Introduction to Java

Lecture Notes for AP Computer Science A

Daniel Szelogowski

2022

About

This text consists of a set of succinct lecture notes introducing computer programming in Java for AP Computer Science A students. These notes are intended to act as a course companion, not necessarily a standalone text for self-study unless the reader already has some familiarity with studying programming languages. The text covers all ten units of the AP Computer Science A exam from the Fall 2020 edition of the Course and Exam Description (the most recent as of the publication of this work), a brief introductory unit on setting up and installing Java, and a code editor, and a bonus unit on extra concepts from beyond the AP subset that are especially useful in real-world applications. Some key topics that may be covered include basic programming concepts, such as variables, data types, control structures, and functions, Object-Oriented Programming principles (encapsulation, inheritance, and polymorphism), using Java libraries, parsing data, algorithms and data structures (searching and sorting), and advanced Java features such as generic typing and Functional Programming. Overall, this text provides a thorough foundation in programming concepts and techniques, as well as the skills and knowledge needed to succeed in the AP Computer Science A course and exam, including pitfalls that students should learn to avoid.

For errata and supplemental programming assignments, projects, labs, and competition problems, visit **http://danielszelogowski.com/research.php#apjavabook** (see the *Langdocs* folder for assignment descriptions and the *Langdat* folder for data files). If you find any errors, please use the contact form on my homepage or email **dan@zerodevelopment.net**, and I will add them to the errata.

First Printing: 2022

ISBN 978-1-387-37972-9

Lulu Publishing

Janesville, Wisconsin 53545

www.danielszelogowski.com

Contents

Unit 0 - Java Development Setup and Installation

Java is a popular **Object-Oriented Programming (OOP)** language that runs on a wide variety of platforms. It is commonly used for building web and mobile applications, as well as developing desktop and server-side applications. Today, it is especially popular because it is easy to learn and transfer concepts to other programming languages, has a large and active community of developers and continues to be used in enterprise applications due in part to it being cross-platform (i.e., usable on any operating system where the Java Runtime is installed) since it runs on a compatibility layer known as the **Java Virtual Machine (JVM)**. Before doing anything, you first need to install Java and a code editor of your choice.

To set up Java on your computer, you need to follow these steps:

1. Download and install the **Java Development Kit (JDK)**. The JDK is a software development kit that includes the tools you need to compile and run Java programs. You can download the JDK (not the Java Runtime Environment/JRE) from the Oracle website (https://www.oracle.com/java/technologies/javase-downloads.html) or from OpenJDK (https://openjdk.org/)
2. Once the download is complete, run the installer and follow the prompts to install the JDK
3. After the installation is complete, you need to set the PATH environment variable to include the directory where the JDK was installed, which is necessary to allow the **Java Compiler** (`javac`) and **Java Runtime** (`java`) to be found from any directory on your machine. First, run the command `java -version` in your command line/terminal; if it runs successfully, you can skip the next two steps; otherwise (if you get an error like *"command not found"*), follow the next steps to properly set up the PATH variable for Java
4. To set the PATH environment variable on **Windows**, open the Control Panel, click on "System and Security," and then click on "System." From there, click on "Advanced system settings" and then click on the "Environment Variables" button. Under the "System variables" section, find the "Path" variable, and click on "Edit." Add the path to the JDK bin directory to the end of the "Variable value" field, separated by a semicolon
5. To set the PATH environment variable on **macOS** or **Linux**, open your terminal and enter the following command:

```
export PATH=$PATH:/path/to/jdk/bin
```

Replace `/path/to/jdk/bin` with the actual path to the JDK bin directory on your machine.

Once you have Java set up, you can use it to write and run programs using your preferred development environment or text editor.

Installing a Text Editor or Development Environment (IDE)

To start programming, you need some way to edit your code; either a **Text Editor** or **Integrated Development Environment (IDE)** — preferably some program with syntax highlighting to make it easy to read and write Java code.

