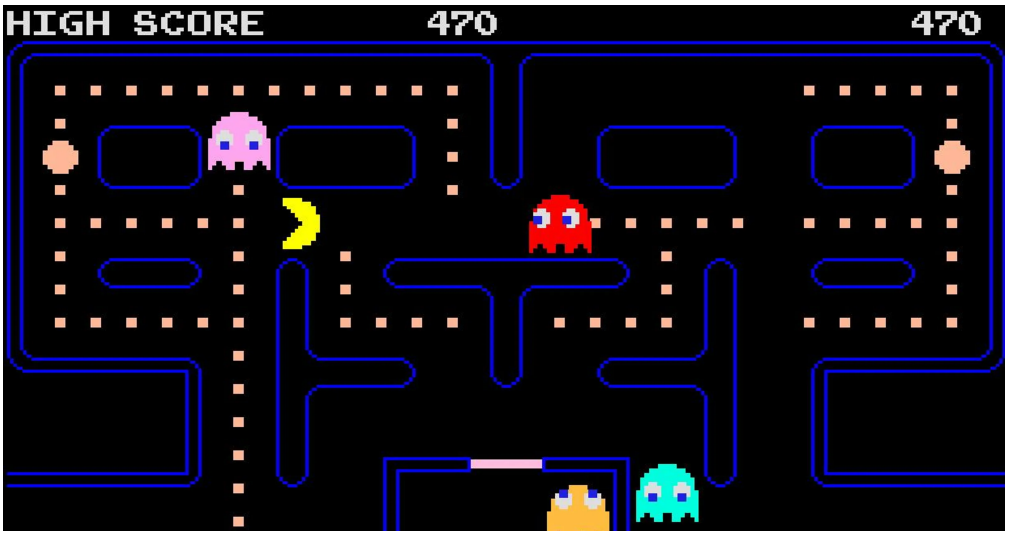


Figure 5. Pac-Man arcade game, released in 1980 by Namco

If we decide to create Pac-Man, we'll still be using methods, but they won't be the basic building blocks of our programs. Instead, we'll be organizing our programs into **classes**, which are blueprints—or recipes—for creating **objects**. A class defines the **attributes** and **behaviors** of the

objects—and those are the variables and methods we'll write to create our programs.

Time To Watch!

Intro to OOP: Classes and Objects

▶ https://www.youtube.com/watch?v=Hnzm4sVsIAI *(YouTube video)*

4.1.1. Class Diagrams

In OOP, our planning for a program starts by deciding what objects—and therefore, what classes—we'll need in order to implement the functionality we want. To help organize our thinking, we'll use a tool called a **class diagram**, which gives a visual representation of the classes and their relationships in our program. The format programmers use for class diagrams is called UML (Unified Modeling Language). The UML standards are extensive, but we'll be using a simplified version in this course.

Consider a program to handle orders at a restaurant. A simple class to represent one person's order might look like this:

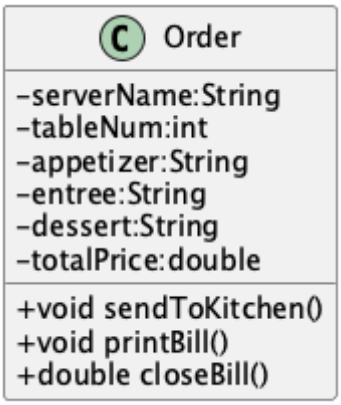


Figure 6. Example of a class diagram for a restaurant order

The top section of the diagram has the name of the class, which is `Order`. The next section is for the attributes of the class—the information the class will store. This example has attributes for the server's name, the table number, the items ordered, and the total price. The bottom section is for the behaviors of the class, which are the actions the class can perform (or that we can do with the class). With an order, we might send it to the kitchen, print the bill, or close the bill.

We'll learn more about the symbols and conventions of UML as we go along, but for now the important part is that we can use class diagrams to help us plan our programs—and to talk about OOP concepts without bogging down on code details.

Your Canvas course includes access to **Lucid (Whiteboard)**, which is a free online tool you can use to create UML diagrams. If you create your account through the link in the left-hand navbar in Canvas, you'll have access to the premium features for free.

4.2. Encapsulation

Object-oriented programming is all about creating objects that can interact with each other. Since the objects will be interacting, we need to think about how to keep them from interfering with each other in ways we don't want. If we have an `Order` class in a program used by a restaurant, we don't want some other class to change attributes in a way that disrupts the program—like changing the entree selection to something that's not on the menu, or setting the price to a negative number, for example.

To prevent this kind of tampering, whether it's intentional or accidental, OOP relies on a concept called **encapsulation**. Encapsulation of a class means that attributes are hidden from the outside world, and only the behaviors of the class can access and change them. As an analogy, consider the counter at a fast food restaurant. We can't just reach over and grab a handful of fries; we have to ask the employee behind the counter to get them for us. In this analogy, the food is encapsulated and we can only access it by using a behavior, like "order food".

Another way to think of encapsulation is the way we interact with other people in social situations. When we encounter a stranger, they don't automatically know our name and phone number; they have to ask us for that information. We've encapsulated our personal information, and we only share it when and how we choose to.

In Java, encapsulation is not a strict requirement, and our code will still work if we don't use it. But it's a best practice—and an important one—so we will encapsulate all of our classes in this course. In fact, I would argue that if we don't encapsulate our classes, we're not really doing object-oriented programming. And that's what we're here to learn.

4.3. Defining and Using a Class

We'll look at a program to keep rudimentary weather records; for a single day's weather data, we'll have a class called `WeatherRecord`.

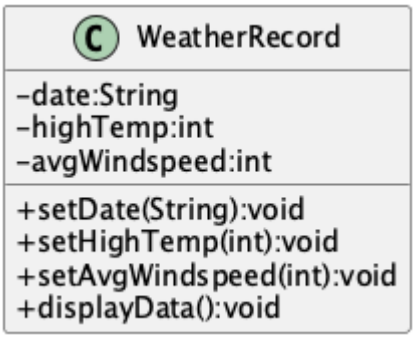


Figure 7. Class diagram for the `WeatherRecord` class

To implement this class in code, we'll start with a class header. The class header always follows the same pattern: an **access modifier**, the keyword `class`, and the name of the class. The class header is followed by a code block, enclosed in curly braces.

`WeatherRecord.java`. *A class header and code block.*

```
public class WeatherRecord {
```

```
    // class code goes here

}
```

Access modifier

The `public` keyword means that the class can be accessed from any other class. Though this is technically optional, we should always use `public` for now.

`class` keyword

The keyword that tells Java we're defining a class. It's required. We'll eventually be able to create different kinds of classes and OOP structures, but for now we're just creating regular classes.

Class name (or *identifier*)

The name of the class, which should be a noun that describes the object the class represents—and is singular, so there's no *s* at the end. The identifier should start with a capital, with the first letter of each word capitalized (like `WeatherRecord`). This is similar to the *camelCase* naming convention we've been using for variables and methods; it's called *PascalCase*.

The class code block is where we define the different components that make up the class, which we call the **instance members**. To begin with, we'll focus on two types of instance members: **fields** and **methods**.

4.3.1. Fields

Fields are the implementation of the attributes of the class. They are also known as **instance variables** because they are similar to the variables we've been using in our programs, but their scope is the object created from the class, not the method where they're defined. A field is unique to the object; if we make two objects from our `WeatherRecord` class, each object will have its own date, high temperature, and average temperature.

Fields are declared like our other variables, but they are encapsulated using the `private` access modifier. This means that the fields can only be accessed and changed by the methods of the class, not by other classes—which controls how the data is used and prevents accidental or invalid changes.

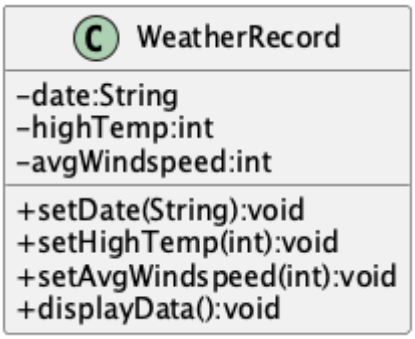
Since a class will compile and run even if we leave off the `private` access modifier, it's easy to forget to use it. But don't worry, I'll help you remember by taking huge points off your assignments if you don't make your fields `private`. As I've mentioned, you're not really doing OOP if you don't encapsulate your fields, and we're learning OOP here.

`WeatherRecord.java`. *Fields added to the* `WeatherRecord` *class.*

```
public class WeatherRecord {
   private String date;
   private int highTemp;
   private int avgTemp;
```

```
}
```

In our original class diagram, we indicated that the fields were private by using a `-` in front of the field name.



You might remember from the section on variable scope in Chapter 0011 that using global variables is terrible, and every time we create a global variable, a puppy loses its favorite toy. And these fields look an awful lot like global variables. But fields in a class are **not** global variables; they're **instance** variables, and they're a good thing. The fields are **encapsulated**, so they can only be accessed and changed by the methods of the class—which is a good thing. And the fields are unique to each object, so we can have multiple objects with different values for the fields—which is also a good thing. And so no puppies' toys will be harmed as long as we use **private** fields correctly.

4.3.2. Methods

Ensuring that our fields are `private` is the first step in encapsulating our class, but it's not the only step. We also need to create methods that can access and change the fields—otherwise, the fields are useless. So far, our methods have included the keyword `static`; we'll learn more about that shortly, but when we make methods for an OOP class, we'll leave off that `static` keyword. These *nonstatic* methods are called **instance methods**, and they are otherwise very similar to the `static` methods we've been using.

Though there are exceptions, most of these instance methods will be `public`, so they can be accessed from other classes. Remember, the foundation of encapsulation is having `private` fields and public methods to permit interactions with that data.

In broad terms, we can categorize instance methods into two types: **accessor** methods and **mutator** methods.

Accessor Methods

Accessor methods give access to the fields of the class, but they don't change the fields. Think of them as "read only" methods, and often all they do is return the value of a field. Java naming conventions specify that accessor methods should start with `get` and then the name of the field they access, formatted in camelCase. Because of that convention, another name for accessor methods is

getters.

Example of an accessor method (or "getter")

```
public int getHighTemperature() {
        return highTemperature;
    }
```

The return type of an accessor method is the same as the type of the field it accesses; in this case `highTemperature` is an `int`, so the return type of our getter is `int`.

A getter allows other classes to be able to read the value of a field; if they don't need to know the value, we just don't write a getter for that field. But read-only access usually does no harm, so often we'll have getters for all of our fields.

`WeatherRecord.java`. *Getters added to the* `WeatherRecord` *class.*

```
public class WeatherRecord {
   private String date;
   private int highTemp;
   private int avgTemp;

   public String getDate() {
      return date;
   }

   public int getHighTemp() {
      return highTemp;
   }

   public int getAvgTemp() {
      return avgTemp;
   }

}
```

Mutator Methods

Mutator methods change the fields of the class. Though they sometimes return a value, their primary purpose is to change the value of a field and they often have a `void` return type. As we're getting the hang of this OOP thing, we'll create a lot of mutator methods that are just **setters** --methods that set the value of a field. The naming convention for setters is to start with `set` and then the name of the field they change, formatted in camelCase; they usually have a void return type.

```
public void setHighTemperature(int temp) {
        highTemperature = temp;
    }
```

The parameter of a setter is the same type as the field it changes; in this case `highTemperature` is an `int`, so the parameter of our setter is also an `int`. All this method does is accept a new value and assign it to the field.

Choosing to write setters isn't quite as straightforward as with getters, where there's generally no harm in exposing read-only access to everything. But we really should only write setters for fields that we want to be able to change from outside the class.

As a rule of thumb for beginners, create **getters** for all of your fields when you first write your class, and then add **setters** only as you need them. Because this is sometimes tricky for beginners to determine, I don't deduct points for writing unnecessary setters—but sometimes my directions will explicitly tell you not to write a setter for a field, and I do deduct for that.

If you're paaying attention to what we're doing here, you might be thinking that these setters really just give public access to the fields, which seems to go against the whole idea of encapsulation. That's true for now, but only because we don't know enough Java yet to do anything about it. As we learn more about Java, we'll be able to write more complex methods that can control **how** fields are changed—for example, by checking the new value to make sure it's valid and won't break anything. But for now, this is just another one of those frustrating rules that you just have to follow until you know enough to understand it.

`WeatherRecord.java`. *Setters added to the* `WeatherRecord` *class, and comments identifying the parts.*

```
public class WeatherRecord {
    // Fields
    private String date;
    private int highTemperature;
    private double averageWindSpeed;

    // Getters
    public String getDate() {
        return date;
    }

    public int getHighTemperature() {
        return highTemperature;
    }

    public double getAverageWindSpeed() {
        return averageWindSpeed;
    }

    // Setters and Mutators
    public void setDate(String date) {
        this.date = date;
    }

    public void setHighTemperature(int highTemperature) {
        this.highTemperature = highTemperature;
```

```
    }

    public void setAverageWindSpeed(double averageWindSpeed) {
        this.averageWindSpeed = averageWindSpeed;
    }
}
```

Sometimes mutator methods don't follow the exact pattern and purpose of setters (simply setting a field's value). For example, a method might perform a series of calculations and changes to multiple fields, or it might change a field based on the value of another field. These methods are still mutators, and we might even still refer to them as *setters*, but they don't always follow the `setFieldName` naming convention.

Time To Watch!

Designing an OOP Class

▶ https://www.youtube.com/watch?v=xcdLgbwtYdc *(YouTube video)*

4.3.3. Using the Class

As we've learned, defining a class establishes a blueprint; to make use of a class in a program, we need to use that blueprint to create an object. We can as many objects from a class as we need, and each object is known as an **instance** of the class. And creating an instance is called **instantiating** a class.

