Essential Java for AP CompSci

From Programming to Computer Science

Doug Winnie

Apress®

Essential Java for AP CompSci

From Programming to Computer Science

Doug Winnie

Apress®

Essential Java for AP CompSci: From Programming to Computer Science

Doug Winnie
Mission Hills, KS, USA

ISBN-13 (pbk): 978-1-4842-6182-8 ISBN-13 (electronic): 978-1-4842-6183-5
https://doi.org/10.1007/978-1-4842-6183-5

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar
Cover image by Devin Avery on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail editorial@apress.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484261828. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

For Mike, and all of the great decisions we have made together.

Table of Contents

About the Author xvii

About the Technical Reviewer xix

Sprint 1: Introduction 1

What You Need 3

Sprint 2: Setting Up the Java JDK and IntelliJ 5

Coding Tools and IDEs 5

Installation and Setup 6

Install the JDK 6

Install IntelliJ 6

Sprint 3: Setting Up GitHub 9

GitHub 9

How GitHub Works 10

Lifecycle of a Repository 10

Sprint 4: Programming Languages 13

Origin of Programming 13

Forms of Programming 14

Machine Language 14

Interpreted 14

Compiled 15

Object-Oriented 15

Data 15

Functional 16

Scripting 16

Sprint 5: History and Uses of Java 17
Java Beginnings 17
Java's Primary Goals 17
Uses of Java 18
Sprint 6: How Java Works 19
The Problem with Compiled Languages 19
The JVM and JRE 20
Compiling Java Bytecode 21
Precompiled Files 22
OpenJRE 22
Sprint 7: Flowcharting 23
Flowcharting Tools 23
Paper 23
Tablet and Stylus 24
Apps 24
Flowcharting Basics 24
Terminus 24
Process/Action 24
Input and Output 25
Decisions 25
Annotations 25
Other Shapes 25
Take Out the Trash 26
But Is It Really That Simple? 26
Sprint 8: Hello, World 29
Create Your IntelliJ Java Project 29
IntelliJ IDEA 29
First Time Only: Configure the JDK 30
Create Project 33
About Your Project 36

Coding Your Project 38
Writing Your First Program 38
Create Your Repo in GitHub 40
Upload Your Code to GitHub 45
Sharing Program Output 47

Sprint 9: Simple Java Program Structure 49

Sprint 10: Text Literals and Output 51

Text Output 51
Escape Sequences 52

Sprint 11: Value Literals 55

Literal Formatting 56

Sprint 12: Output Formatting 59

Decimal Formatters 59
Thousands Formatters 60
Currency Formatters 60
Spacing and Alignment Formatters 61
Multiple Items in Formatters 62

Sprint 13: Comments and Whitespace 65

Sprint 14: Abstraction of Numbers 67

Sprint 15: Binary 75

Binary Numbers 76
Bit Size and Values 81
Overflow 84

Sprint 16: Unicode 87

Text Encoding 87
ASCII + Unicode 88
Emoji 88

Sprint 17: Variables 89
Essentials of Variables 89
Code Examples 90
Sprint 18: Math! 93
Basic Operators 93
Order of Operations 93
String Concatenation 94
Code Examples 94
Sprint 19: Math Methods 97
Working with Simple Methods 97
Multiparameter Methods 98
Illegal Value Types in Methods 98
Math Constants 99
Code Examples 99
Sprint 20: Managing Type 101
Mixing Types in Evaluations 101
Numbers to Strings 102
Strings to Numbers 102
Casts 103
Cast Errors 103
Code Examples 104
Sprint 21: Random Numbers 107
Create a Random Number Generator 107
Random Integers 108
Random Decimals 109
Code Examples 110

Sprint 22: Capture Input 113
Hello, Scanner 113
Capturing Strings 114
Capturing Integers 114
Capturing Decimals 115
Code Examples 115
Sprint 23: Creating Trace Tables 117
It's a Spreadsheet 117
Um. Why? 118
Sprint 24: Methods 119
Method Basics 119
Writing a Method 119
Call a Method 120
Method Flow 120
Code Guide 120
Code Examples 121
Sprint 25: Calling Methods Within Methods 123
Methods Within Methods 123
Infinite Methods 124
Code Examples 125
Sprint 26: Methods and Values 127
Accepting Values in Methods 127
Returning a Value 128
Overloading a Method 128
Code Guides 129
Code Examples 132