To install a text editor or development environment for Java, you will need to follow these steps:

1. Download and install the JDK
2. Choose a text editor or development environment and install it
3. Set up your Java development environment. Follow the instructions provided by your text editor or development environment to set up your development environment, including specifying the location of the JDK and configuring any necessary project settings

Once you have completed these steps, you should be ready to start writing and running Java programs. Depending on your operating system, you can find many resources for installing Java specifically for your computer setup; note that while some operating systems include the **Java Runtime Environment (JRE)**, you still need the JDK as well. If you installed Java manually (not through an IDE), you can always run the following command in your terminal/command line to verify that Java has been installed properly: `java -version`

Two of the most popular software development programs for Java include:

- **IntelliJ (Community Edition is free):** https://www.jetbrains.com/idea/download/ — very useful since it will also install the **Java Development Kit** for you
- **Visual Studio Code:** https://code.visualstudio.com/download — you'll also likely want a few extensions like **Code Runner**[1] and the **Extension Pack for Java**[2]

While not nearly as popular today, **Eclipse** (https://www.eclipse.org/downloads/) used to be one of the most used free Java editors for quite some time and still has a large following. Another free (archaic) Java IDE often used in schools is **BlueJ** (https://www.bluej.org/), though its interface is not quite as friendly. You can also use various online editors such as https://replit.com to make a new Java project and run the code instantly, which is a useful option for cloud or browser-based devices such as Chromebooks. If you're on a Unix-based operating system, check out **Nano**, **Vim**, **Emacs**, or **Neovim** for terminal-based code editors.

[1] https://marketplace.visualstudio.com/items?itemName=formulahendry.code-runner
[2] https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-java-pack

Once you have Java and an editor installed, make a new file in the project directory of your choice named **Hello.java** and paste in the following code:

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

Find the **"Run"** button on your code editor to compile and run your program, or change the directory in your terminal to the folder you made the file in and run the command `java Hello.java` to execute your new program.

All Java programs follow this same set of steps:

1. Make a new file named `Your_Filename_Here.java` (name it whatever you want with the `.java` extension)
2. Type out the following **code template** (where `Your_Filename_Here` matches the **exact** filename minus the extension, case-sensitive):

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

  }
}
```

3. Write some code in the blank line(s) between the `public static void main(String[] args) {` and the penultimate `}`
4. Run your program either with your editor's Run button or by typing `java YOUR_FILENAME_HERE.java` in the command line
5. Repeat!

You can find a brief tutorial on writing and running your first program at https://youtu.be/xGy5dKKFF3Y. Note that many of the code examples provided are "snippets" of programs (i.e., parts of the code necessary to run a program) rather than every detail (such as the `public class` and `public static void main...` declarations). For all intents and purposes, assume that any code snippets may be run by pasting them into a new program using the aforementioned template in the blank space in the middle. For example:

To print out some text to the console, use ***`System.out.println("YOUR TEXT GOES HERE IN THE QUOTES");`****, making sure each line ends with a semicolon* ***;****:*

```java
System.out.println("Hello, world!");
System.out.println("My name is NAME");
System.out.println("This is a simple Java program!");
```

This code could be run in a Java file named **PrintingTest.java** for example, by pasting the code snippet into our template as follows:

```java
public class PrintingTest {
  public static void main(String[] args) {
    System.out.println("Hello, world!");
    System.out.println("My name is NAME");
    System.out.println("This is a simple Java program!");
  }
}
```

Also, note that whenever we enter a new set of curly braces `{}`, we typically indent our code one tab space (with the **Tab** key) or either 2, 4, or 8 spaces (with the **Spacebar** key).

Setting up Git and GitHub for Version Control

One of the best ways to organize programming portfolios and large software projects is by using the **Git Version Control System (VCS)** (https://git-scm.com/) along with the website **GitHub** (https://github.com). Git is a version control system that allows you to track changes made to files and coordinate work on those files among multiple people, and GitHub is a web-based hosting service for Git repositories that provides version control, collaboration, and project management features. There are many good online tutorials on setting up various text editors and IDEs with Git, but if you simply want to get started right away, you can check out the following video on using the web editor https://replit.com/ with GitHub: https://youtu.be/sTh6B-KyCjA

You can also check out the official documentation for VS Code[3] or IntelliJ[4] to find steps on connecting to and/or sharing remote repositories.

In general, to set up Git and GitHub, you will need to follow these steps:

1. Install Git on your computer: You can download the latest version of Git from the official website (https://git-scm.com/) and follow the instructions to install it on your computer
2. Create a GitHub account: Go to the GitHub website (https://github.com/) and sign up for an account
3. Configure Git: After installing Git, you will need to configure it with your name and email address. This is important because every Git commit uses this information to identify you as the author. You can do this by running the following commands in your terminal:

```
git config --global user.name "Your Name"
git config --global user.email "your-email@example.com"
```

3. Create a new repository: A repository is a collection of files that are tracked by Git. To create a new repository on GitHub, click the "New" button on the top right of the dashboard. Enter a name for your repository and a short description, and then click the "Create repository" button

[3] https://code.visualstudio.com/docs/sourcecontrol/github
[4] https://www.jetbrains.com/help/idea/using-git-integration.html

4. Clone the repository: Cloning is the process of creating a local copy of a remote repository on your computer. To clone your newly created repository, go to the repository page on GitHub and click the "Clone or download" button. Copy the URL of the repository, and then run the following command in your terminal:

```
git clone <repository-url>
```

This will create a new directory with the same name as the repository and download all the files from the repository into it

5. Make changes and commit: Now that you have a local copy of the repository, you can make changes to the files and commit them to the repository. To commit your changes, use the following commands:

```
git add <file>
git commit -m "Commit message"
```

The `git add` command stages the changes you have made to the file, and the `git commit` command records the changes and adds a message describing the changes

6. Push to GitHub: To upload your local commits to the remote repository on GitHub, use the following command:

```
git push origin master
```

This will upload your commits to the **master** branch of the repository on GitHub.

These are the basic steps for setting up Git and GitHub for version control using the command line. You can find more information about using Git and GitHub in the official documentation.

Unit 1 - Primitive Types

In programming, whenever we store data in a variable (like a mathematical variable where values can change or be immutable like π), we divide that data into various classifications — is it a number, text, or something else? Numbers, two of our **primitive** types are separated by computers into being either whole numbers or numbers that may have a decimal. Individual characters of text are also **primitive**, but not used in the AP subset; however, combinations (or *strings*) of characters do have a special **class** of their own, known as a **String** *object*. There are three primitive types that the AP subset focuses on, along with one additional useful type that Java provides:

- **int**: any whole number (Integer), positive or negative
- **double**: any Real number (with or without a decimal), positive or negative; short-hand for "double-precision floating-point number"
- **boolean**: either `true` or `false`
- ~~**char**~~ (***WARNING***: *NOT IN AP SUBSET*): a single (keyboard) character wrapped in apostrophes/single quotes, including numbers, letters, special characters, and escape characters (like `'\n'` for a newline, `'\t'` for a tab space); not tested on the AP subset but can be extremely useful

While not a *primitive* type, the **String** *class* is extremely important — i.e., a "string" of characters, wrapped in quotes, such as `"Hello, world!"`.

Variables, equivalent to a mathematical variable, are defined in the sequence `<TYPE> <NAME> = <VALUE>;`, such as `int x = 5;`.

Variable names can start with letters or underscores (_), either of which can appear anywhere in the variable name. However, numbers may only appear at the END.

Some valid variable names include `_test`, `Te_ST_` `test_123`, `test`, and `TEST`. Note, however, we typically use an underscore before a name for private variables (inside a *class*) and all-caps for constants.

As well, we can define a constant variable using the **final** keyword:

```java
final double E = 2.71828;

// Or...
final char ENDL = '\n';
```

One other trick we can do is declare multiple variables, with or without instantiating them (providing initial values):

```java
int x, y, z;
x = 2;
y = 4;
z = 6;

// Or...
final double fedinctax = .15, statetax = .035, socsectax = .0575;
```

However, be careful with this, because if you do this on the AP exam and forget to provide a value, you've just caused a **"Not Initialized" error** which will cost you points on the question(s).

Basic Programs and Console Output

Every Java program must start with `public class FILENAME {...}`, where **FILENAME** is the exact name of the *.java* file, minus the file extension (case-sensitive). Recall that Java is a case-sensitive language, but not whitespace-sensitive, also.

Most of our programs will include a `main` function, with the signature `public static void main(String[] args) {...}` — this is where Java looks for what code we actually want to run. Hence, it's given the name "entry-point" by most compilers.

Each line of code must end in a semicolon (;) unless it ends in a curly-brace ({ or }).

For example, a file named *Hello.java* should look like the following:

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

To print to the console, we either use `System.out.println("...");` to print some text with a newline/line-break, or `System.out.print("...");` to print the text without the added line break.

To print a variable, we can either print it by itself, such as `System.out.println(xyz);` or append it to the end of or between strings: `System.out.println("num = " + num);` or `System.out.println("x = " + x + "\ty = " + y);`.

It's less common, but we can also use `System.out.printf("some formatting: %.3f\n", doubleval);` to print out variables with specific formats (see Format Specifiers)[5] — note that this function also does not append a line break unless you put "\n" or "\t" for a tab space. Likewise, if you wanted to print a double or String, or a mix of all, you simply write

[5] https://www.geeksforgeeks.org/format-specifiers-in-java/

out the string followed by a comma-separated list of the arguments you want displayed in order of the format specifier:

```java
int a = 50;
double b = 3.14159;
String c = "Hello!";
System.out.printf("%d   %.3f   %s\n", a, b, c);
// Prints out:
// 50   3.142   Hello!
```

A few of the most important format specifiers to remember are:

- `%d`: int
- `%f`: double (`%.#f` to round to # decimal places)
- `%c`: char
- `%s`: String

This technique of formatting a String (alongside the same with the String.format() function) is known as ***string interpolation*** — we specify inside the string what type of variable we want in that spot in the text, and pass in the variable(s) as arguments in the order we want them to appear.

You can also use the `","` format specifier to separate a large number with commas every third digit from the right (like hand-written numbers):

```java
int x = 1000000;
System.out.printf("%,d\n", x);
// Prints out:
// 1,000,000
```

You can even write out large numbers[6] with the `_` separator:

```java
double salary = 1_234_567.890;
System.out.printf("%,.2f\n", salary);
// Prints out:
// 1,234,567.89
```

Comments

There are two ways to leave a comment in Java (i.e., some text that only we can see; the compiler will delete it from the actual program) — either a line comment (by itself or appearing at the end of a line of code) or a block comment (spans multiple lines or can appear within a line of code):

- `// This is a single-line comment`
- `int x = y;  // set x equal to y`
- `double z /* z will be a placeholder */ = 5;`

[6] https://docs.oracle.com/javase/7/docs/technotes/guides/language/underscores-literals.html

or:

```java
/*
This
comment
spans across
multiple lines
*/

/*
 * This comment
 * also spans
 * multiple lines
 */

/*
* This comment also spans multiple lines
*/

// For documentation, these also tend to look like:
/**
 * Sets the tool tip text.
 *
 * @param text  the text of the tool tip
 */
public void setToolTipText(String text) {
// You will see these a lot on AP exam questions
```

User Input

User input is also not on the AP subset, but again, is extremely useful to know. The **Scanner** class provides a great number of input functions which we will use for both user and file input.

First of all, we need to import the class from the **java.util** library, either using `import java.util.*;` to pull the entire library, or `import java.util.Scanner;` — either way, this should be your FIRST line of code in an input program.

For user input, we make a new **Scanner** object and give it a *name*, like "input": `Scanner input = new Scanner(System.in);`

Now, we have four major methods that the class provides:

- `NAME.nextInt()`: get an `int` from the user/file
- `NAME.nextDouble()`: get a `double` from the user/file
- `NAME.nextLine()`: read an entire line of text (include the line-feed) as a `String`
- `NAME.next()`: reads one word (token) at a time from a line of text separated by a space (see Example 1[7])

For example, a file named *Prog52a.java*:

```java
import java.util.*;
// Full Name
// Lang52a
// MM/DD/YYYY

public class Prog52a {
  public static void main(String[] args) {
    // Input Section
    Scanner input = new Scanner(System.in);
    System.out.print("What is the length of the rectangle: ");
    int len = input.nextInt();
    System.out.println();  // Print a blank line

    System.out.print("What is the width of the rectangle: ");
    int wid = input.nextInt();
    System.out.println();

    // Calculation Section
    int area = len * wid;
    int perim = len + len + 2 * wid; // Numerous ways to calculate

    // Output Section
    System.out.println("The area of the rectangle is " + area);
    System.out.println("The perimeter of the rectangle is " + perim);
  }
}

/*
What is the length of the rectangle: 5
What is the width of the rectangle: 6

The area of the rectangle is 30
The perimeter of the rectangle
*/
```

[7]https://www.javatpoint.com/post/java-scanner-next-method

Elementary Mathematics

Java supports numerous elementary math operations, as well as trigonometric and algebraic functions provided by the **Math** class.

Arithmetic Operators

Java supports 5 different elementary arithmetic operations:

- `+`: plus
- `-`: minus
- `*`: times
- `/`: divide
- `%`: modulus (or MOD) — returns the remainder from long (integer) division

Remember that the result of these operations returns the same **type** as the variables; an `int + int` operation will return an `int`. To get a `double`, **one** or **both** values must be defined as or *casted* as a `double`:

```
// Either...
int x = 17;
int y = 3;
double quotient = (double)x / y;
// Could also do:
// double quotient = (double)x / (double)y;