To create our first objects, we use the same two steps we've been using to create variables: a declaration statement and an assignment statement. The declaration is still a *data type* and an _identifier, but in this case the data type is the name of the class:

Example of a declaration statement for an object.

```
WeatherRecord day1;
```

This creates a variable called `day1` that will point to—or *reference*--the memory location where our object will be stored. The identifier follows the same rules we learned for primitive variables: a descriptive name typed in camelCase (with a lowercase first letter). In this case, the `day1` object is going to maintain the record for the first day of our weather tracking.

The assignment statement works the same, but what we're assigning looks a lot different. We'll use the `new` keyword to allocate memory, and then we'll call a **constructor**.

Example of an assignment statement for a newly declared object.

```
WeatherRecord day1;
day1 = new WeatherRecord();
```

We're soon going to spend a lot of time learning about constructors, but here are the takeaways for now: the identifier is exactly the same as the class name, and it's followed by parentheses.

We've already learned that parentheses in Java **always** means we're referring to a method. A constructor is a special method called when instantiating an object.

Just like with variables, we often combine those two statements into one line of code:

```
WeatherRecord day1 = new WeatherRecord();
```

Now that we have an object, we can call its instance methods using *dot notation*, which means we put the object name (**not** the class name!), followed by a dot, followed by the method call:

Instance method calls using dot notation.

```
WeatherRecord day1 = new WeatherRecord();
day1.setHighTemperature(87);
System.out.println("High temperature on day 1:" + day1.getHighTemperature());
```

In this example, we're setting the `highTemperature` field of `day1` to 87 degrees, and then we're retrieving the high temperature and outputting the returned value. This is a good test of the *set* and *get* methods for the `highTemperature` field.

It's easy for beginners to forget to use that dot notation. To see why it's necessary, consider the following example.

An instance method call using dot notation with multiple objects.

```
WeatherRecord day1 = new WeatherRecord();
WeatherRecord day2 = new WeatherRecord();

day1.setHighTemperature(87);
```

If we left off the `day1.` part of the call, the compiler would not know which `setHighTemperature()` method to use, `day1` or `day2`. Even when we only have one instance, the compiler needs to know where to find that method, so the dot notation is required every time we call an instance method.

Object Classes vs. Driver Classes

Ok, time for another convention that seems only intended to be nitpicky and pointless, but is important and is expected on assignments in this course. OOP nerds value keeping parts of our programs compartmentalized, and that includes separating the class definition and the code that uses the class. A class definition goes in its own file, which must have a filename exactly match the name of the class (with *.java* as an extension)--and that one's not a convention, that's a syntax rule for the compiler. A class we define for use as object can be called an **object class** or a **user-defined class**.

The code that uses the *object class* should be in its own file, and is often called a **driver class**. The driver class contains the `main()` method, which is the entry point for the program. A driver class

actually goes by several different names. Some people call it a *main class* because, well, it's the class with the `main()` method; I don't hear that term a lot, but it is out there. I often use the terms *demo class* or *test class* because, as learners, we're often making a class just to try a specific concept or skill, and the only thing our program really does is show that the object class is working. And in those cases, we often see "test" or "demo"; so a driver class for our `WeatherRecord` object class might be called `WeatherRecordDemo` or `TestWeatherRecord`, or something similar.

The point here is that, if we've created an object class called `WeatherRecord`, we're not going to put our `main()` method in that same class/file. We're going to make a separate class—a driver, or demo class, or test class. I don't much care what term you use, as long as it's a separate class and file.

Your pitchforks are already sharpened, but here's the part where you light your torches. **All of your input and output should be in the driver class.** That is, you generally can't have any `print()` or `println()` statements, any dialog popups (if you know how to use `JOptionPane` or similar), or any `Scanner` input calls in your object classes. My examples always demonstrate this *separation of concerns*, so you'll have plenty of examples of what I mean.

Interesting!

I asked an AI platform to give me an image of an angry mob with pitchforks and torches (coming after me because I make them separate their input and output), and the AI spit out this nightmare fuel. I decided it was too creepy to display in the text, but this crime against nature should be preserved for posterity. If a mob like that comes for me, I'll let them put `print()` statements wherever they want!

Why can't we put input/output in our object classes? * To "decouple" the UI from the *business logic* or guts of our program. This makes our code reusable in a variety of projects, such as web pages, mobile apps, and GUI applications—none of which are friendly to console input and output. Look up MVC and MVVM for all kinds of information about that. * To keep our code more readable by keeping the parts clearly identifiable. * Because I just don't care much about input and output. I care about the classes you create, so I want to look at (and grade) that work separately. If your input and output don't work but your object class looks good, you're still going to get a good grade—if I'm able to separate out those mistakes.

Unfortunately, this is one of those things that boils down to, "because I said so" and "you'll thank me later." Sorry, I can't do much better than that for now.

`WeatherRecordDemo.java`. *A driver class to demonstrate the* `WeatherRecord` *class.*

```
public class WeatherRecordDemo {
    public static void main(String[] args) {
        // Instantiate two objects
        WeatherRecord day1 = new WeatherRecord();
        WeatherRecord day2 = new WeatherRecord();

        // Set field values for both instances
        day1.setDate("2024-10-01");
        day1.setHighTemperature(87);
```

```
        day1.setAverageWindSpeed(1.5);

        day2.setDate("2024-10-02");
        day2.setHighTemperature(75);
        day2.setAverageWindSpeed(8.25);

        // Output field values for both instances
        System.out.println("Date: " + day1.getDate());
        System.out.println("High Temperature: " + day1.getHighTemperature());
        System.out.println("Average Wind Speed: " + day1.getAverageWindSpeed());

        System.out.println("-----------------------------");

        System.out.println("Date: " + day2.getDate());
        System.out.println("High Temperature: " + day2.getHighTemperature());
        System.out.println("Average Wind Speed: " + day2.getAverageWindSpeed());
    }
}
```

The driver class above creates two instances of the `WeatherRecord` class, uses each setter, then outputs the return from each getter. This ensures that instance variables are independent of each other and all instance methods work correctly. In general, I ask students to create at least two instances of each class they are demonstrating.

Time To Watch!

Implementing and Using a Class in Java

▶ https://www.youtube.com/watch?v=E0HFACqWgP4 *(YouTube video)*

Files from video:

- Completed code: `CellCustomer.java`
- Completed code: `CellCustomerDemo.java`

The Lab Assignments in Canvas can be completed using what we've covered to this point. You might choose to complete that work now, then move onto the next section—which you'll need for the Programming Project.

4.4. Constructors

When we instantiate a new object, the syntax includes a call to a method, immediately following the `new` keyword:

```
WeatherRecord day1 = new WeatherRecord();
```

This is a call to a special method called a **constructor**. A constructor runs when an object is instantiated, and it's used to set up the object with any initial values or behaviors. A constructor's

primary job is to initialize the fields of the object—to give each instance variable a value. If we don't write a constructor, the compiler will create one for us; it's called a default constructor, and it will set all fields to their default values. For example, numeric fields like `int` and `double` will be set to `zero, and `String` fields will be set to `null`. We've been using setters to change those initial values to what we want, but we can also write our own constructors to set those values when the object is created.

Constructors are a special kind of method, so their syntax is a little different from other methods. A constructor is always public, it has no return type (not even `void`), and its name is the same as the class name. A definition for a constructor for the `WeatherRecord` class would look like this:

Example of a constructor for the `WeatherRecord` class.

```
public WeatherRecord() {
    // code to initialize fields goes here
}
```

The most important job of a constructor is setting values for each field of the object. As a beginner, our rule of thumb is to just make a simple assignment statement for each field.

Example of a constructor that initializes fields.

```
public WeatherRecord() {
    date = "2025-01-01";
    highTemperature = 0;
    averageWindSpeed = 0.0;
}
```

Since our WeatherRecord class has three fields, we've got three assignment statements in our constructor. We can initialize those fields to any value we want, but we should choose values that make sense for the object; whatever we put there will be the default values that each object gets when it is instantiated.

Constructors should be written at the top of the class, before the fields and methods.

This constructor is called a **parameterless** constructor because it doesn't have any parameters in the parentheses. It's technically **not** a default constructor, because we wrote it ourselves rather than letting the compiler do it, but so many people call it a default constructor that the term is used more often than *parameterless constructor*.

Constructors can also have parameters, which allows us to pass values to the constructor when we instantiate an object. This is useful when we want to set the initial values of the fields to something specific, rather than the default values. We add parameters to our constructor just like we do with any other method, by listing the data type and identifier in the parentheses.

Example of a constructor with parameters.

```
public WeatherRecord(String date) {
```

```
    this.date = date;
    this.highTemperature = 0;
    this.averageWindSpeed = 0.0;
}
```

To use this constructor, we pass a `String` value when we instantiate the object:

```
WeatherRecord day1 = new WeatherRecord("1998-01-25");
```

There are a couple of important things to note about this example:

1. This constructor only has one parameter but it still has three assignment statements. All fields need values, so if we don't have a parameter to get a field's value, we need to set it to a default value.
2. The parameter has the same name as the field: `date`. This is a common practice, but it's potentially confusing. And it also violates guidance I gave you when we learned about variable scope.

   In this case, the parameter is a *local variable* to the constructor, and it's *shadowing* the field. Our assignment statement needs to be carefully written: `this.date` refers to the field, and `date` refers to the parameter.

We can also overload constructors, which means we can have multiple constructors with different parameters—just like we can with any other method. That can include having a parameterless constructor and one or more constructors with parameters, or having multiple constructors with different numbers of parameters.

To see a complete example of the `WeatherRecord` class with constructors, fields, and methods, as well as a driver class to demonstrate it, visit the_Source code examples from this chapter and associated videos are available on GitHub repository for this chapter.

Time To Watch!

Constructors in Java

▶ https://www.youtube.com/watch?v=zHxssPU1_Hw *(YouTube video)*

https://youtu.be/zHxssPU1_Hw

Files from video:

- Completed code: `MobileCustomer.java`
- Completed code: `MobileCustomerDemo.java`

4.4.1. Constructors and Encapsulation

Constructors allow us to be stricter with our encapsulation since now we don't have to have setters

to put data into our objects. We can provide a constructor to accept all the data the object needs, decide if we give access to change a field after the object has been instantiated.

For example, if we're making a bank account object, we'd need to provide an account number when we create the account, but we probably shouldn't allow the account number to be changed after the account is created. In that case, our constructor would accept the account number, but we wouldn't provide a setter for the account number.

4.5. `static` Constants and Methods

Since our first `Hello World` program, we've been using the `static` keyword to create methods—beginning with the `main()` method that is the starting point for every Java program. However, we haven't had enough context to understand what that keyword means.

We've learned about *instance members*, which are the fields and methods that belong to a class, and *instance members*, which are the fields and methods that belong to an object. Fields are instance members, which means that each object has its own copy of the field that can be changed without affecting other objects. Instance methods are the code that an object can run, and they can access and change the fields of the object. Instance members are defined without using the `static` keyword, so we refer to them as *nonstatic* members.

When we use the `static` keyword, we're creating a *class member*--a field or method that belongs to the class itself, not to any object created from the class. Put another way, a class member is shared by all objects created from the class, and it can be accessed without creating an object.

`System` is a class that includes the print() and println() methods we've been using, and those methods are `static`. Since they're `static`, we can call them without creating a `System` object:

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

If `println()` were an instance method, we'd have to create a `System` object before we could call it:

```
System mySystem = new System(); mySystem.println("Hello, World!");
```

That would be a pain, so we're glad that `println()` is `static`.

The `print()` and `println()` are slightly more complicated than that, since they're actually *instance methods* of the `PrintStream` class, which is a class that `System.out` is an instance of. But for our purposes, we can think of them as `static` methods of the `System` class. You don't need to know that right now, but if I don't mention it then somebody will claim I'm an idiot. I probably *am* an idiot, but not for this reason.

Throughout our learning, we'll learn more about using `static` in our programs and classes, but an important one to known about for now is the `static` constant. When we put a `static` constant in an object class (which would also use the keyword `final`), we're creating a value that is shared by all objects created from the class. If we have a savings account class, a common example when learning OOP, we might have a `static final double INTEREST_RATE` constant. This would mean that every savings account would earn the same interest rate, which is often how banks work.

`SavingsAccount.java`. *A class with a* `static` *constant.*

```
public class SavingsAccount {
    private static final double INTEREST_RATE = 0.02;

    // other fields and methods go here. See the repository for the complete code.

    public void addInterest() {
        balance += balance * INTEREST_RATE;
    }

}
```

Check Yourself Before You Wreck Yourself (on the assignments)

Can you answer these questions?

1. What is the primary difference between procedural programming and object-oriented programming (OOP)?
2. Explain the concept of a class and how it relates to objects in OOP.
3. What is encapsulation, and why is it important in OOP? Provide an example.

Sample answers provided in *Stuff That's Tacked On The End*.

5. 0101 - Decisions

Help Make These Materials Better!

I am actively working to complete and revise this eBook and the accompanying videos. Please consider using the following link to provide feedback and notify me of typos, mistakes, and suggestions for either the eBook or videos:

CIS150AB Course Materials Feedback (Google Form)

What's the Point?

- Understand Boolean expressions
- Implement `if` and `if-else` statements
- Use `else if` and nested `if-else` statements

- Apply decision structures to object-oriented programming

Source code examples from this chapter and associated videos are available on GitHub.

Decisions, or *branching* gives our programs the ability to execute different code depending on what's happening as it runs. At the core of decisions is the concept of Boolean logic, which is named after a 19th-century mathematician named George Boole. Boolean logic is the foundation of the `if` statements that will be the foundation of our branching code.

5.1. Boolean Expressions

Until now, our programs have all executed very sequentially and predictably—one line of code after another. How boring! To give our programs the ability to *branch* and execute different code based on different conditions, we need to introduce the concept of *decisions*.

In computer programming, a decision is made based on a **Boolean expression**, which is an expression that evaluates to either `true` or `false`. Think of them as questions that can be answered with a simple "yes" or "no". *Is this student's GPA high enough to qualify for the scholarship? Has this cell phone customer used all of their data? Did the user enter "exit"?*

The `true` or `false` value of a Boolean expression can be used to determine which code block will execute next, so it's important to understand how they work.