Sprint 27: Methods and Scope 135
Variable Scope Errors 135
Defining Class-Scoped Variables 135
Class Conflicts 136
Code Examples 138

Sprint 28: Boolean Values and Equality 141
Creating a Boolean Variable 141
Boolean Logic Operators 142
Altering a Boolean Value 142
Combining Logic with Evaluations 142
Compound Logic Operators 143
Code Examples 143

Sprint 29: Simple Conditional Statements 147
The if Statement 147
The else Statement 147
The else if Statement 148
Understanding Conditional Flow 148
Code Examples 149

Sprint 30: Matching Conditions with the switch Statement 153
Creating a switch Statement Code Block 153
Things to Look Out for with the switch Statement 155
Code Examples 155

Sprint 31: The Ternary Operator 157
The if-else Statement Equivalent 157
Converting to a Ternary Operator 158
Using the Ternary Operator Inline with Code 158
Code Examples 158

Sprint 32: The Stack and the Heap 161

Understanding the Stack 161

Understanding the Heap 162

Why This All Matters 162

Sprint 33: Testing Equality with Strings 163

When the Heap Throws Equality 163

How to Better Compare String Values 164

Code Examples 165

Sprint 34: Dealing with Errors 167

Coding to Catch Errors 167

"Catching" Specific Errors 168

Code Examples 169

Sprint 35: Documenting with JavaDoc 173

Using JavaDoc Syntax 173

Generating Documentation 175

Code Examples 177

Sprint 36: Formatted Strings 179

Creating a Formatted String Literal 179

Code Examples 179

Sprint 37: The while Loop 181

Create a while Loop 181

Code Examples 182

Sprint 38: Automatic Program Loops 183

Creating a Program Loop 183

Code Examples 185

Sprint 39: The do/while Loop 187

Creating a do…while Loop 187

Run at Least Once 188

Sprint 40: Simplified Assignment Operators 189

Combined Assignment 189

Increment and Decrement 190

Placement and Program Flow 190

Code Examples 191

Sprint 41: The for Loop 195

Creating a for Loop 195

Changing the Step 196

Code Examples 197

Sprint 42: Nesting Loops 199

Creating Nested Loops 199

Displaying as a Grid 200

Code Examples 202

Sprint 43: Strings as Collections 203

Creating Strings Using the String Class 203

Getting a String Length 203

Getting a Specific Character from a String 203

Finding a Character in a String 204

Extracting a Substring 205

Comparing Strings 205

Code Examples 206

Sprint 44: Make Collections Using Arrays 209

Creating an Array with Values 209

Getting a Value from an Array 210

Creating an Array by Size 210

Things to Avoid with Arrays 210

Getting the Number of Values in an Array 211

Looping Through an Array 211

Code Examples 212

Sprint 45: Creating Arrays from Strings 215

Delimited Strings 215

Splitting It Up 216

What About Numbers? 217

Code Examples 218

Sprint 46: Multidimensional Arrays 219

Define a Multidimensional Array 220

Assign Values to Multidimensional Arrays 221

Access Values in Multidimensional Arrays 221

Rectangular and Irregular Arrays 221

Code Examples 222

Sprint 47: Looping Through Multidimensional Arrays 225

Creating Nested Loops for Arrays 225

Code Examples 227

Sprint 48: Beyond Arrays with ArrayLists 229

Create an ArrayList 229

Add Items to ArrayLists 230

Get Elements in ArrayLists 230

Remove Elements from ArrayLists 231

Find Items in ArrayLists 231

Replace Items in ArrayLists 232

Get the Size of an ArrayList 232

Copy Elements to a New ArrayList 232

Clear an ArrayList 232

Code Examples 233

Sprint 49: Introducing Generics 235

Create an ArrayList with Generics 235

Typing Using Generics 236

Code Examples 237

Sprint 50: Looping with ArrayLists 239

Working with size() and get() Methods 239

Code Examples 240

Sprint 51: Using for…each Loops 241

Mechanics of a for…each Loop 241

This Is the Mechanics of the for…each Loop 242

ArrayLists Without Generics 243

Yep, Arrays Work Too 243

Code Examples 244

Sprint 52: The Role-Playing Game Character 247

What Is a Role-Playing Game Character? 247

Filling Out Our Character Sheet with Data 248

Classes, Instantiation, and Construction 250

Player Character Sheets 250

Fighter 251

Mage 251

Paladin 252

Priest 252

Sprint 53: Polymorphism 253

Creating a Class Hierarchy 253

Party Up—All the Same—but All Different at the Same Time 254

The Essential Tool: The Die 255

Class Hierarchy, Polymorphism, Abstract, and Static 255