Time To Watch!

Boolean Logic in Java

▶ https://www.youtube.com/watch?v=npMQkQ2oCAU *(YouTube video)*

5.1.1. Boolean Values

The simplest way to use Boolean in Java is with the keywords `true` and `false`. These can be assigned to variables of type `boolean`:

```
boolean isTimAmazing = true;
boolean isClassBoring = false;
```

5.1.2. Relational Operations

Also known as *comparison operations*, relational operations are expressions that compare two values. You will remember them from math class. *Is* `x` *greater than* `y`*? Is* `a` *less than or equal to* `10`*? Is* `c` *equal to* `d`*?* Relational operators, like arithmetic operators, are *binary* operators that require two operands. In other words, we need two values to compare. In our math class, we could draw symbols that aren't on our keyboard, like a ≥ for "greater than or equal to." In Java, where we have to stick with stuff on our keyboard, we use the following symbols:

Table 3. Java relational operators

Operator	Description
`==`	Equality (checks if two values are equal)
`!=`	Inequality (checks if two values are not equal)
`<`	Less-than
`>`	Greater-than
`<=`	Less-than-or-equal-to
`>=`	Greater-than-or-equal-to operator.

A relational operation will always evaluate to either `true` or `false`. We can store that result in a `boolean` variable, as shown below, and we'll also learn how to use these expressions in `if` statements later in this chapter.

```
boolean canBuyAlcohol = age >= 21;
boolean isNegative = number < 0;
```

These assignment statements work like any other: the expression to the right of the equals symbol is evaluated, and the result is stored in the variable on the left.

5.1.3. Testing a String for Equality

In Java, we can't use the `==` operator to compare two strings. The following code will not work as we might expect:

Example of an incorrect `String` comparison.

```
// Should output "true", but (sometimes) doesn't
String name = "Paul McCartney";
System.out.print("Is this person Paul McCartney?");
System.out.println(name == "Paul McCartney");   // DON'T DO THIS!
```

The `==` operator does not compare the contents of the strings, but rather the memory addresses where the strings are stored. Due to the nuances of how the Java runtime handles `String` objects, this code will work sometimes, by coincidence, but it's not reliable.

Instead, we'll need to use the `equals()` method, which is a method of the `String` class and gets called using dot notation. This method will examine the contents of the strings and return `true` if they are the same, and `false` if they are different.

Example of a correct `String` comparison.

```
// Outputs "true"
String name = "Paul McCartney";
```

```
System.out.print("Is this person Paul McCartney? ");
System.out.println(name.equals("Paul McCartney"));
```

The `equals()` method is case-sensitive, so `"Paul McCartney"` is not the same as `"paul mccartney"`. The String class also has a `equalsIgnoreCase()` method that will compare two strings without regard to case.

5.2. `if` Statements

The `if` statement is the most basic decision-making structure in Java. It allows us to execute a block of code only if a certain condition is true.

The syntax of an `if` statement is the keyword `if`, followed by a Boolean expression in parentheses, followed by a block of code in curly braces. If the Boolean expression evaluates to `true`, the block of code will execute. If the Boolean expression evaluates to `false`, the block of code will be skipped. In either case the program will continue executing the next line of code after the `if`-block.

Example of an `if` Statement.

```
int age = 20;
if (age < 21) {
    System.out.println("You can't buy alcohol.");
}
System.out.println("Keep that in mind when you go to the store!");
```

In this example, the `if` statement checks if the variable `age` is less than `21`. Since `20` is less than `21`, the Boolean expression evaluates to `true`, and the block of code inside the `if` statement is executed—and it prints "You can't buy alcohol." The program then continues to the next line of code, which prints "Keep that in mind when you go to the store!"

If the value of `age` were `22`, the Boolean expression would evaluate to `false`, and the block of code inside the `if` statement would be skipped. The program would then continue to the next line of code, which prints "Keep that in mind when you go to the store!"

Keep in mind, the parentheses after the `if` keyword can contain any Boolean expression—not just this simple example.

5.3. Adding an `else` Block

An `if` statement simply determines whether or not to execute a single block of code. If we want to choose between two blocks of code, we can add an `else` block to the `if` statement. The syntax is simple: after the `if` block, add the keyword `else`, followed by a block of code in curly braces.

Example of an `if-else` Statement.

```
int age = 20;
if (age < 21) {
    System.out.println("You can't buy alcohol.");
```

```
}
else {
    System.out.println("You can buy alcohol.");
}
System.out.println("Keep that in mind when you go to the store!");
```

An `if-else` statement will always execute one block of code or the other, but never both. Basically, it's an either-or situation.

Time To Watch!

`if` and `if-else` Statements in Java

▶ https://www.youtube.com/watch?v=YytQwS4F6fE *(YouTube video)*

File from video:

- Completed code: `MadJokes.java`

Interesting!

Java includes a shorthand form of an if-else statement called the *ternary operator*, which uses the question mark symbol. It's a useful little trick, but it can be confusing for beginners—and for the people reading our code later. We won't look at them in this course, but a web search should turn up plenty of examples if we are curious.

Assignments in my course require you to actually use if-else statements, so you shouldn't use the ternary operator in code you submit to me.

5.4. The `if-else if` Structure

The `if-else` structure is great for choosing between two blocks of code, but what if we have more than two options? To handle this, we can chain multiple `if-else` statements together.

`WeatherRecord.java` *(excerpt). Example of an* `if-else if` *statement. See GitHub for the complete file.*

```
public class WeatherRecord {
    // Fields
    private String date;
    private int highTemperature;
    private double averageWindSpeed;

    public String getTempDescription() {
        if (this.highTemperature > 90) {
            return "Hot";
        } else if (this.highTemperature > 70) {
            return "Warm";
```

```
        } else if (this.highTemperature > 50) {
            return "Cool";
        } else {
            return "Cold";
        }
    }
}
```

In this example, the `getDescription()` method will return a `String` that describes the weather based on the high temperature of the day.

- If the high temperature is greater than `90`, the method will return "Hot".
- If the high temperature is greater than `70`, the method will return "Warm".
- If the high temperature is greater than `50`, the method will return "Cool".
- If the high temperature is `50` or less, the method will return "Cold".

The code begins with the first statement, and if it evaluates to `true`, the corresponding block of code will execute. If the first statement evaluates to `false`, the program will move on to the next `else if` statement, and so on. Once a code block is executed, it will hit a `return` statement, which will exit the method and not evaluate any other blocks of code. Therefore, only one block of code will execute. If the program gets through the entire structure without finding a `true` condition, it will execute the block of code in the `else` block, if one is present.

If we're looking at that code critically, we might notice that the `else` block is not strictly necessary. We could just put the `return "Cold";` statement at the end of the method, and it would work the same way. However, that depends on the logic of the `if-else if` structure and whether or not we're using `return` statements in the blocks of code.

Interesting!

The above example shows one clever use of decisions in an object class. You'd think that a `getDescription()` method would return the value of a field called `description`, but there is no field. Instead, it just uses a decision structure to return a description based on the `highTemperature` field.

In summary, An `if-else if` structure can execute, at most, one block of code. If an `else` block is included at the end, it guarantees that exactly one block of code will execute.

Time To Watch!

`if-else if` Statements in Java

Note: This video also shows how to use `if` statements in an OOP context, specifically to help with encapsulation.

▶ https://www.youtube.com/watch?v=ismb63p4N4A *(YouTube video)*

Files from video:

- Starter code: `K12Student.java`
- Starter file: `K12StudentDemo.java`
- Completed code: `K12StudentFinished.java`
- Completed code: `K12StudentDemo.java`

5.5. Nested `if-else` Statements

If we want a block of code to execute only if two different conditions are met, we can place `if` statements inside of each other—which is called *nesting*. Nested if statements check multiple conditions in a hierarchical way: if one condition is met, it will proceed and check the next condition; if the first condition is not met, it will skip the inner `if` block.

Basic structure of a nested `if-else` statement.

```
if (condition1) {
    if (condition2) {
        // executes if both condition1 and condition2 are true
    }
    else {
        // executes if condition1 is true and condition2 is false
    }
}
else {
    // executes if condition1 is false
}
```

In the example below, the outermost if-else structure checks the high temperature of the day. The if-else structures within those blocks check the average wind speed and return an appropriate description.

`WeatherRecord.java` *(excerpt). Example of a nested `if-else` statement. See GitHub for the complete file.*

```
public String getFullDescription() {
        if (this.highTemperature > 90) {
            if (this.averageWindSpeed > 10) {
                return "Hot and Windy";
            } else {
                return "Hot";
            }
        } else if (this.highTemperature > 70) {
            if (this.averageWindSpeed > 10) {
                return "Warm and Windy";
            } else {
                return "Warm";
            }
        } else if (this.highTemperature > 50) {
```

```
            if (this.averageWindSpeed > 10) {
                return "Cool and Windy";
            } else {
                return "Cool";
            }
        } else {
            if (this.averageWindSpeed > 10) {
                return "Cold and Windy";
            } else {
                return "Cold";
            }
        }
    }
```

Time To Watch!

Nested `if-else` Statements in Java

▶ https://www.youtube.com/watch?v=bKqlmhtaKd8 *(YouTube video)*

Note: This video also shows how to use `if` statements in an OOP context, specifically to help with encapsulation.

Files from video:

- Starter code: `VinylRecord.java`
- Starter code: `VinylRecordDemo.java`
- Completed code: `VinylRecordFinished.java`
- Completed code: `VinylRecordDemoFinished.java`

5.6. Using Logical Operators

In addition to the relational operators, Java also includes logical operators we can use to make more complex Boolean expressions. A logical operator is a binary operation, so it takes two operands—but the operands are Boolean expressions instead of numbers.

Table 4. Java logical operators

Operator	Name	Description
`&&`	AND	Evaluates to `true` if **both** operands are `true`
`||`	OR	Evaluates to `true` if **either** operand is `true`
`!`	NOT	Evaluates to `true` if the operand is `false`; evaluates to `false` if the operand is `true`

These operators can be used to combine multiple Boolean expressions into a single, more complex

expression. For example, we could check if a student is eligible for a scholarship based on both their GPA (3.5 or better) and their age (younger than 25).

Example of a decision using a logical AND operation.

```
if (gpa >= 3.5 && age < 25) {
    System.out.println("You qualify for the scholarship!");
}
```

In this example, the `&&` operator is used to combine two Boolean expressions. The `if` statement will only execute the block of code if both expressions are `true`.

Often, the logic we create using an AND operation can be implemented using nested if-else statements, and vice versa.

The OR operation is similar, but only one of the expressions needs to be `true` for the entire expression to be `true`.

Example of a decision using a logical OR operation.

```
boolean isTimAmazing = false;
boolean isClassFun = true;

if (isTimAmazing || isClassFun) {
    System.out.println("You should take this class!");
}
```

Both operands in an AND or OR operation have to be complete Boolean expressions. Put another way, each side of the `&&` or `||` operator must be able to evaluate to `true` or `false` on its own. The following code is a very common beginner mistake and will **not** compile:

```
if (percentage >= 80 && < 90) { ··· }
```

This reads like "if the percentage is greater than or equal to 80 and less than 90," but the second part of the expression is not a complete Boolean expression. We need to include the variable name and the comparison operator on both sides of the `&&` operator.

The NOT operation is a little different, as it only takes one operand (making it a *unary* operator_, if you're nerdy about words, like I am). It simply inverts the value of the operand. If the operand is `true`, the NOT operation will evaluate to `false`. If the operand is `false`, the NOT operation will evaluate to `true`.

Example of a decision using a logical NOT operation.

```
boolean isTimAmazing = false;

if (!isTimAmazing) {
    System.out.println("At least his mom still loves him!");
}
```

5.6.1. Range Checking

There are a lot of situations where we might need to combine multiple conditions to make a decision, but one of the most common is *range checking*. Range checking means we want to see if a value is within a certain range.

A common example of range checking is to convert a percentage grade to a letter grade.

Example of range checking using logical operators.

```
public String getLetterGrade(int percentage) {
    if (percentage >= 90 && percentage <= 100) {
        return "A";
    } else if (percentage >= 80 && percentage < 90) {
        return "B";
    } else if (percentage >= 70 && percentage < 80) {
        return "C";
    } else if (percentage >= 60 && percentage < 70) {
        return "D";
    } else if (percentage >= 0 && percentage < 60) {
        return "F";
    } else {
        return "Invalid percentage";
    }
}
```

The AND operator `&&` used in this example means that in order to return `"B"`, for example, the percentage must be greater than or equal to `80` *and* less than `90`. If either of those conditions is not met, the program will move on to the next `else if` statement.

5.7. `switch` Statements

Java includes a structure called a `switch` statement that can be used to choose between multiple options. It is essentially another way to write an `if-else if` structure, but it can be more readable and easier to write in some situations. I generally consider `switch` structures to be optional—you can complete all of the assignments in this course without using them—but they are a useful tool to have in your programming toolbox. And since you see them often in code written by others, it's good to know how they work.

The basic structure of a `switch` statement is as follows:

```
switch (expression) {
    case value1:
        // Code to be executed if expression equals value1
        break;
    case value2:
        // Code to be executed if expression equals value2
        break;
    case value3:
```

```
        // Code to be executed if expression equals value3
        break;
    default:
        // Code to be executed if expression doesn't match any case
}
```

The `expression` in the parentheses after the `switch` keyword is evaluated, and then the program will jump to the `case` that matches the value of the expression. If there is no match, the program will execute the `default` block, if it is present.

The `break` statement is used to exit the `switch` block due to a behavior of `switch` that can be confusing to beginners, known as *fall-through*. If we don't include a `break` statement at the end of a `case` block, the program will continue executing the code in the next `case` block, even if the value of the expression doesn't match the `case`. This can be useful in some situations, but it's generally not what you want, so you'll usually see a `break` statement at the end of each `case` block.

Example of a `switch` statement.

```
public void trafficInstructions(String lightColor) {
    switch (lightColor) {
        case "red":
            System.out.println("Stop!");
            break;
        case "yellow":
            System.out.println("Slow down!");
            break;
        case "green":
            System.out.println("Go!");
            break;
        default:
            System.out.println("Invalid light color.");
    }
}
```

5.8. Solution Walkthrough

In "solution walkthrough" videos, I give a problem/prompt that is similar to the kinds of work I assign, and then I record myself writing a solution. It's not absolutely mandatory to watch this video, but students report that these videos are particularly helpful.

Time To Watch!

Decisions - Java Solution Walkthrough

▶ https://www.youtube.com/watch?v=7khBJXDXirs *(YouTube video)*

Solution file from video:

Not yet available!

Check Yourself Before You Wreck Yourself (on the assignments)

Can you answer these questions?

1. Explain what Boolean expressions are and how they are used to make decisions in Java.
2. Explain the difference between a relational operator and a logical operator.
3. What is the difference between an `if` statement and an `if-else` statement?
4. How can you write code that runs one code block from multiple options?

Sample answers provided in Stuff That's Tacked On The End.

6. 0110 - Loops

Help Make These Materials Better!

I am actively working to complete and revise this eBook and the accompanying videos. Please consider using the following link to provide feedback and notify me of typos, mistakes, and suggestions for either the eBook or videos:

CIS150AB Course Materials Feedback (Google Form)

What's the Point?

- Understand the purpose of loops
- Understand the difference between definite and indefinite loops
- Use `while` loops
- Use `do-while` loops
- Use `for` loops

Source code examples from this chapter and associated videos are available on GitHub.

A computer is designed to execute a series of instructions, in order, very quickly. Now that we understand how to use Boolean expressions to control the flow of our programs, we can use that concept for a programming structure that really unlocks the power of the computer: *repetition*.

Many of the tasks we think of as "computer tasks" are repetitive in nature. Processing data often involves performing the same operation on each piece of information, one after the other, until everything is processed. A computer game has to draw the screen, check for collisions, check what the user is doing with the gamepad, and update the positions of all the objects on the screen, over and over again, until the game ends. When we use a search engine on the web, that search engine has scanned the web one page at a time, indexing what it finds and then moving on to the next page, until it's scanned all of the pages in its database.

Computers are great for these kinds of tasks: they don't get bored, they don't get tired, and they don't make mistakes—other than the mistakes we make when programming them!

To create the kind of repetition that leverages this processing power, programmers use *loops*. A loop is a structure that repeats a block of code as long as a certain condition is true. Each time the loop repeats, it's called an *iteration*.

Time To Watch!

Introduction to Loops

▶ https://www.youtube.com/watch?v=eq9oHVazmZI *(YouTube video)*

6.1. `while` Loops

The first structure Java provides for creating loops is the `while` loop. The `while` loop repeats a block of code as long as a certain condition is true. The condition is checked before the block of code is executed, so the block of code might not execute at all if the condition is false the first time it's checked.

A `while` loop is exactly like an `if` statement: a Boolean expression is checked, and if it's true, the block of code is executed. The only difference is that, after the block of code is executed, the program jumps back to the beginning of the loop and checks the condition again. If the condition is still true, the block of code is executed again.

Example of a `while` loop

```
int count = 0;
while (count < 10) {
    System.out.println("Count is: " + count);
    count++;
}
```

In this example, the Boolean expression `count < 10` is checked; if it's true, the block of code is executed. The block of code prints out the value of `count`, then increments `count` by 1. The program then jumps back to the beginning of the loop and checks the condition again. This process repeats until the expression `count < 10` is false.

The result of this loop is that the program prints out the value of `count` 10 times, starting with 0 and ending with 9. It is a definite loop because, before the loop starts, it's already known that the loop

will repeat 10 times.

6.1.1. Infinite Loops

One thing to be careful of when using a `while` loop is the possibility of creating an *infinite loop*. An infinite loop is a loop that repeats forever, because the condition that determines whether the loop should repeat is never false. Infinite loops are a common mistake for new programmers, and they can cause our program to hang or crash.

Example of an infinite loop

```
int count = 0;
while (count < 10) {
    System.out.println("Count is: " + count);
}
```

In this example, we've forgotten to increment `count` by 1 after printing it out. `count` will always be `0`, so the condition `count < 10` will always be true. The program will print out "Count is: 0" over and over again, forever.

Your IDE will have a way to stop the program if it's stuck in an infinite loop. Often, this is a square button—essentially the symbol for "stop". In VS Code, you can also kill the terminal window that's running the program by clicking the trash can icon in the terminal window. Finally, in many cases you can press `Ctrl+C` to stop the program.

The simplest infinite loop we can create in Java is:

```
while (true) {
    // hey, there's nothing to stop this loop!
}
```

This loop will repeat forever, because the condition `true` is always true. Some programmers use this to start a loop and then use an if statement to break out of the loop when a certain condition is met, but that's not really an infinite loop—it just puts the boolean expression that controls the loop in an `if` statement instead of the `while` statement. I personally consider it less readable than using a clear condition in the `while` statement, so I don't write loops like that.

Time To Watch!

`while` Loops in Java [COMING SOON!]

▶ https://www.youtube.com/watch?v=wMbpslAjDDc *(YouTube video)*

Files from video:

- Completed code: `WhileLoopDemo1.java`

- Completed code: `WhileLoopDemo2.java`
- Completed code: `WhileLoopDemo3.java`

6.2. `do-while` Loops

As we've seen, a `while` loop checks the condition before executing the block of code. A `do-while` loop is similar, but it checks the condition **after** executing the block of code: *run first, then check*. This means that the block of code will always execute at least once. Other than that, a `do-while` loop is exactly the same as a `while` loop.

Example of a `do-while` loop

```
int count = 0;
do {
    System.out.println("Count is: " + count);
    count++;
} while (count < 10);
```

This is the same loop as the `while` loop we looked at earlier, but the condition is checked after the block of code is executed. The `while` statement is at the end of the loop; the `do` statement at the beginning indicates the block of code that should iterate.

Time To Watch!

`do-while` Loops in Java [COMING SOON!]

https://youtu.be/urwIMiEqGBw

▶ https://www.youtube.com/watch?v=urwIMiEqGBw *(YouTube video)*

Files from video:

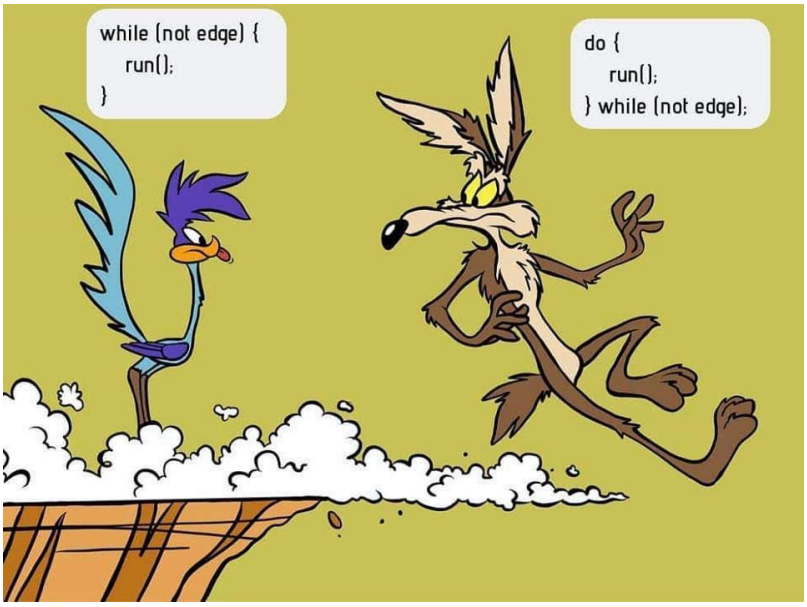
- Completed code: `DoWhileLoopDemo1.java`
- Completed code: `DoWhileLoopDemo2.java`

6.2.1. Choosing Between `while` and `do-while` Loops

Both `while` and `do-while` loops work well for *indefinite loops* (though they can be used for definite loops as well). There's nothing in the structure of these loops that requires a counter or other control variable, so they can be used for loops that repeat until a certain condition is met, however many iterations that requires.

In many cases, it doesn't matter whether we use a `while` or a `do-while` loop. We really can use either one to create the same loop. However, in some cases, one might be a better choice than the other. The simple rule of thumb for now is: if we need to guarantee that the block of code will execute at least once, we should use a `do-while` loop; if we need to check the condition before executing the

block of code, we should use a `while` loop.



6.3. Input Validation with Loops

A common use for indefinite loops is *input validation*. Input validation is the process of checking the data that a user inputs into a program to make sure it's valid. For example, if a program displays a menu with three options, a loop could keep asking for a selection until the user enters one of the three choices.

There are more advanced techniques we'll eventually use for input validation, but indefinite loops are a simple and effective way to make sure the user's input is what we expect.

Example of input validation with a `do-while` loop

```
import java.util.Scanner;

public class InputValidation {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int choice;

        do {
            System.out.println("Choose an option:");
            System.out.println("1. Check balance");
            System.out.println("2. Deposit");
            System.out.println("3. Withdraw");
            System.out.println("4. Exit");
            System.out.print("Your choice: ");
            choice = input.nextInt();
        } while (choice < 1 || choice > 4);

        // Rest of the code...
    }
}
```

This snippet of a program will keep displaying the menu and asking for the user's choice until the user enters a number between 1 and 4.

6.4. `for` Loops

Definite loops are really common, especially when we learn about things like *arrays* later on, so Java provides a keyword that allows for a compact way to create that kind of loop: the `for` loop. The syntax of a `for` loop can be a little intimidating for new coders, but it really just combines into one line of code all three of the pieces we need for a loop: initializing a counter, checking the counter, and changing the counter.

`for` *loop syntax*

```
for (int count = 0; count < 10; count++) {
    System.out.println("Count is: " + count);
}
```

The `for` loop has three parts, separated by semicolons: 1. Initialize a counter. *Example:* `int count = 0`. 2. A Boolean expression that determines if the loop should repeat. *Example:* `count < 10`. 3. Change the counter at the end of the iteration. *Example:* `count++`.

Once you get the hang of the syntax, the `for` loop is a really convenient way to create a definite loop.

Time To Watch!

`for` Loops in Java [COMING SOON!]

▶ https://www.youtube.com/watch?v=uRoLVgHkWbA *(YouTube video)*

File from video: * Completed code: `ForLoopDemo1.java`

6.5. OPTIONAL: `break` and `continue` Statements

I believe that `while`, `do-while`, and `for` loops, when written with clear Boolean expressions, are the most readable loops, and any loop a coder will need in their career can be written with those structures. A well-written loop will execute the block of code as many times as necessary, and then stop when the condition is false, without any additional help from the programmer. However, Java provides two statements that can be used to control the flow of a loop: the `break` statement and the `continue` statement.

Since they aren't necessary for writing loops, I consider them optional: none of my assignments or quiz questions will require you to know them.

The `break` statement is used to exit a loop early. When the `break` statement is executed, the loop stops, and the program continues with the next statement after the loop; think of it as a `return` statement for a loop (except that it can't pass a value anywhere). Some programmers use `break` when they need to get out of a loop before the controlling condition is false. My own opinion is that

this is a sign the controlling condition should be rethought, but because you're likely to see `break` in other people's code, I think it's important to know about it.

The `continue` statement is used to skip the rest of the block of code in a loop and jump back to the beginning of the loop. When the `continue` statement is executed, the program stops executing the loop's block of code, and jumps to the Boolean expression that controls the loop to see if it should run again.

Time To Watch!

`break` and `continue` in Java [COMING SOON!]

▶ https://www.youtube.com/watch?v=gc4D-7UhxR0 *(YouTube video)*

- Completed code: `BreakDemo1.java`
- Completed code: `BreakDemo2.java`
- Completed code: `EvilDemo.java`

6.6. A Word About Nested Loops

We can put a loop inside another loop, and that's called a *nested loop*. They are useful in some situations, and studying them can improve our ability to think through loop-based algorithms.

However, they are beyond the scope of this course, which focuses on the fundamentals of object-oriented programming.

If you want to explore them, the textbook addresses nested loops in section 6.6 (page 220), and there are many great resources available on the web and YouTube.

6.7. JUST FOR FUN: Recursion

Content labelled as **Just for Fun** is not required for the course, but is included for students who are interested in learning more about the topic. If you're struggling to learn the material in this course, please skip this—maybe you can return when you have more time.

I plan to record a brief video on recursion, but it's not a high priority. This topic is not covered in the textbook, so if you *really* can't wait, search for the topic on the internet.

6.8. Solution Walkthroughs

In "solution walkthrough" videos, I give a problem/prompt that is similar to the kinds of work I assign, and then I record myself writing a solution. It's not absolutely mandatory to watch this video, but students report that these videos are particularly helpful.

Time To Watch!

Loops Practice Solution A (Performing a Calculation)

▶ https://www.youtube.com/watch?v=9PCFX5ttJK8 *(YouTube video)*

Solution file from video:

Not yet available!

Loops Practice Solution B (Data Validation)

▶ https://www.youtube.com/watch?v=q91PC_M042A *(YouTube video)*

Solution file from video:

Not yet available!

Check Yourself Before You Wreck Yourself (on the assignments)

Can you answer these questions?

1. Describe the difference between a while loop and a do-while loop.
2. What is a control variable, and how is it used in loops?
3. Give an example of an indefinite loop.

Sample answers provided in Stuff That's Tacked On The End.

7. 0111 - Debugging and Generative AI

Help Make These Materials Better!