Sprint 54: Make All the Things…Classes 257

Creating Some Class 257

Instantiate Thyself, Class! 263

Sprint 55: Class, Extend Thyself! 269

Sprint 56: I Don't Collect Those; Too Abstract 283

Sprint 57: Access Denied: Protected and Private 291

Sprint 58: Interfacing with Interfaces 305

Sprint 59: All I'm Getting Is Static 309

Sprint 60: An All-Star Cast, Featuring Null 315

Index 321

About the Author

Doug Winnie has been teaching programming in the classroom or with online videos for over 15 years. Online, his videos have over two million views on Adobe, Lynda.com, and LinkedIn Learning. Doug's courses cover topics like computer science principles, Java, C#, JavaScript, product management fundamentals, virtual machines, and other products and technologies.

He has written two books on programming and collaboration between user experience designers and developers.

Currently, he is the Chief Learning Officer at MentorNations, an international nonprofit focused on evolving digital literacy and entrepreneurial skills across the world.

Previously, Doug was head of community for the LinkedIn Learning instructor organization, representing the interests of over 1400 teachers and instructors worldwide. He was an internal contributor to the Windows Insider Program and part of a global initiative at Microsoft to teach 7.6 billion people around the world digital coding and programming literacy skills. He was also a LinkedIn Culture Champion, working with LinkedIn employees around the globe to put on employee cultual events on special days called InDays.

Earlier, Doug was a principal product manager at Adobe and specialized working on new products for the user experience, interactive design, and web design audiences.

You can find out more about Doug on his LinkedIn profile:

www.linkedin.com/in/sfdesigner

About the Technical Reviewer

Jeff Friesen is a freelance teacher and software developer with an emphasis on Java. In addition to authoring *Java I/O, NIO and NIO.2* (Apress) and *Java Threads and the Concurrency Utilities* (Apress), Jeff has written numerous articles on Java and other technologies (such as Android) for JavaWorld (`JavaWorld.com`), InformIT (`InformIT.com`), `Java.net`, SitePoint (`SitePoint.com`), and other websites. Jeff can be contacted via his website at `JavaJeff.ca` or via his LinkedIn profile (`www.linkedin.com/in/javajeff`).

SPRINT 1

Introduction

Computer science has become a basic life skill that everyone is going to need to learn. Whether you are going into a career or side hustle in business, technology, creativity, architecture, or almost any other field, you will find programming, coding, and computer science play a role.

In fact, if you look at the top skills on LinkedIn for the United States, in the last year all ten of the top were programming, computer science, or code related. These include cloud and distributed computing, statistical analysis, data mining, mobile development, storage systems management, user interface design, network and information security, middleware and integration software, web architecture and development frameworks, algorithm design, and Java development.[1]

What is unique about computer science is how it has become a skill, and not just a career. While there are jobs and titles of "Computer Scientist," the skill of computer science, and specifically coding and programming, is almost everywhere.

In marketing, you need to analyze and sift through tons of user data and metrics on how people use your products, website, apps, or services. In medicine, doctors and researchers need to gather and analyze data gathered from clinical studies or to find new breakthroughs. In agriculture and farming, thousands of IoT devices need to be deployed and managed to gather data on soil conditions, humidity, and crop health. In architecture and construction, mapping out and analyzing how people use elevators, building infrastructure, or public spaces can help design better buildings for people to work and live in. Even when building a side business, or "hustle," entrepreneurs may need to perform research, build a website, program a mobile app or game, or perform tons of different activities involving code, logic, automation, and programming.