I am actively working to complete and revise this eBook and the accompanying videos. Please consider using the following link to provide feedback and notify me of typos, mistakes, and suggestions for either the eBook or videos:

CIS150AB Course Materials Feedback (Google Form)

What's the Point?

- Distinguish between compile-time and runtime errors
- Learn some strategies for debugging your code
- Use the debugging tools available in your IDE
- Understand the basics of generative AI in coding
- Use a generative AI tool to generate source code

Source code examples from this chapter and associated videos are available on GitHub.

7.1. Debugging

A *bug* is an error in our code that causes it to behave in an unexpected way. This is different from a syntax error, like forgetting a semicolon or misspelling a keyword—those kinds of mistakes will prevent our code from compiling, so we can't even run it. A bug is when our code runs, but it doesn't do what we want it to do.

Bugs are frustrating, but they are a part of programming. Like Thanos, they are inevitable. And like Thanos, we can use ludicrous time traveling to fix (decapitate?) our bugs. Well, we can't do that last part, but we can fix them in the present; and hopefully, we can do it without too much frustration.

Finding and fixing bugs is called *debugging*, and without realizing it, we've been developing our own strategies for debugging since we started writing code. But as our code becomes more complex, we'll benefit from a more systematic approach to debugging—and we'll want to take advantage of the tools available in our IDE to help us.

7.2. Types of Errors

We categorize bugs into two general types:

Compile-time errors

Errors that prevent the compiler from fully processing our source code. These are generally the result of incorrect syntax—in other words, breaking the rules of the language.

Runtime errors

Errors in which our code compiles, but it does not execute as intended. Crashes are obvious runtime errors, but making an incorrect calculation is also an example of a runtime error.

Fixing compile-time errors is just a matter of looking over our code and correcting the mistake. That's not always as easy as it sounds—especially for beginners—but at least the compiler and/or our IDE can give us feedback about what and where the mistake is.

Runtime errors can be especially frustrating, especially since we can't always tell at what point the

actual error is occurring. Did I make the mistake at the start of the program when I calculated the answer, is the mistake at the very end where I output it? Or did I do something in the middle that accidentally changed the result? Who knows! And if you're like me, you might have done all three…

Time To Watch!

Intro to Debugging

▶ https://www.youtube.com/watch?v=jcslOC6-cbk *(YouTube video)*

7.3. Debugging Tools in Our IDE

Most modern IDEs have built-in tools to help us debug our code. These tools allow us to monitor the variables in our program, step through our code line-by-line, and set breakpoints to pause our program at a specific point. The more we know about programming, the more useful these tools become. But even as beginners, these tools can be helpful in finding and fixing bugs in our code.

The most common debugging tools we'll use are: Watch window:: A window that displays the values of variables in our program. Breakpoints:: A marker that tells our program to pause at a specific point in our code. Step Into:: A command that tells our program to execute the next line of code. Step Over:: A command that tells our program to execute the next line of code, but not to step into any methods that are called. Step Out:: A command that tells our program to execute the rest of the current method and then pause. Resume:: A command that tells our program to continue running until it hits the next breakpoint.

These functions are available in most IDEs, but the specifics of how they work can vary. Even if you're not using Visual Studio Code, the concepts are the same, so you should be able to apply what you learn here to your IDE of choice.

Time To Watch!

Debugging in Java with VS Code

▶ https://www.youtube.com/watch?v=t4whOTYQ10k *(YouTube video)*

Files from video:

- Starter code: `MethodDebugging.java`
- Completed code: `MethodDebuggingFinal.java`

The first Lab Assignment in Canvas can be completed using what we've covered to this point. You might choose to complete that work now, then move onto the next section—which you'll need for the second Lab Assignment.

7.4. Generative AI in Coding

The AI content is under construction. Check back soon for updates!

Generative Artificial Intelligence is a type of AI that can generate new content based on existing data. Tools like ChatGPT and Copilot are well-known examples of generative AI. We can ask a generative AI tool to write song lyrics about Java programming, for example, and it will produce a new song for us.

Artificial Intelligence is a broad field, and the topic is well beyond the scope of this course. But generative AI has important implications for the field of software development, so we'll focus on just that aspect of the technology.

7.4.1. The Role of Generative AI in Education

Like many teachers, I'm actively wrestling with the role of generative AI in a coding course. I have many ethical concerns about AI in general, and about generative AI to create code in particular. Simply put, these tools can be used in ways that are indisputably unethical. Most obviously, students can easily use these tools to generate code that is then submitted as the student's own work.

I would like to think it goes without saying, but based on the number of students who deny this, I guess I need to say it: using AI to generate code that you submit as your own is cheating. It is no different than copying someone else's code and submitting it as your own.

It presents a real challenge for teachers trying to assess student learning, and a real temptation for students struggling to learn—and able to simply let AI do the work for them.

But the impact this technology has had—and will continue to have—on the field of software development is undeniable, and I would be doing you a disservice if we didn't learn a little about it.

7.4.2. The Promise of Generative AI in Coding

What does this AI revolution look like for coders? What does it mean to someone learning to code? How does it change the job outlook for someone considering a career in software development?

The short answer is: I don't know. The longer answer is: I don't know, but I'm excited to find out.

For now, we can look at what generative AI can do for us today. Here are a few of the current and near-future applications of generative AI in coding:

- **Code completion:** Many IDEs already have code completion features that suggest code as you type. Generative AI can take this a step further by suggesting entire lines of code, or even entire methods.
- **Code generation:** Generative AI can generate code based on a description of what you want the code to do.
- **Code refactoring:** *Refactoring* is the process of rewriting code without changing the task the code performs. Once we get code working, we can refactor it to make it more efficient, more

secure, or more maintainable. In other words, we can make it better. AI can look at our existing code and recommend changes that make it better.

- **Debugging:** Generative AI can help us find and fix bugs in our code—often, before we even run it.
- **Documentation:** Writing good documentation is an important part of software development, but many programmers hate doing it. Generative AI can help us write documentation that is clear, concise, and accurate.

And that's just a few of the obvious applications of generative AI in coding.

What does that mean for the coding profession? All I can do is guess, but here are some things I **hope** AI does for us:

- **Better software:** If AI can help us write better code, that should lead to better software.
- **Faster development and update cycles:** Again, if AI makes us more efficient, we should be able to develop and update software faster.
- **Improved security:** Hopefully, AI will help us identify and address security vulnerabilities in our code.
- **More time for high-level effort:** If AI can take on some of the more tedious and boring coding tasks, that should free up mental bandwidth to focus on the more interesting and creative.

My most optimistic hope is that AI will free up programmers to focus on creative applications and problem-solving—and ultimately allow us to create software that improves the world around us.

7.4.3. Potential Negative Impacts of AI on Coding

However, I worry that AI will lead to some negative impacts on the programming profession, as well. Again, I can only guess, but here are some things I **fear** might happen due to AI:

- **Decreased job opportunities:** If AI can write code faster and more accurately than humans, that could lead to fewer job opportunities for human programmers. I think this will be especially true for junior programmers, whose workload will be most easily automated.
- **Loss of institutional knowledge:** Every programming team relies on veteran coders who have been around and have a deep understanding of the codebase. When there's a question about what a module does, or why an algorithm was implemented a certain way, those veterans are the ones who have the answers. If AI is generating code, we may lose that institutional knowledge.
- **Decreased quality of entry-level programmers:** If AI can generate code for us, it's possible that we'll see a decrease in the quality of entry-level programmers. People who rely heavily on AI while learning to code may not develop the same problem-solving skills as those who learn to code without AI.

7.4.4. What Does That All Mean for You?

I don't know. As excited as I am to see AI reach this tipping point in software development, it's kind of a scary time to be a programming teacher. Until recently, I've always

Coders who rely heavily on artificial intelligence tools to solve problems may be able to pass themselves off as more skilled than they actually are—indeed, they might even **believe** themselves to be more skilled than they actually are. But such coders will be less skilled at testing, debugging and maintaining code, and that could lead to a decrease in the quality of software. Putting code into production (releasing it in software that people actually use) without understanding it well enough to regiorously test it is a recipe for disaster.

I think there is still a bright future for programmers, but the landscape will be different. Coders will need to have a deeper understanding of the code they write, and they will need to be able to solve problems that AI can't. They'll need to be able to evaluate the code that AI generates, and they'll need to be able to maintain and update that code. The role of software architect—the person who designs the overall structure of a software project—will become even more important, as will the role of the software tester.

Students will need to have the discipline to learn to code without relying on AI tools, and they'll need to develop the problem-solving skills that AI can't provide. Testing and debugging skills will be more important than ever, and those can really only be developed through practice—and that practice comes from writing code, testing it to find errors, and fixing problems.

7.4.5. Using Generative AI Tools

Online tools like ChatGPT and Microsoft's Copilot can help you generate code without any specialized software. Simply enter a description of what you want the code to do, and the AI will generate code for you. You can then copy and paste that code into your IDE and work with it from there.

But IDEs are also beginning to integrate generative AI tools focused on coding. For example, Visual Studio Code now includes a feature called **GitHub Copilot** that has been trained on billions of lines of code. The user interface provides a chat window where you can describe what you want the code to do, and Copilot will generate code for you. You can choose to accept that suggestions, or you can refine your prompt to get a different suggestion.

Additionally, Copilot can generate code as you type. It will analyze the code you've already written and suggest the next line of code. You can accept that suggestion, or you can ignore it and keep typing. It essentially acts as a very advanced code completion tool—or like the "autocomplete" feature on your phone's keyboard, but for code.

At this time, GitHub Copilot requires a subscription, but you can use it for free for a limited number of lines of code; and students get an expanded free tier.

And yes, I used Copilot to help me prepare the content on AI in this chapter...

Time To Watch!

Intro to GitHub Copilot in VS Code

▶ https://www.youtube.com/watch?v=gHuJgnpG7pI *(YouTube video)*

Check Yourself Before You Wreck Yourself (on the assignments)

Can you answer these questions?

1. What is the difference between a compile-time error and a runtime error?
2. How can using output statements help in debugging a program?
3. What are some strategies you can use when you're frustrated by a bug in your code?

Sample answers provided in Stuff That's Tacked On The End.

8. 1000 - Arrays

Help Make These Materials Better!

I am actively working to complete and revise this eBook and the accompanying videos. Please consider using the following link to provide feedback and notify me of typos, mistakes, and suggestions for either the eBook or videos:

CIS150AB Course Materials Feedback (Google Form)

What's the Point?

- Understand the purpose of arrays
- Create and use arrays
- Write loops to iterate through arrays

Source code examples from this chapter and associated videos are available on GitHub.

8.1. Arrays and Indexes

The variables we've used to this point can only store one piece of data at a time, and sometimes we need to store more than that. For example, we might need to store a list of items to buy at the grocery store. I could create a variable for each item, but that would be a lot of variables to keep track of. Arrays give us a way to store multiple pieces of data in a single collection.

Each individual value in an array can be referred to as an *element*. Each element in an array has a unique *index* that tells us where it is in the array. The number of elements in an array is called the *length* of the array. The first element in an array has an index of 0, the second element has an index of 1, and so on. The last element in an array is always the length of the array minus one. If our

grocery list has 5 items, the indexes of the items would be 0, 1, 2, 3, and 4.

Time to Watch!

Introduction to Arrays

▶ https://www.youtube.com/watch?v=08IJJod4O3o *(YouTube video)*

8.2. Defining and Using Arrays

In Java, arrays are objects, so they have to be created with the `new` keyword, and array identifiers use square brackets [] to specify array elements.

Any time you see square brackets in Java, you're dealing with an array.

The following example shows how an array is declared and initialized:

```
String[] groceryItems = new String[5];
```

This code creates an array of `String` objects called `groceryItems` that can hold 5 elements.

Once we have an array, we can assign values to its elements using the index, which is placed in square brackets after the array name:

```
groceryItems[0] = "milk";
groceryItems[1] = "eggs";
groceryItems[2] = "bread";
groceryItems[3] = "butter";
groceryItems[4] = "cheese";
```

Each element in the array is assigned a value, and the index is used to specify which element is being assigned.

Once we have values in an array, we can access them using the index. The element is just like any other variable, so we can use it in expressions, pass it to methods, and so on.

```
System.out.println("The first item on the list is " + groceryItems[0]);
System.out.println("The last item on the list is " + groceryItems[4]);
groceryItems[2] = "bananas";
System.out.println("Bread has been replaced with " + groceryItems[2]);
```

When using arrays, we have to be careful to stay within the *bounds* of the array. If we try to access an element that doesn't exist, we'll get an `ArrayIndexOutOfBoundsException`. This is a runtime exception, so it won't be caught by the compiler, and it will cause our program to crash. In all arrays, any index less than 0 or greater than or equal to the length of the array is out of bounds. In

our grocery list example, any index greater than 4 would be out of bounds.

Time to Watch!

Array Syntax in Java

▶ https://www.youtube.com/watch?v=RBY8zz7f-bU *(YouTube video)*

Files from video:

- Starter code: `BasicSyntax.java`
- Completed code: `BasicSyntaxFinished.java`

8.3. Traversing Arrays

Working with individual elements in an array can be useful, but to really see the power of arrays, you need loops. With a loop, we can easily move through an array and perform some task or operation on each element. For example, if we have an array of quiz scores, we could use a loop to add up all the scores and calculate the average.

When we use a loop to go through an array, we are *traversing* the array. We can use a traversal to put values into an array (which we call *populating* the array), to modify values in an array, or to read values from an array. We can also use a traversal to search for something within an array.

Most of the time, we'll use a simple `for` loop to traverse an array. The syntax for a `for` traversal is:

Example of a `for` loop to traverse an array

```
int scores[] = {90, 85, 95, 88, 92};
int sum = 0;
for (int i = 0; i < scores.length; i++) {
    sum += scores[i];
}
System.out.println("The sum of the scores is " + sum);
```

In this example, we have an array of quiz scores. We use a `for` loop to go through each element of the array and add it to the `sum` variable. Using a variable like `sum` to accumulate a value is a common pattern in programming, and we refer to such as variable as an *accumulator*. Notice that we declare and initialize it before the loop, and then we update it inside the loop. If we declare it inside the loop, it will be reset to zero each time the loop runs, and we won't get the correct sum. It also wouldn't be accessible outside of the loop due to its scope.

At this point, clever students point out that we could get the sum of the scores without using a loop at all, using the `Arrays` class from the `java.util` package. Sure, but since we're learning about loops, that would defeat the point. My assignments for this chapter assess your ability to write loops, so you won't get any points for code that doesn't at least try to traverse the array.

In the example, notice that we use the `length` property of the array to set the loop condition. That way, this same loop will work for any array of any size. If we hard-coded the size of the array into the loop, we would have to change our code every time we changed the size of the array.

Example of a hard-coded loop

```
int scores[] = {90, 85, 95, 88, 92};
int sum = 0;
for (int i = 0; i < 5; i++) { // Uh oh! What if we add another score?
    sum += scores[i];
}
System.out.println("The sum of the scores is " + sum);
```

Time to Watch!

Loops and Arrays in Java

▶ https://www.youtube.com/watch?v=s48mWBeLkuY *(YouTube video)*

Files from video:

- Starter code: `ArrayLoops.java`
- Completed code: `ArrayLoopsFinished.java`

The Lab Assignments in Canvas can be completed using what we've covered to this point. You might choose to complete that work now, then move onto the rest of the chapter—which you'll need for the Programming Project.

8.4. Putting Objects in Arrays

In Java, an array can hold a primitive type, like an `int`, or an object. We've been using arrays of Strings, which are objects, but students sometimes don't realize that they can also create arrays of objects they define themselves.

If we were to define a GroceryItem class with fields for the name and the aisle where it's located, we could create an array of GroceryItem objects.

```
GroceryItem[] groceryItems = new GroceryItem[5];
groceryItems[0] = new GroceryItem("milk", 4);
groceryItems[1] = new GroceryItem("eggs", 9);
groceryItems[2] = new GroceryItem("bread", 7);
groceryItems[3] = new GroceryItem("butter", 4);
groceryItems[4] = new GroceryItem("cheese", 4);
```

Putting objects in an array expands the possibilities of what we can do with arrays. Our array can hold multiple objects, and each object can have multiple fields—this allows arrays to manage large

amounts of data in a single collection.

Time to Watch!

Putting Objects in Arrays

▶ https://www.youtube.com/watch?v=S2ufDoBKWx4 *(YouTube video)*

Files from video:

- Starter code: // * Starter code: `Dog.java`
- Completed code: `ArrayOfObjects.java`

8.5. "For-Each" Loops

Because array traversal is such a common task, Java provides a special kind of loop that makes it easier to traverse an array. It is officially known as an *enhanced for loop*, but it is often called a "for-each loop" because it goes through each element in the array. A for-each loop is easy to write, and because it handles index management for us, it is less error-prone than a traditional for loop. However, it is less flexible than a traditional for loop: we can't use it when we need to know the index of the element we're working with, or when we need to move through the array in a different order.

The for-each loop is optional. You can always use a traditional for loop to traverse an array, and you'll need to know how to do that for the assignments in this chapter.

Time to Watch!

"For-Each" Loops in Java

▶ https://www.youtube.com/watch?v=GXVSXNhX2O0 *(YouTube video)*

Files from video:

- Starter code: `ForEachLoop.java`
- Completed code: `ForEachLoopFinished.java`

Check Yourself Before You Wreck Yourself (on the assignments)

Can you answer these questions?

1. What is an array and how does it differ from a single variable?
2. What is an ArrayIndexOutOfBoundsException and when does it occur?
3. How can loops be used to traverse an array? Provide an example of a for loop that sums the elements of an integer array.

Sample answers provided in Stuff That's Tacked On The End.

9. 1001 - Inheritance

Help Make These Materials Better!

I am actively working to complete and revise this eBook and the accompanying videos. Please consider using the following link to provide feedback and notify me of typos, mistakes, and suggestions for either the eBook or videos:

CIS150AB Course Materials Feedback (Google Form)

What's the Point?

- Design classes using inheritance
- Use the `extends` keyword to create a subclass
- Override methods in a subclass
- Implement constructors in a subclass
- Use polymorphism to create arrays of subclass objects

Source code examples from this chapter and associated videos are available on GitHub.

9.1. Inheritance Overview

Inheritance is a core object-oriented programming concept that allows us to reuse code from one class—known as a *parent class*--in a new class, which we call a *child class*. In the same way that people can inherit traits from their parents, child classes inherit the attributes and behaviors of their parent classes. This means that a child class automatically has all of the fields and methods of a parent class, and just like people, a child class can also have its own unique fields and methods.

Though there are other ways to think about inheritance, I encourage students to think of it as a way to create a class that is a more specific version of another class. For example, if we have a class called `Athlete`, we can create a child class called `FootballPlayer`. A `FootballPlayer` is a type of `Athlete`, and there might be other types of athletes that are not football players: gymnasts, tennis

players, and so on.

We can think of inheritance as an "is-a" relationship: a `House` *is a* `Building`, for example. When deciding whether to use inheritance, ask yourself if the child class *is a* type of the parent class. We call this the *is-a test*.

When we design classes in UML, we use an arrow to indicate inheritance. The arrow points from the child class to the parent class.

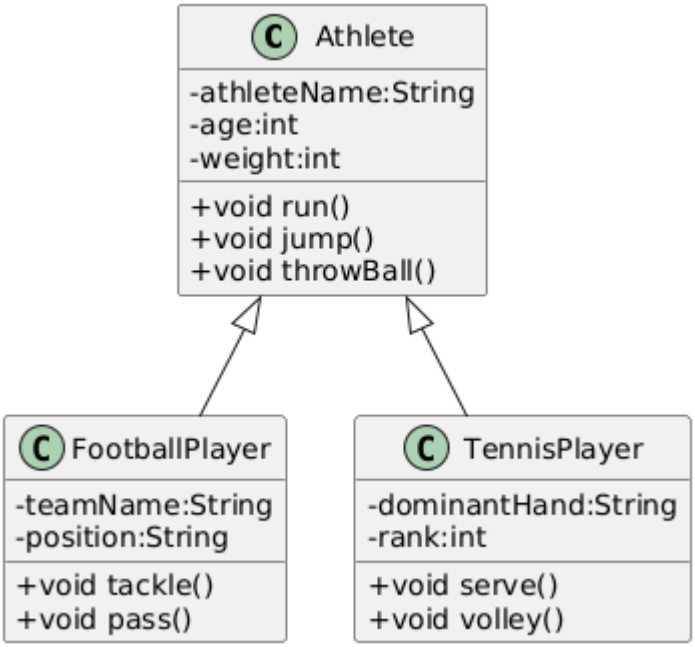


Figure 8. Example of an `Athlete` parent class with two child classes, `FootballPlayer` and `TennisPlayer`

In this example, the `FootballPlayer` class inherits the fields and methods of the `Athlete` class, so we don't have to rewrite those, and then it adds on its own unique fields and methods, like `teamName` and `position`.

Time To Watch!

Introduction to Inheritance

▶ https://www.youtube.com/watch?v=LXKXWT0y3oQ *(YouTube video)*

9.2. Implementing Inheritance

When we move from a design in UML to writing actual code—which we refer to as *implementation* --we use different terms to describe the relationships between classes.

A parent class becomes a *superclass*, and a child class becomes a *subclass*. You may also hear the terms *base class* and *derived class* to describe the same relationship, especially in other programming languages.

In Java, we use the `extends` keyword to indicate that one class is a subclass of another class. Here's

an example of a `FootballPlayer` class that extends the `Athlete` class:

```
public class Athlete {
    private String athleteName;
    private int age;
    private int weight;

    public void run() {
        System.out.println("Running!");
    }

    public void jump() {
        System.out.println("Jumping!");
    }

    public void throwBall() {
        System.out.println("Throwing!");
    }
}

public class FootballPlayer extends Athlete {
    private String teamName;
    private String position;

    public void tackle() {
        System.out.println("Tackling!");
    }

    public void pass() {
        System.out.println("Passing!");
    }
}
```

Time To Watch!

Extending a Class in Java

▶ https://www.youtube.com/watch?v=HjI19TvQkII *(YouTube video)*

Files from video:

- Starter code: `BankAccount.java`
- Starter code: `TestAccounts.java`
- Completed code: `CheckingAccount.java`
- Completed code: `SavingsAccount.java`

9.3. Inheritance Hierarchies and Overriding

When we extend a class, we are building an *inheritance hierarchy*. This is a tree-like structure where each class has only one parent class (such as our `Athlete` class), but can have multiple child classes (like `FootballPlayer` and `TennisPlayer`).

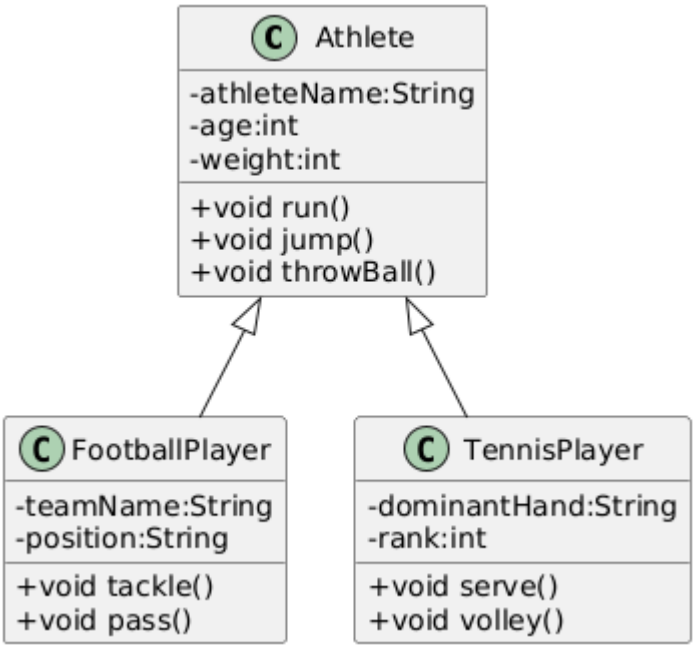


Figure 9. Inheritance hierarchy for the `Athlete` class

When a method is called on an object, Java begins by looking for that method in the object's class. If it doesn't find the method there, it looks in the superclass. If there's no method in there, it goes to **that** class's superclass, and so on up the hierarchy. If it gets to the top of the hierarchy and still hasn't found the method, the code will not compile.

```
TennisPlayer serena = new TennisPlayer();
serena.serve(); ①
serena.run(); ②
```

① The compiler checks the `TennisPlayer` class for the `serve()` method. Since it finds an implementation there, it will execute that code.

② The compiler checks the `TennisPlayer` class for the `run()` method. Since it doesn't find an implementation there, it will check the superclass, `Athlete`. Since it finds the `run()` method there, it will execute that code.

In some cases, we might want to provide a different implementation of a method in a subclass. This is called *method overriding*. Because the compiler starts at the bottom of the inheritance hierarchy and works its way up, it will use the overridden method in the subclass instead of the method in the superclass.

If a `TennisPlayer` has a specific way of running that is different from the way an `Athlete` runs, we can override the `run()` method in the `TennisPlayer` class. The `FootballPlayer` class will still use the `run()` method from the `Athlete` class, unless we override it in the `FootballPlayer` class as well.

```
public class TennisPlayer extends Athlete {
    private String dominantHand;
    private int rank;

    public void serve() {
        System.out.println("Serving!");
    }

    public void volley() {
        System.out.println("Volleying!");
    }

    @Override
    public void run() {
        System.out.println("Running like a tennis player!");
    }
}
```

The `@Override` annotation is optional, but it's a good idea to use it. It tells the compiler that you intend to override a method from the superclass. If you make a mistake in the method signature, the compiler will let you know.

9.3.1. The `Object` Class

In Java, every class is a subclass of the `Object` class. This means that every class that does not explicitly extend another class is a subclass of `Object`. The result is that `Object` is at the top of the inheritance hierarchy for all Java classes—and every object inherits the methods in the `Object` class.

The `Object` class provides a handful of methods, but there is one that is particularly useful at this stage of our learning: the `toString()` method. This method returns a `String` representation of the object.

The compiler will automatically call the `toString()` method when we try to print an object:

```
TennisPlayer serena = new TennisPlayer();
System.out.println(serena);
// Compiler changes this to: System.out.println(serena.toString());
```

Since `TennisPlayer` does not have an implementation of the `toString()` method, the compiler will work up the inheritance hierarchy until it finds an implementation in the `Object` class. The default implementation of `toString()` in the `Object` class returns a string that includes the class name and the memory address of the object, which looks something like this: `TennisPlayer@15db9742`.

This is probably not very useful to us, but we can override the `toString()` method in our `TennisPlayer` class to provide a more meaningful representation of the object. A common practice is to return a string that includes the values of the object's fields.

Example of overriding the `toString()` method in the `TennisPlayer` class

```
@Override
public String toString() {
    return "TennisPlayer{" +
            "dominantHand='" + dominantHand + '\'' +
            ", rank=" + rank +
            '}';
}
```

Now when we print a `TennisPlayer` object, we will see a string that looks something like this: `TennisPlayer{dominantHand='right', rank=1}`. However, we can return any information we want in the `toString()` method, so we can customize it to fit our needs.

The `toString()` method is used in many places in Java, so it's a good idea to override it in your classes to provide a more meaningful representation of your objects.

The `Object` class also provides other methods, such as `equals()`, which is used to compare two objects for equality. For example, we use the `equals()` method when we check if two strings are the same, like this: `if (bestSchool.equals("EMCC"))`.

Time To Watch!