[1] "The Top 10 Skills You Will Be Hiring For in 2017," https://business.linkedin.com/talent-solutions/blog/trends-and-research/2016/linkedin-new-report-reveals-the-latest-job-seeking-trends

D. Winnie, *Essential Java for AP CompSci*, https://doi.org/10.1007/978-1-4842-6183-5_1

When people create projects based on code, they organize their work into multiple chunks or segments of development. They call these "sprints." So, we will learn Java in the same way, using multiple sprints that will teach a single topic to help us keep pace with things along the way. Part of what makes learning coding difficult is it can be difficult to find real-world examples to draw parallels from. If you are planning on taking the AP Exam in Computer Science, many of the questions will not have any real-world context and will require you to understand and follow code without that as a reference.

For you, this can make it confusing, but just remember: there are two sides to programming—what is being done and how it happens.

The "what" is pretty easy, and this gives the code context, like parsing transit data, rolling a virtual die, or adding sales data for a quarter together.

"How" it happens is a completely different story however. This is where you need to break things down into individual steps and start to think like a computer. A computer is going to think only about one single step at a time. It has no concept of what comes next or what came before it. It only is concerned with the present. So, if it can't find something—you get an error. If it tries to do something that doesn't exist—you get an error. If you try doing something out of order—you guessed it, you get an error.

In fact, the computer is pretty dumb. I mean, like really dumb. What makes it smart is how we are able to string multiple actions together to make something work. And with growing advances in artificial intelligence, the computer itself can start making adjustments. And then Skynet activates, and we all know what happens after that.

So when we learn programming, we are going to focus on three things:

- What is the process
- What is the syntax
- What is the flow

The process is represented as a flowchart. We will learn how to make these to help you plan out what you are going to do before you write a line of code. At first, the flowcharts will be pretty simple, but then they will get more complex. And yep, you will get pretty annoyed with flowcharts before the year is over, but trust me…they help.

The syntax is the code; this is what you write that translates the process you create in a flowchart to the instructions that the computer can understand.

Finally, there is the flow. This is where you trace through the code and see how the data and information it stores along the way changes, and you can see how the operation of the program cascades from line to line. You will be building charts that will capture the programming flow so you can better understand how the computer processes code to make your next program easier to conceive and code.

What You Need

You will need a Windows or macOS computer to complete the projects in this book. You will need to install and configure the Java JDK (Java Development Kit) and IntelliJ from JetBrains (our IDE or code editor for the class). If you are on macOS, it is suggested that you uninstall the built-in Java JRE (Java Runtime Environment) and use the one included in the Oracle Java JDK. This will avoid warnings that appear when you build your projects.

SPRINT 2

Setting Up the Java JDK and IntelliJ

To get started with our exercises, we need to set up our computers to work with Java code and run it. This is called creating a development environment.

As you learn more about coding and programming, you will encounter many different tools to help you do your job. There are so many in fact, that you might find that one fits better than another. While they all essentially do the same thing, there are features and options in different tools that mesh better with different types of coders.

Coding Tools and IDEs

Coding tools, and their more sophisticated brothers, the integrated development environment, or IDE, can sometimes spark passionate debate on which one is considered "the best," but ultimately it comes down to you, the coder, to decide what is best for you.

To help make this course and the tools we use easy to understand and the same across Windows and macOS, the tool we will use is **IntelliJ IDEA**. IDEA is a free coding tool from JetBrains that is easy to use, not very complicated, and it is the same across platforms.

There are others that you can use for this course as well. Some examples are Eclipse or NetBeans. These tools provide the same features, but I wouldn't recommend using them to help us all be consistent with our tools when working together.

The other key tool you will need is the **Java Development Kit** or JDK. In order for the IDE to work, it needs to have the Java compiler to create your Java bytecode and then use the Java Runtime Environment or JRE to run your program in a virtual machine, called the JVM.