Method Overriding in Java

▶ https://www.youtube.com/watch?v=NFi4uhXK8FM *(YouTube video)*

Files from video:

- Sample code: `Friend.java`
- Sample code: `CloseFriend.java`
- Sample code: `BestFriend.java`
- Sample code: `TestFriend.java`
- Starter code: `BankAccount.java`
- Starter code: `CheckingAccount.java`
- Starter code: `SavingsAccount.java`
- Starter code: `TestAccounts.java`
- Completed code: `VIPCheckingAccount.java`

The Lab Assignments in Canvas can be completed using what we've covered to this point. You might choose to complete that work now, then move onto the next section—which you'll need for the Programming Project.

9.4. Constructors & Inheritance

As we've seen, when we create an object of a class, Java automatically calls the class's constructor to initialize the object. When we create an object of a subclass, Java will call the constructor of the superclass first, and then the constructor of the subclass. Remember that the compiler will create default constructors for us if we don't provide any, but if we do provide a constructor, the compiler will not create a default constructor. Things are pretty straightforward when using default or parameterless constructors.

However, if we provide a constructor with parameters in the superclass, we need to make sure that the subclass constructor calls the superclass constructor and provides values for the parameters. To *explicitly* call the superclass constructor, use the `super` keyword.

```
public class Athlete {
    private String athleteName;
    private int age;
    private int weight;

    public Athlete(String athleteName, int age, int weight) {
        this.athleteName = athleteName;
        this.age = age;
        this.weight = weight;
    }
}

public class FootballPlayer extends Athlete {
    private String teamName;
    private String position;

    public FootballPlayer(String athleteName, int age, int weight, String teamName,
String position) {
        super(athleteName, age, weight); ①
        this.teamName = teamName;
        this.position = position;
    }
}
```

① The `super` keyword is used to call the superclass constructor. Since the `Athlete` class constructor requires three parameters, we need to provide values for those parameters here.

The superclass constructor call must be the first statement in the subclass constructor, so we can't have any other code before it. If we don't provide a call to the superclass constructor, Java will automatically call the superclass's default/parameterless constructor, which may not be what we want.

Time To Watch!

Constructors and Inheritance in Java

▶ https://www.youtube.com/watch?v=IFTod6pc7Go *(YouTube video)*

Files from video:

- Starter code: `BankAccount.java`
- Starter code: `CheckingAccount.java`
- Starter code: `SavingsAccount.java`
- Starter code: `TestAccounts.java`
- Completed code: `CheckingAccountFinished.java`

9.5. Introduction to Polymorphism

Polymorphism is a powerful concept in object-oriented programming that allows us to treat objects of different classes as if they were objects of a common superclass. It can be difficult for inexperienced programmers to recognize all of the ways that polymorphism can be used, but one of the most common uses is to create arrays of objects of different subclasses.

This means that we can create an array of `Athlete` objects, and we can use it to store objects of the `Athlete` class, the `FootballPlayer` class, and the `TennisPlayer` class. This is possible because every `FootballPlayer` *is an* `Athlete`, and every `TennisPlayer` *is an* `Athlete`.

```
Athlete[] athletes = new Athlete[3];
athletes[0] = new Athlete("Alice", 25, 150);
athletes[1] = new FootballPlayer("Bob", 30, 200, "Broncos", "Quarterback");
athletes[2] = new TennisPlayer("Charlie", 20, 175, "right", 1);
```

When we access an object in the array, we can only use the methods that are available in the `Athlete` class (unless we use something called *casting*, which is beyond our scope here). This means that we can call the `run()` method on any object in the array, but we can't call and any subclassclass methods like `tackle()` in `FootballPlayer` or `serve()` in `TennisPlayer`.

Time To Watch!

Arrays of Subclasses in Java

▶ https://www.youtube.com/watch?v=7sBbGqkEr0Q *(YouTube video)*

Files from video:

- Starter code: `BankAccount.java`
- Starter code: `CheckingAccount.java`
- Starter code: `SavingsAccount.java`
- Starter code: `InheritanceArrays.java`

Check Yourself Before You Wreck Yourself (on the assignments)

Can you answer these questions?

1. What is inheritance in object-oriented programming, and what are its benefits
2. How do you override a method in a subclass, and why might you want to do this?
3. Explain how to use the `super` keyword in a constructor.
4. Explain how different subclasses can be managed in a single array.

Sample answers provided in Stuff That's Tacked On The End.

10. 1010 - Graphical User Interfaces with Swing

Help Make These Materials Better!

I am actively working to complete and revise this eBook and the accompanying videos. Please consider using the following link to provide feedback and notify me of typos, mistakes, and suggestions for either the eBook or videos:

CIS150AB Course Materials Feedback (Google Form)

What's the Point?

- Identify the characteristics of a GUI
- Understand the role of the Swing library
- Create a GUI using Swing widgets
- Create event handlers that interact with widgets

Source code examples from this chapter and associated videos are available on GitHub.

10.1. Graphical User Interfaces

Until now, we've been creating *console applications*, in which the entire user interface is text-based. Though many utility applications are text-based, those are generally used by "power users," software developers, and system administrators. End users are used to applications that rely graphical elements to interact with the user. This is called a *graphical user interface*, which is

abbreviated as *GUI* and pronounced "gooey."

GUIs on desktop and laptop computers have elements like windows, buttons, text fields, and checkboxes, and the user interacts using a mouse or touchpad. These GUI elements are called *widgets* or *controls*. Mobile applications have similar widgets, but they are designed for touchscreens and are generally more simplified than desktop applications.

Time To Watch!

Intro to GUIs

▶ https://www.youtube.com/watch?v=ghe5TA1qA28 *(YouTube video)*

10.2. The Swing Library

The code required to create and display a functioning GUI is complex and far beyond our current skills, but we can use pre-built GUI widgets written by other developers. These are typically bundled into *libraries*, or *frameworks*, that we can use in our own programs. The two most common GUI frameworks for Java are called *Swing* and *JavaFX*; we'll use Swing in this course because it's a little simpler for beginners and does not require any additional installations.

JavaFX is newer and more powerful than Swing, but it's also more complex and has a steeper learning curve. The concepts learned in Swing will transfer to JavaFX, so learning Swing is a great place to start.

To use Swing classes, we simply need to add an `import` statement at the top of our Java file:

```
import javax.swing.*;
```

We'll begin by instantiating a `JFrame` object, which is the main window of our application. We can use setters on the `JFrame` object to set attributes like the title and size of the window; we'll also want to set the attribute that determines what happens when the user closes the window. We can then add other widgets to the frame, like buttons, text fields, and labels. Finally, we'll set the frame to be visible, which will cause the window to appear on the screen.

Example of a simple GUI using Swing

```
import javax.swing.*;

public class BasicGUI {

    public static void main(String[] args) {
        JFrame frame = new JFrame("Hello, World!");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JLabel lblHello = new JLabel("It's so gooey!");
        frame.add(lblHello);
        frame.setVisible(true);
    }
```

```
}
```

This code creates a window with the title "Hello, World!" and a label that says "It's so gooey!".

The `JFrame.EXIT_ON_CLOSE` attribute tells the program to exit when the user closes the window. If this attribute is not set, the program will continue running in the background after the window is closed. There are a few other options for this attribute, but `EXIT_ON_CLOSE` is the most common.

Figure 10. Screenshot of `BasicGUI.java` in action on macOS

As you can see, the code to create a simple GUI is a little more complex than the single `println()` statement of an equivalent console application, which is why we've waited until now to learn it.

Time To Watch!

Intro to Swing in Java

▶ https://www.youtube.com/watch?v=djh5Cd0cPmA *(YouTube video)*

Files from video:

- Starter code: `SimpleGUI.java`
- Completed code: `SimpleGUIFinished.java`

10.3. Event-Driven Programming with Swing

A GUI application offers a lot more flexibility than a console application, since the user can interact with the program in many ways. In a console application, the program runs from top to bottom, and the user can only interact by typing text. In a GUI application, the user can click buttons, type in text fields, and select items from drop-down lists. This means that the program must be able to respond to these events; we call this *event-driven programming*, and it is a key concept in GUI programming.

In Swing, we can add *event listeners* to widgets, which are objects that respond to events. For example, we can add an event listener to a button that will run a method when the button is clicked. The method that runs in response to an event is called an *event handler*. Once we've added an event listener to a widget, the event handler is like any other method in our program, and we can write it to do whatever we want.

Time To Watch!

Swing Event Handling

▶ https://www.youtube.com/watch?v=rsZ6f-twWfI *(YouTube video)*

Files from video:

- Completed code: `GreetingFrame.java`

Note: there is no starter code for this video.

10.4. Processing User Input with Swing

Once we've learned how to work with widgets and add event listeners, we can put everything together to create a GUI application that gets input from the user, performs actions or calculations with that data, and displays the results.

Time To Watch!

Calculations in Swing

▶ https://www.youtube.com/watch?v=dJfuwUfRczI *(YouTube video)*

Files from video:

- Starter code: `GUICalculations.java`
- Completed code: `GUICalculationsFinished.java`

10.5. Widgets

Using `JLabel` and `JTextField`, we've been about to create GUI programs that function much like console applications, but that doesn't really take advantage of the power of GUIs. A well-designed GUI application utilizes specialized widgets that are designed for specific types of user input and output.

Here are some common widgets and their purposes:

JLabel
: Displays text or an image.

JTextField
: Allows the user to type in a single line of text.

JTextArea
: Allows the user to type in multiple lines of text.

JButton

A clickable button that can run a method when clicked.

JCheckBox

A checkbox that can be checked or unchecked.

JRadioButton

A radio button that can be selected or deselected, and can be grouped with other radio buttons to limit the user to selecting only one.

JComboBox

A drop-down list that allows the user to select one item from a list.

There are many more widgets in the Swing framework, but these will cover most of what you'll need for basic GUI applications. There is a great deal of documentation available online for the Swing framework, so you can always look up how to use a specific widget.

Use widgets as they are intended, even if they are able to be used in other ways, such as displaying output in a text field. Users are accustomed to certain behaviors from widgets, and using them in unexpected ways can make the application harder to use.

10.5.1. Widget Naming Conventions

When naming widgets, it's a good idea to use a consistent naming convention that makes it clear what type of widget it is. There are a variety of conventions for naming widgets, but two are most common:

- Prefix the name to identify the widget type, like `lbl` for a label, `txt` for a text field, or `btn` for a button.
- Append the type of widget to the end of the name, like `helloLabel`, `nameTextField`, or `submitButton`.

I use the prefix method in my code, mostly because I'm used to that from C# programming (where that's the preferred style), but you can use either approach on your assignments. The most important thing is to be consistent in your naming so that anyone reading your code can easily understand what each widget is for.

Time To Watch!

More Swing Widgets

▶ https://www.youtube.com/watch?v=2HlIsvtYTec *(YouTube video)*

Files from video:

- Sample code: `JCheckBoxDemo.java`
- Sample code: `JRadioButtonDemo.java`

- Sample code: `JComboBoxDemo.java`
- Sample code: `JComboBoxDemo2.java`

10.6. GUI Layouts

As we add more widgets to our GUI, we'll need to consider how they are arranged on the screen. While we can set the position of each widget manually, this is tedious and doesn't work well when the window is resized. Instead, we should use *layout managers*, which are objects that arrange widgets in a specific way and respond to window and screen sizes in predictable ways. Think of a layout manager as a set of rules that determine how widgets are arranged in a window.

The Swing framework is itself built on top of **another** framework called the *Abstract Window Toolkit*, or *AWT*. Swing hides most of that from us by implementing classes that extend the AWT classes. For example, the `JLabel` class is a Swing widget that extends the `Label` class from AWT. So we don't directly use AWT very often, but the exception to that is layout managers. Layout managers are part of AWT, so we'll have to import them from the `java.awt` package:

```
import java.awt.*;
```

There are several layout managers available in AWT, each with its own strengths and weaknesses. The most common layout managers are:

FlowLayout
: Widgets are arranged in a single row or column, and wrap to the next row or column when the window is resized.

GridLayout
: Widgets are arranged in a grid, with a specified number of rows and columns.

BorderLayout
: Widgets are arranged in five regions: north, south, east, west, and center.

To create more complex layouts, we can nest layout managers, which means that we can put a layout manager inside another layout manager.

There are tools that allow us to create GUIs visually, by dragging and dropping widgets onto a window, and then generating the code that will produce the GUI. But our goal is to learn how layout managers work, so we'll be creating our GUIs by writing the code ourselves. For GUI designs that don't have to be turned into an obnoxious professor for a grade, you're welcome to take advantage of these tools.

Time To Watch!

Swing Layouts

▶ https://www.youtube.com/watch?v=kM3p7V_3UOk *(YouTube video)*

Files from video:

- Sample code: `FlowLayoutDemo.java`
- Sample code: `JPanelDemo.java`
- Sample code: `GridLayoutDemo.java`
- Sample code: `BorderLayoutDemo.java`
- Sample code: `NestedLayoutDemo.java`
- Sample code: `NestedLayoutDemoColors.java`

Check Yourself Before You Wreck Yourself (on the assignments)

Can you answer these questions?

1. What is a Graphical User Interface (GUI) and why is it important for end users?
2. Explain the role of the Swing library in Java and why it is preferred for beginners over JavaFX.
3. Describe the process of creating a simple GUI using Swing, including the main components involved.
4. What is event-driven programming in the context of GUIs, and how does it differ from console-based programming?

Sample answers provided in Stuff That's Tacked On The End.

11. Stuff That's Tacked on the End

Sample answers to *Check Yourself Before You Wreck Yourself (on the assignments)* questions

These review questions are intended to be a little open-ended, so your answers will vary. Many of the sample answers in this section are adapted from responses generated by Copilot in early 2025, using the web interface. Unfortunately, I didn't take careful notes on which responses I used as I was experimenting with Copilot and only expected to use the material as a placeholder until I wrote my own. It turned out that the Copilot responses were pretty good.