D. Winnie, *Essential Java for AP CompSci*, https://doi.org/10.1007/978-1-4842-6183-5_2

The version that you will use is the Java SE or Standard Edition that you can download for free from Oracle. Regardless if you use IDEA, or another IDE, you will need to install the JDK in order to compile and build your programs.

Installation and Setup

If you want to follow along with the same tools that I'll be using, you will need to install the JDK and IntelliJ IDEA.

To install the tools I'm using, you will need a PC with Windows 7, 8, 8.1, or 10 already installed.

For a macOS computer, you will need Yosemite, High Sierra, or Mojave. It is recommended that you uninstall the version that Apple installs by default (called OpenJDK). Refer to instructions online for how to do this.

Install the JDK

First, we need to install the JDK.

With your browser, go to the Oracle website:

```
www.oracle.com/technetwork/java/javase/downloads/index.html
```

From here, click the Java JDK download button.

Then accept the license agreement, and then select Windows which must be installed on a 64-bit computer and operating system or for macOS.

Your browser will prompt you to save the installer to your computer. Save and run the installer using the defaults for the configuration.

Remember the location of where the JDK is installed; you will need this later on to connect IntelliJ IDEA to the JDK.

Install IntelliJ

Now we need to install IntelliJ IDEA.

With your browser, go to the JetBrains website:

```
www.jetbrains.com
```

Under Tools, look for the column IDEs, and select IntelliJ IDEA. We will use the Community Edition, which is free.

Your browser will prompt you to save the installer to your computer. Click Save to continue and download.

Run the installer and accept the defaults for all the settings.

That's it! IntelliJ IDEA and the Java Development Kit are successfully installed!

SPRINT 3

Setting Up GitHub

When you code, you need to have a location to store it and share that code with others to work with it and review it. Tools called code repositories are ideal ways to do this, and there are several different types that are available for developers to use; one of the most popular is GitHub, and we will explore it here. GitHub, in the context of this book, is a tool that you can use on your own. I'm introducing it here, because as you get more familiar with programming, you'll find it to be an indispensable tool.

GitHub

The one that we will use is called GitHub. GitHub is a free public code platform that developers use to build, archive, and manage coding projects for individual or group collaboration. It is built on the Git technology that is ordinarily used for private, internal projects, but GitHub builds their public platform on Git to make it available to anyone.

To create a GitHub account is simple. Go to `www.github.com/` and create an account there. For our use in class, please create your account as your Serra email address so I can easily identify your account.

When you create your account, be sure to go to GitHub Desktop to download the desktop client for Windows or macOS. You can download the installer here: `https://desktop.github.com/`.

Install the client and sign in to your account. You can then create a repository to keep track of the changes you make in your classwork projects. You can create a single project to capture all of your homework assignments, or you can create an individual repository for each project. Whichever you choose is up to you.

D. Winnie, *Essential Java for AP CompSci*, https://doi.org/10.1007/978-1-4842-6183-5_3

How GitHub Works

GitHub and Git create a special directory that it uses to keep track of all changes that you make to the files inside of it. When you make a change to a file, it tracks what the change was and notes that.

This is different than a backup. A backup creates a copy of everything that is in a folder. A change in Git and GitHub only records what changed, which can save a lot of space and is much faster to process.

There are many advanced workflows and automations that you can tie into Git and GitHub, but we will only be using the platform to store and share projects and homework assignments.

There are ways to tie in GitHub directly into the IntelliJ IDEA IDE, but for simplicity, we can use the GitHub Desktop client to make, track, commit, and sync changes to our GitHub account.

Lifecycle of a Repository

A repository, or repo, starts with the master branch, sometimes known as a trunk. For individual developers, or small teams, projects might only use a master branch and never create alternate versions of it.

As the program evolves and new changes are made, they are saved to the repository by making a commit. A commit contains the changes to the code and a brief comment and description made by the developer on what changed for future reference. Over time, there will be dozens or hundreds of commits on a repository.

At some point, a repository will need to split in some way. Either there needs to be simultaneous work and they don't want to have code conflict with others, or a project needs to maintain an existing version and they want to do something else with it. In that case, a branch is created from the main trunk, and development can happen in parallel with the original code.

At some point, a branch will need to rejoin the main development, and this is done by merging that branch back into the master trunk.

For open source repositories, it is common for a developer to find a helpful framework and want to add it to their own development environment. To do that, they would fork the repository, creating a copy in their own environment to work with and potentially make changes.

When a developer makes changes or improvements to a forked repository, they might want to contribute those changes back to the main repository they forked from. To do that, they submit a pull request back to the original repository. As part of this pull request, the developer outlines the changes that were made, and the person who receives the requests can compare the proposed changes to the repo by performing a diff that shows the changes side by side from one another.

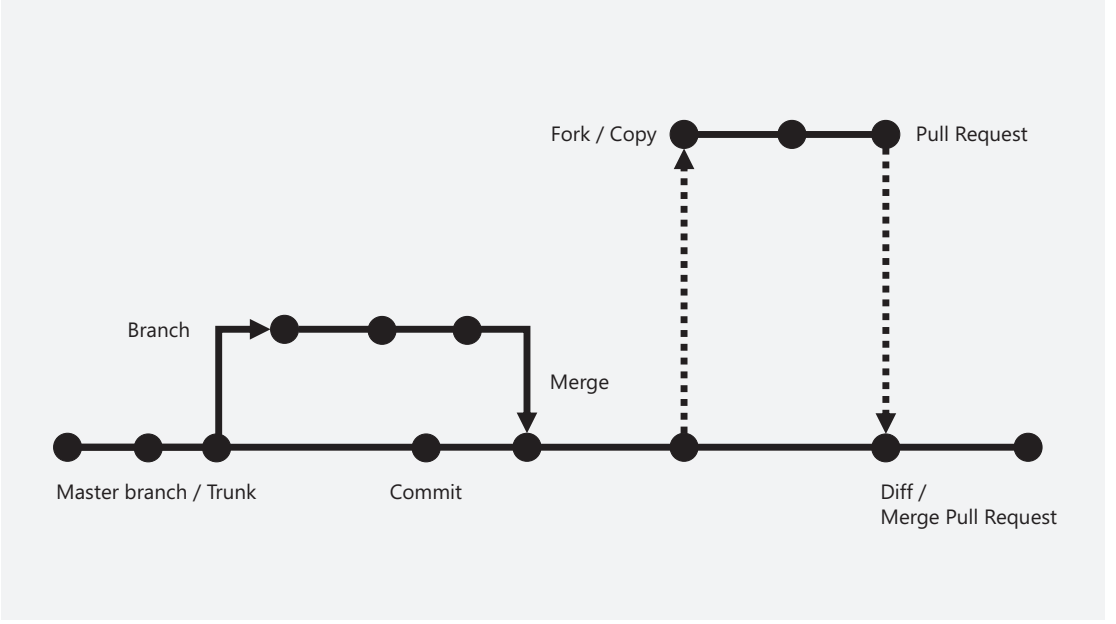

Figure 3-1. *Repository lifecycle*

Repositories can vary in complexity from something very simple with a single or a handful of developers to a large open source project with tens of thousands of developers. Code repos are the core of how software developers work together on projects of any size.

SPRINT 4

Programming Languages

Programming languages started as an abstract concept and then adapted and changed over time. From the origins of machine language, switch input, extending into the early days of simple line-by-line coding like COBOL or BASIC to object-oriented languages like Java and C#, to quantum computer languages like Q#, the evolution of programming languages has developed over the course of many decades.

Origin of Programming

In the early 1840s, Charles Babbage proposed a machine called the Analytical Engine. It was only a proposal—no actual machine was built, but one inventive woman by the name of Ada Lovelace decided to write an article that provided detailed instructions on how to represent Bernoulli [ber-noo-li] numbers, a recursive equation based on number theory, on the Analytical Engine. This article is considered to be the very first computer program.

Since then the devices that can be programmed went from theoretical to physical, manual to automatic, analog to digital. With each evolutionary step, the way we program computers needed to evolve as well.

With the birth of mainframe computers, data processing required instructions to be sent to the machine to process and interpret the instructions from the programmer. This was then applied to data to organize and analyze the data. Instructions were entered through a keyboard, but without the benefit of a monitor, so everything was done through printouts on paper or storing data in the form of punched holes on cards that were stored in stacks. If you look carefully at text encodings and at some programming languages, you'll see things like "carriage return" or "print" that are carryovers from those printer days from decades ago.