Getting Started Chapter

1. ***What does JDK stand for?***

Java Development Kit

2. ***What is an example of an Integrated Development Environment?***

   Visual Studio Code, eclipse, IntelliJ, and many others.

Computers and Coding

1. ***What is the primary role of a programming language in the context of computer programming?***

   The primary role of a programming language is to act as a bridge between human language and machine language. It allows humans to write instructions in a way that is easier for them to understand and use, which can then be accurately translated into machine language that a computer can execute.

2. ***Explain the difference between compiled and interpreted programming languages. Provide an example of each.***

   - **Compiled languages** are those where the source code is translated into machine language by a compiler before the program is run. This machine language file can then be executed directly by the computer. An example of a compiled language is C.
   - **Interpreted languages** are those where the source code is translated into machine language on the fly, as the program is running, by an interpreter. An example of an interpreted language is Python.
   - **Java** is a unique case as it is both compiled and interpreted. The source code is first compiled into an intermediate language called bytecode, which is then interpreted by the Java Runtime Environment (JRE) at runtime.

3. ***Describe the basic structure of a simple Java program, such as the "Hello World" example provided in the chapter.***

   - A simple Java program, like the "Hello World" example, consists of:
   - **Class declaration**: This defines the class, which is a blueprint for objects. In the example, the class is named `HelloWorld`.
   - **main() method**: This is the entry point of the program where execution begins. It is declared as `public static void main(String[] args)`.
   - **println() statement**: This is used to output text to the console. In the example, `System.out.println("Hello World!");` prints "Hello World!" to the console.
   - **Code blocks**: These are enclosed in curly braces `{}` to define the scope of the class and the main method.

4. ***What are the key steps in the software development process as outlined in the chapter? Why is it important to follow these steps?***

   - The key steps in the software development process are:
   - **Analysis**: Identifying the goals and scope of the program. This step ensures that you understand what the program needs to do.
   - **Testing Plan**: Determining how the final program will be tested. This step ensures that you have a clear understanding of how to verify that the program works correctly.
   - **Implementation**: Writing and testing the code iteratively. This step involves developing the

program piece by piece, testing each part to ensure it works before moving on to the next.

- **Revise or Maintain**: Updating the program as needed based on changing requirements or ensuring it continues to perform as expected over time.
- Following these steps is important because it helps keep the development process organized and focused, ensures efficient use of time, and makes complex programming tasks more approachable.

Variables

1. ***What is the difference between a constant and a variable in Java? Provide an example of each.***
   - A **constant** is a piece of data that cannot change during program execution. It is declared using the `final` keyword and must be assigned a value when it is declared. For example:

     ```
     final double SALES_TAX_RATE = 8.7;
     ```

   - A **variable** is a piece of data that can change (or "vary") during program execution. It is declared by specifying a data type and an identifier, and can be assigned different values throughout the program. For example:

     ```
     int age = 21;
     age = 22; // The value of age can be changed
     ```

2. ***Explain the purpose of the `Scanner` class in Java and provide an example of how it is used to get user input.***
   - The `Scanner` class in Java is used to get input from the user through the keyboard. It provides methods to read different types of input, such as strings, integers, and doubles. To use the `Scanner` class, you need to import it and create an instance of the class. Here is an example:

     ```
     import java.util.Scanner;

     public class InputDemo {
       public static void main(String[] args) {
         Scanner input = new Scanner(System.in);
         String name;
         int age;

         System.out.print("Enter your name: ");
         name = input.nextLine();

         System.out.print("Enter your age: ");
         age = input.nextInt();

         System.out.println("Hello, " + name + ". You are " + age + " years old.");
       }
     ```

```
}
```

3. ***Describe the difference between integer division and floating-point division in Java. Why is it important to be aware of this distinction?***
   - **Integer division** in Java occurs when both operands are integers. The result is also an integer, and any fractional part is truncated (i.e., discarded). For example:

     ```
     int result = 10 / 3; // result is 3, not 3.33
     ```

   - **Floating-point division** occurs when at least one of the operands is a floating-point number (e.g., `float` or `double`). The result is a floating-point number, and the fractional part is preserved. For example:

     ```
     double result = 10.0 / 3; // result is 3.333333...
     ```

   - It is important to be aware of this distinction because using integer division when you expect a floating-point result can lead to incorrect calculations and unexpected behavior in your programs.
4. ***What are compound assignment operators in Java? Provide examples of how they are used in arithmetic operations.***
   - Compound assignment operators in Java are shorthand ways of writing common arithmetic operations that update the value of a variable. They combine an arithmetic operator with the assignment operator (`=`). Here are some examples:

     ```
     int x = 5;
     x += 3; // Equivalent to x = x + 3; x is now 8

     int y = 10;
     y -= 2; // Equivalent to y = y - 2; y is now 8

     int z = 4;
     z *= 2; // Equivalent to z = z * 2; z is now 8

     int a = 20;
     a /= 4; // Equivalent to a = a / 4; a is now 5

     int b = 15;
     b %= 6; // Equivalent to b = b % 6; b is now 3
     ```

Methods

1. ***What is the main purpose of using methods in Java, and how do they contribute to code maintainability?***

   The main purpose of using methods in Java is to organize code into reusable blocks that

perform specific tasks. Methods help in breaking down complex programs into smaller, manageable pieces, making the code easier to read, understand, and maintain. By encapsulating functionality within methods, changes can be made in one place without affecting other parts of the program. This improves maintainability because if a bug is found or a change is needed, it can be addressed within the method itself, rather than having to update multiple instances of the same code throughout the program.

2. ***Explain the difference between a parameter and an argument in the context of Java methods. Provide an example to illustrate your explanation.***

   In Java, a parameter is a variable defined in the method declaration that acts as a placeholder for the value that will be passed to the method. An argument, on the other hand, is the actual value that is passed to the method when it is called. For example, in the method definition `public static void printArea(double radius)`, `radius` is a parameter. When the method is called with `printArea(5.0)`, the value `5.0` is the argument passed to the method.

3. ***Why is it generally better to return values from methods rather than printing them directly within the method? How does this practice improve the modularity and reusability of code?***

   It is generally better to return values from methods rather than printing them directly within the method because returning values allows the method to be more versatile and reusable. When a method returns a value, it can be used in various contexts, such as in calculations, assignments, or further processing, without being tied to a specific output format. This practice improves modularity by keeping the method focused on a single task (e.g., performing a calculation) and leaving the decision of how to use the result to the calling code. It also enhances reusability, as the method can be used in different parts of the program or even in different programs without modification.

Classes and Objects

1. ***What is the primary difference between procedural programming and object-oriented programming (OOP)?***

   Procedural programming organizes programs as a collection of tasks or methods that need to be performed in a specific order. In contrast, OOP organizes programs as a collection of interacting objects, each representing a real-world entity with attributes (data) and behaviors (methods).

2. ***Explain the concept of a class and how it relates to objects in OOP.***

   A class is a blueprint or template for creating objects. It defines the attributes and behaviors that the objects created from the class will have. An object is an instance of a class, meaning it is created with the fields and methods implemented in the class.

3. ***What is encapsulation, and why is it important in OOP? Provide an example.***

   Encapsulation is the concept of hiding the internal state of an object and requiring all interaction to be performed through the object's methods. This is important because it protects the object's data from unintended or harmful modifications. For example, in a bank account class, the balance field is private and can only be accessed or modified through a public getter method, and modified through public deposit and withdraw methods.

Decisions

1. ***Explain what Boolean expressions are and how they are used to make decisions in Java.***

   Boolean expressions are statements that evaluate to either true or false. In Java, they are used to make decisions by determining which blocks of code to execute based on the evaluation of these expressions.

2. ***Explain the difference between a relational operator and a logical operator.***

   Relational operators compare two values and determine their relationship, such as equality (==), inequality (!=), less than (<), and greater than (>). Logical operators, on the other hand, combine multiple Boolean expressions to form a single, more complex expression, such as AND (&&), OR (||), and NOT (!).

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

   An if statement allows you to execute a block of code only if a certain condition is true. An if-else statement extends this by providing an alternative block of code to execute if the condition is false, allowing you to choose between two possible actions.

4. ***How can you write code that runs one code block from multiple options?***

   You can use a switch statement to run one code block from multiple options based on the value of an expression. Alternatively, you can use an if-else if structure to chain multiple if-else statements together, allowing you to handle more than two options.

Loops

1. ***Describe the difference between a while loop and a do-while loop.***

   A while loop checks the condition before executing the block of code, meaning the code might not execute at all if the condition is false initially. A do-while loop, on the other hand, checks the condition after executing the block of code, ensuring that the code is executed at least once.

2. ***What is a control variable, and how is it used in loops?***

   A control variable is a variable used to control the execution of a loop. It is typically initialized before the loop starts, checked in the loop's condition to determine whether the loop should continue, and updated within the loop to eventually meet the condition that stops the loop.

3. ***Give an example of an indefinite loop.***

   An example of an indefinite loop is a program that asks the user to guess a number and keeps prompting the user until they guess correctly. The loop repeats until the user's input matches the target number. Another example is a loop that reads user input until a specific sentinel value is entered, indicating the end of input.

Debugging and Generative AI

1. ***What is the difference between a compile-time error and a runtime error?***

   Compile-time errors occur when the compiler cannot translate the source code into machine code due to syntax or semantic issues. These errors must be fixed before the program can run. **Runtime errors**, on the other hand, occur during the execution of the program and are often due to logic errors, causing the program to behave unexpectedly or crash.

2. ***How can using output statements help in debugging a program?***

Output statements, such as `print()` statements, allow you to monitor the values of variables and the flow of execution in your program. This visibility helps you pinpoint where things are going wrong, making it easier to identify and correct errors.

3. ***What are some strategies you can use when you're frustrated by a bug in your code?***

   When you're frustrated by a bug in your code, take a break to clear your mind and return with fresh eyes. Use output statements to track variables and execution flow, clean up your code for better readability, and break down the problem into smaller parts. If needed, seek help from others and stay patient, knowing that debugging is a valuable part of the learning process.

Arrays

1. ***What is an array and how does it differ from a single variable?***

   An array is a data structure that allows you to store multiple pieces of data in a single collection. Unlike a single variable, which can only store one piece of data at a time, an array can hold multiple values, each of which is referred to as an element. Each element in an array is accessed using its unique index, which starts at 0 for the first element and increments by 1 for each subsequent element.

2. ***What is an ArrayIndexOutOfBoundsException and when does it occur?***

   An `ArrayIndexOutOfBoundsException` is a runtime error that occurs when you try to access an element at an index that is outside the bounds of the array. This means you are trying to access an index that is either less than 0 or greater than or equal to the length of the array. For example, if an array has 10 elements, valid indices are 0 through 9. Attempting to access the element at index 10 or higher will result in this exception.

3. ***How can loops be used to traverse an array? Provide an example of a for loop that sums the elements of an integer array.***

   Loops can be used to traverse an array by iterating through each element, performing operations such as reading, modifying, or accumulating values. Here is an example of a for loop that sums the elements of an integer array:

   ```
   int[] scores = {90, 85, 95, 88, 92};
   int sum = 0;
   for (int i = 0; i < scores.length; i++) {
       sum += scores[i];
   }
   System.out.println("The sum of the scores is " + sum);
   ```

   In this example, the for loop iterates through each element of the scores array, adding each element's value to the sum variable. After the loop completes, the total sum of the array elements is printed.

Inheritance

1. ***What is inheritance in object-oriented programming, and what are its benefits***

   Inheritance is a feature in object-oriented programming that allows a class (subclass) to inherit attributes and methods from another class (superclass), promoting code reuse and organization

by enabling the creation of hierarchical relationships between classes.

2. ***How do you override a method in a subclass, and why might you want to do this?***

   To override a method in a subclass, you define a method in the subclass with the same name, return type, and parameters as the method in the superclass. This allows the subclass to provide a specific implementation of the method, which can be used to customize or extend the behavior of the inherited method.

3. ***Explain how to use the `super` keyword in a constructor.***

   The super keyword is used in a subclass constructor to call the constructor of its superclass. This must be the first statement in the subclass constructor and is used to ensure that the superclass is properly initialized before the subclass adds its own initialization code. For example: super(parameter1, parameter2);.

4. ***Explain how different subclasses can be managed in a single array.***

   Different subclasses can be managed in a single array by declaring the array to be of the type of their common superclass. This allows the array to hold instances of any subclass, enabling polymorphism. For example, an array of type `BankAccount` can hold instances of both `CheckingAccount` and `SavingsAccount`, and common methods can be called on these instances through the superclass reference.

Swing GUIs

1. ***What is a Graphical User Interface (GUI) and why is it important for end users?***

   A Graphical User Interface (GUI) is a type of user interface that allows users to interact with electronic devices using graphical elements such as windows, icons, and buttons, rather than text-based commands. It is important for end users because it makes software more accessible and easier to use, especially for those who are not familiar with command-line interfaces.

2. ***Explain the role of the Swing library in Java and why it is preferred for beginners over JavaFX.***

   The Swing library in Java provides a set of pre-built GUI components that developers can use to create graphical user interfaces. It is preferred for beginners over JavaFX because it is simpler to use, does not require additional installations, and the concepts learned in Swing can be easily transferred to JavaFX.

3. ***Describe the process of creating a simple GUI using Swing, including the main components involved.***

   To create a simple GUI using Swing, you start by importing the necessary Swing classes, then instantiate a JFrame object to serve as the main window. You can set attributes like the title and size of the window, and add other components such as JLabel, JButton, and JTextField to the frame. Finally, you make the frame visible by calling the setVisible(true) method.

4. ***What is event-driven programming in the context of GUIs, and how does it differ from console-based programming?***

   Event-driven programming in the context of GUIs involves writing code that responds to user actions, such as clicking a button or entering text. This is different from console-based programming, where the program runs sequentially from top to bottom and waits for user input at specific points. In event-driven programming, the program continuously listens for events and executes the corresponding event handlers when an event occurs.