D. Winnie, *Essential Java for AP CompSci*, https://doi.org/10.1007/978-1-4842-6183-5_4

As computers got smaller and more powerful, more languages were created. Languages also were created to serve specific types of projects and industries like mathematics and science, data storage, and graphics.

Today, we work with programming languages that can serve many different purposes. In fact, a programmer often needs to use multiple programming languages to get a project completed. As languages have evolved, they have become specialized to complete specific tasks. As a programmer, you will use the best languages for specific tasks and combine them together to create your project.

The programming languages you learn today will continue to evolve and change in the future. With future waves of new technology, new languages will be developed to allow programmers to drive even more innovation.

Forms of Programming

As programming has evolved over the decades, the types of programming you can do have changed as well. Depending on what you want to do, there are different types or forms of programming languages that work in different ways. As a programmer, some forms of programming give you direct access to the computer processor, while others abstract the hardware into more human language that needs to be translated or converted into the native language of the hardware. Here are some example forms of programming that you might encounter.

Machine Language

Machine languages allow programmers to code instructions directly to the processor or hardware. Processors can be programmed by sending sequences and patterns of ones and zeros through the processor to enable actions to take place. As a result, the code that is entered by the programmer is almost natively written. Assembly language, which is an abstraction of machine language, uses special codes to modify processor registers and perform functions.

Interpreted

Interpreted languages are readable by humans more easily than assembly or machine languages. The programmer writes the code and then runs it. A component called an interpreter reads each line of code and then “interprets” it into native instructions for

the computer. The process is much slower than machine language since the interpreter needs to convert each instruction provided by the programmer, even if it repeats a line of code multiple times—it needs to interpret it each time. JavaScript is an example of an interpreted language. A programmer can stop the execution of the program, make a change to a line, and then run it again without any other steps.

Compiled

A compiled language takes instructions written by a human and sends that code to something called a compiler. A compiler takes the program instructions and converts it to binary bytecode or native code for the hardware and creates a program called an executable. This program is native to the hardware and operating system and can't easily be converted back to the original programmed instructions. With the code now in the native computer format, it runs much faster than interpreted code, but if you need to make a change, you need to adjust the original program instructions and recompile it to create a new executable. If you are creating programs for multiple types of processors, you need to compile unique versions for each native instruction code for the target platforms. C is an example of a compiled language.

Object-Oriented

Object-oriented programming, or OOP, treats everything as an object. An object can store values; perform actions, called methods; and accept and return values. An object is defined using a template, called a class, that defines what an object can do. A programmer can then create an instance of that class that has all of the capabilities defined by the class. Java and C# are examples of object-oriented languages.

Data

There are languages that are specifically designed at working with data. One example is SQL, pronounced as see-quel, which is a language designed for working with databases. This is a query language, where you ask a database a question and it gives you a set of data as a result. You can use SQL to combine multiple databases together to create combinations that you can then analyze. R is another example of a data language. R is designed for statistical computing and graphing.

Functional

Functional programming approaches programming in a much different way. Think of it like this: In a traditional programming language, which is called imperative, you are defining the state of a value, object, or component. You create and define tasks to complete, called an algorithm, that go from beginning to end. Functional programming isn't bound by an algorithm. In a functional language, you perform transformations on values, like a function in mathematics. You take a value or object and modify it, with the ability to string multiple transformations together using functions. Functional languages focus on what needs to be done, without much care for how it is performed. Examples of functional languages include Haskell, Scala, and F#.

Scripting

Operating systems regularly need to execute commands to configure servers, install software, or perform maintenance. To automate that process, there are scripting languages that allow systems like Windows, Linux, and macOS to save common commands as a script that can be run multiple times or distributed to multiple computers. PowerShell, perl, and bash are examples of scripting languages.

As you can see, there are multiple types of programming languages that you can work with to perform different types of tasks. This course will focus on concepts found in object-oriented, compiled, and interpreted languages like JavaScript, Java, C#, and Swift.