// Or...
double miles = 235;
double gallons = 14.3;
double mpg = miles / gallons;
```

Assignment Operators

Java supports shorthand assignment operators for each of its arithmetic operations:

- `x = y`: set *variable* on left side equal to *value* (or variable) on the right side
- `x += y`: shorthand for `x = x + y`; add `y` to the current value of `x`
- `x -= y`: shorthand for `x = x - y`
- `x *= y`: shorthand for `x = x * y`
- `x /= y`: shorthand for `x = x / y`
- `x %= y`: shorthand for `x = x % y`
- `x++`: shorthand for `x = x + 1` or `x += 1`
- `x--`: shorthand for `x = x - 1` or `x -= 1`

Unit 2 - Using Objects

Objects, by definition, are *instances* of **Classes**, many of which are built into the language which we frequently use (i.e., **Scanner**, **File**, **Exception**, etc.) given the convenience they provide. A ***Class***, put simply, is a collection of methods/functions and/or variables — **Scanner** provides us with all of the methods necessary to work with user and file input. One of the most important concepts to remember is that *objects* cannot use the traditional `==` operator — we need to use `obj1.equals(obj2)` instead, because `==` compares memory addresses; likewise, for making objects into Strings we need to use `obj.toString()` or we need implement a `toString()` method for the Class ourselves.

Objects

All types in Java are extensions of the **Object** class (for all primitive types and classes). This means that anything can be converted to an object through casting (known as **Boxing**) and back (**Unboxing**):

```java
int x = 5;

// Box int as Object
Object y = x;

// Unbox Object as int
int z = (int)y;

System.out.println(z);
```

We can also **Autobox** primitive types to their *Wrapper Class* and unbox them:

```java
// Creating an Integer Object with value 10
Integer i = new Integer(10);

// Unboxing the Object
int i1 = i;

// Print statements
System.out.println("Value of i:" + i);
System.out.println("Value of i1: " + i1);

// Autoboxing of character
Character chr = 'a';

// Auto-unboxing of Character
char ch = chr;

System.out.println("Value of ch: " + ch);
System.out.println("Value of gfg: " + gfg);
```

This behavior is also done automatically when using primitive types in **ArrayLists**.

Instantiation

Creating a new ***instance*** of a class is done using the `new` keyword nearly the exact same way as instantiating any other variable (i.e., in the form `ClassName var1 = new ClassName(classArgsHere);`). For example, we can make an instance of the **Scanner** class like so:

```java
// First, import from java.util;
import java.util.Scanner;
// ...
Scanner input = new Scanner();
int x = input.nextInt();
```

Calling Void Methods

A **void** method is simply a function that does not return anything (contrary to something like a mathematical function, which would return the result of its computation). Perhaps the most common example[8] is the `System.out.println(...)` method, which prints to the console but does not return a value that needs to be stored in a variable or used in a calculation, like `Math.sin(x)`.

To actually call a void method, we simply type out the method anywhere in the code that we want it to execute — typically on its own separate line since it cannot be used in a calculation:

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

[8]https://docs.oracle.com/javase/7/docs/api/java/io/PrintStream.html#println()

Calling Methods Without Arguments

Note that some methods may not require any arguments; that is, in the above example, we called `System.out.println()` by itself without passing in a String or value. In this case, based on the documentation, we can infer that this will print an empty string followed by a line-break (`"\n"`). This is very common within classes, such as the calculation functions we will write in our own classes. Another great example is `Math.random()`, which returns a random double within $[0.0, 1.0)$ but requires no arguments.

String Methods

The *String* class provides a great number of methods[9] for working with text data. It is worth noting that a String is simply a mask for an array of **chars**, which means we have the ability to work with either the whole text or loop through it one letter at a time. A few of the most important/useful string methods for the AP exam include:

- `x.length()`: returns the length of the string (i.e., the total number of characters)
- `x.isEmpty()`: returns **true** if the length of the String is 0 (i.e., the String is empty)
- `x.substring(end)` or `x.substring(start, end)`: returns a slice of a string either from [0, end) (where *end* is excluded) or [start, end) if a *start* is specified
- `x.indexOf(string)`: returns the **first** index of *string* ~~*or char*~~ in the String, otherwise returns -1
- `x.lastIndexOf(string)`: returns the **last** index of *string* ~~*or char*~~ in the String, otherwise returns -1
- `x.trim()`: returns a copy of the String with leading and trailing whitespace omitted
- `x.split(delimiter)`: returns an array of Strings split by either some delimiter (e.g., `","`, `" "`, or ";") or regular expression[10]; useful for splitting a sentence into an array of individual words, for example
- `x.toLowerCase()`: returns the String in all-lowercase form
- `x.toUpperCase()`: returns the String in all-uppercase form
- `x.contains(string)`: returns **true** iff (if and only if) the entirety of **string** is found in the String (case-sensitive)
- `x.compareTo(string)`: returns a lexicographical comparison[11] of the Strings — i.e., returns 0 if the Strings are exactly equal (case-sensitive); otherwise, returns 1 or -1
- `x.equals(string)`: returns **true** iff the entirety of **string** equals the String (case-sensitive)
- `x.equalsIgnoreCase(string)`: returns **true** iff the entirety of **string** equals the String (*NOT* case-sensitive)
- ~~`x.charAt(index)`~~ (***WARNING***: *NOT IN AP SUBSET*): returns the char at position *index* (within [0, `x.length() - 1`])
- ~~`x.toCharArray()`~~ (***WARNING***: *NOT IN AP SUBSET*): returns the String as an array of **chars**

[9] https://docs.oracle.com/javase/7/docs/api/java/lang/String.html
[10] https://cheatography.com/davechild/cheat-sheets/regular-expressions/
[11] https://www.w3schools.com/java/ref_string_compareto.asp

Other useful functions include `format(formatStr)`, `startsWith(string)`, and `endsWith(string)`, but these are not necessarily useful for the exam.

Primitive Types as Strings

One other important note is that if we want to convert/return a primitive type (**int**, **double**, **char**, etc.) as a String, we cannot use the `.toString()` function. Instead, we can simply concatenate the variable to an empty string or use `String.valueOf(num)` from the **String** class:

```
int x = 5;
String ret = "" + x;
char y = 'c';
String ret2 = "" + y;
int z = 15;
String ret3 = String.valueOf(z);
```

Wrapper Classes

The **Integer** and **Double** classes are wrappers to wrap the primitive types **int** and **double** in objects. This is similar to the idea of the **String** class, which is simply a wrapper for a `char[]` (i.e., a mask to make it easier to work with the entire text while also providing additional helper methods). For example:

```
// To create an instance of these classes,
// we can either provide a value or variable
Integer x = new Integer(5);
// Or...
int y = 150;
Integer z = new Integer(y);

// To unbox the object into its primitive form...
int a = z.intValue();

// Likewise, the same is true for the Double class
Double b = new Double(3.14);
double PI = b.doubleValue();
```

While this is not particularly useful in general cases, the most important concepts to know between these two classes are the following numeric constants they provide:

- `Integer.MAX_VALUE`: returns the maximum possible value that can be stored in an **int** or **Integer**
- `Integer.MIN_VALUE`: returns the minimum possible value that can be stored in an **int** or **Integer**
- `Double.MAX_VALUE`: returns the maximum possible value that can be stored in an **double** or **Double**
- `Double.MIN_VALUE`: returns the minimum possible value that can be stored in an **double** or **Double**
- `Double.NaN`: returns a constant holding a Not-a-Number (NaN) value of type double
- `Double.POSITIVE_INFINITY`: returns a constant holding the negative infinity of type double
- `Double.NEGATIVE_INFINITY`: returns a constant holding the negative infinity of type double

Though they aren't on the AP subset, these classes provide two very useful functions to parse numbers from strings:

- `Integer.parseInt(str)`: attempts to return an int from a given **String**
- `Double.parseDouble(str)`: attempts to return an double from a given **String**

You can also find the size (in bytes) of either data type using `Integer.SIZE` and `Double.SIZE`.

The *Math* Class

The **Math** class provide s constant definitions for π (Math.PI) and e (Math.E) a great number of methods, as seen in the documentation[12]. A few important ones include:

- `Math.abs(a)`: returns the absolute value of some variable `a`
- `Math.ceil(a)`: rounds `a` ALWAYS UPWARD toward the closest integer ($\lceil a \rceil$)
- `Math.exp(a)`: returns e^a
- `Math.floor(a)`: rounds `a` ALWAYS DOWNWARD toward the closest integer ($\lfloor a \rfloor$)
- `Math.log(a)`: returns the natural logarithm of `a` (i.e., $\ln(a)$ or $\log_e(a)$)
- `Math.log10(a)`: returns $\log_{10}(a)$
- `Math.max(a, b)`: returns the greater of two values
- `Math.min(a, b)`: returns the smaller of two values
- `Math.pow(a, b)`: returns a^b
- `Math.random()`: returns a random `double` in the range $[0.0, 1.0)$ — i.e., ≥ 0 but < 1
- `Math.round(a)`: rounds `a` by the standard elementary rounding definition: if a decimal is .5 or greater, round up (ceil); otherwise, round down (floor) — ($\lfloor a \rceil$)
- `Math.sqrt(a)`: returns $\sqrt{a}$

[12]https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

Random Integer Generation

To get a random integer, we can use the following formula with `Math.random()`: `(int)(Math.random() * (max - min) + min);`

We could make this into a separate function, like:

```java
public int randInt(int min, int max) {
  return (int)(Math.random() * (max - min) + min);
}
```

Hence, to get a random number between 10 and 20, we could do `(int)(Math.random() * (20 - 10) + 10);`. If you want the max to be inclusive, you could change it to `(int)(Math.random() * (max - min + 1) + min);`.

Float Rounding

Unfortunately, to properly round a number to n decimal places for printing is to use String formatting, i.e., using `System.out.printf("%.nf\n", val);` to round `val` to n decimals. You can also use the `String.format(str, value);` function in a similar manner:

```java
// Either...
double x = 32.33434;
String sf = String.format("value is %.2f", x);
System.out.println(sf);

// Or...
System.out.println("X rounded to two decimals: "
      + String.format("%.2f", x) + " is a simple example.");
```

If you want to *clamp* your decimal to a number of decimals in the actual variable itself, one way that works for a large majority of cases is to do `double xroundedtotwodecimals = Math.round(x * 100.0) / 100.0;`, where the number of 0's corresponds to the number of decimal places (i.e., 1000 is 3 decimal places, 10 is 1 decimal place, etc.) — however, note that there are equally many edge cases that this will fail due to the *roundoff error*.

Unit 3 - Boolean Expressions and *if* Statements

Conditional statements, commonly referred to as *if* statements, are snippets of code that allow us to do simple decision-making. We can use them to run a particular block of code *if* the given condition is **true**, and either do nothing or do something *else* if the original condition is **false**.

For example, consider the following *pseudocode*:

```
gasPrice = 3.75
if (gasPrice > 2.50)
  println("These gas prices are way too high!")
else
  println("Gas is pretty cheap for once")
```

Boolean Expressions and Conditions

Recall that the **boolean** data type specifically stores ONLY the values `true` or `false`. Knowing this, we can do simple conditional expressions, or *boolean statements*, and store the result inside of a variable (or use them directly inside an *if* statement):

```
int x = 17;
boolean isEven = x % 2 == 0;
System.out.println(isEven);  // Prints "false"

// or...
boolean isLt20 = x < 20;
System.out.println(isLt20);  // Prints "true"
```

Conditional Operators

Java supports the following conditional/relational operators (use ONLY for primitive types, otherwise, these will compare the memory address of an object):

- `==`: exactly equal to
- `!=`: not exactly equal to
- `>`: greater than
- `<`: less than
- `>=`: greater than or equal to
- `<=`: less than or equal to

Note the important distinction between `==` being our **equality** operator, versus `=`, our **assignment** operator.

Conditional (*if*) Statements

The *if* statement is the simplest form of conditional statement. It's usage is the same as written language — *if* a condition is **true**, then we'll run the code inside its curly braces. Otherwise, we'll do something else or nothing at all.

For example:

```java
int x = 5;
int y = 5;

if (x == y) {
  // The code between these curly braces
  // will only run if x is exactly equal to y
  System.out.println("x and y are equal!");
}
// If x and y were not equal, nothing would be printed.
```

if-else Statements

By attaching an *else* block/statement to the end of an *if* statement, we can create blocks of code for both possible conditions: **if** the condition is true, **otherwise**, do something **else**. Note that **we can only have exactly ONE *if* and ONE *else* statement per condition**.

For example:

```java
if (x > y) {
  // The code between these curly braces
  // will only run if x is greater than y
} else {  // Otherwise...
  // The code between these curly braces
  // will only run if x is less than or equal to y
}
```

Using *else if* Statements

The *else if* statement allows us to add additional chains of conditions to our *if* statement — if the first *if* statement is false, we'll check the next *else if* statement to see if it's true. If it's still false, we'll either:

- Check another *else if* condition,
- Fallback to an *else* statement (if one is provided), or,
- Do nothing at all (if there are no more *else if* or *else* statements.

For example:

```
if (x > y) {
  // This code will only run if x is greater than y
} else if (x == 10) {
  // This code will run if x is equal to 10 and less than or equal to y
} else {
  // This code will run if x is less than or equal to y but not equal to 10
}
```

We can have as many `else if` conditions as we want, but only one `if` and (optionally) `else`:

```
if (condition1) {
  // ...
} else if (condition2) {
  // ...
} else if (condition3) {
  // ...
} else if (condition4) {
  // ...
}
```

Compound Conditions and Logical Operators

More often than not, one condition is not enough to validate if we should run some snippet of code — a realistic example would be logging in to a website: `if (username is correct AND password is correct) then log in`. We create these *compound conditional statements* using operators for **boolean operations** such as `AND`, `OR`, and `NOT`.

Java provides 3 logical operators for compound and negated conditions:

- `&&`: AND
- `||`: OR
- `!`: NOT

That is, we can combine multiple conditions in the same boolean statement, for example:

```java
int age = 18;
int time = 2000;

if (age == 18 && time >= 2000) {
  System.out.println("You're an adult and it's past 8 PM, "
        + "might as well go to sleep!");
}
```

Likewise, we can also check if a condition is NOT true (i.e., false):

```java
if (!allHomeworkCompleted()) {
  System.out.println("Not all of your homework is completed. Back to it!");
}
```

Truth Tables

Truth tables are a method of breaking down compound conditions into each of their smaller parts to deduce their result based on all possible combinations of **true** and **false**. For example, the truth table for the statement `if (A or B) and (!A and B)`:

A	**B**	**!A**	**A or B**	**(!A and B)**	**(A or B) and (!A and B)**
T	T	F	T	F	F
T	F	F	T	F	F
F	T	T	T	T	T
F	F	T	F	F	F

Equivalent Conditions

One thing we can do to make our code easier to read and organize is to optimize conditions to their reduced equivalent, often seen in the technique known as **Guard Clauses**[13] (a technique often used to prevent/decrease nested *if* statements). By rewriting a condition as its opposite form, we can often drastically reduce code and/or increase readability (and possibly performance!).

For example:

- *Greater Than* `>` is the direct opposite of *Less Than or Equal to* `<=` (and vice versa), so `!(x > y)` is the same as `(x <= y)`.
- Not true `!true` is the same as `false`, and vice versa.

Also, check out De Morgan's Law[14] for one of the most important rules in *Discrete Mathematics* to get a better idea of how, when, and where this is applied!

[13]https://www.youtube.com/shorts/Zmx0Ou5TNJs
[14]https://blog.penjee.com/what-is-demorgans-law-in-programming-answered-with-pics/

Comparing Objects

Perhaps the most important occurrence of object comparison in most Java programs is that of comparing **Strings**. Since a **String** is NOT a primitive type (i.e., it's a *class* that we instantiate as an ***object***), we have to use the `.equals(anotherStr)` to check if two strings contain the same exact text, case-sensitive. Using `==` on an ***object*** checks to see if two things share the exact same **memory address** in RAM.

For example:

```java
String str1 = "Hello!";
String str2 = "Hello";
if (str1.equals(str2)) {
  System.out.println("String 1 is the same as String 2");
} else {
  System.out.println("String 2 is different than String 1");
}
// Prints "String 2 is different than String 1"
```

Unit 4 - Iteration

Loops are an extension of **if** statements, simply continuing to run the code within their scope (inside their curly braces `{}`) until the given condition becomes false. Noting this, we can use the same conditional and logic operators[15] as our **if** statements; however, the conditions for a loop tend to be much simpler.

The *while* Loop

A **while** loop is the closest type of statement to **if** statements — their job is to use nearly the same notation as an **if** statement, but continue looping the contained code *while* the given condition continues to be ***true***. It's more common to use these where a condition is not numeric (i.e., not looping through a list or using a counter of some sort), but instead where a condition is simply deterministic. The most common use case for a **while** loop would be to loop while a **Scanner** has data left to scan (e.g., from a file or for networking) or while some state has not changed (e.g., while a client is connected to a server).

The syntax is very simple:

```
while (<some_condition == true>) {
  // Run some code repeatedly until the condition becomes false
}
```

For example, a counter-based loop:

```
int counter = 0;
while (counter < 10) {
  // Run some code repeatedly, 10 times in total
  counter++;
}
```

Here, *counter* is acting as our **Look Control Variable (LCV)**; while we often use names like *counter*, *cnt*, *i*, *index*, etc. for our loop controller, ***lcv*** is a great alternative that is also easy for graders to read on paper.

[15]https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op2.html

The *for* Loop

A **for** loop is a shorthand, counter-based loop (i.e., rather than iterating specifically on a condition only, we iterate based on some counter variable). It allows us to declare a counter, check the condition for the counter, and increment/decrement the counter all in one line of code (similar to the **while** loop example above). For example:

```
for (int counter = 0; counter < 10; counter++) {
  // Run some code repeatedly, 10 times in total
  // (same as the while loop example)
}
```

Again, one must be careful (especially in hand-written code) with its use, but a one-line statement does not require curly braces:

```
while (true) System.out.println("Infinite loop");
// Or...
for (int lcv = 0; lcv < 10; lcv++) System.out.println(lcv);

// Many people will also indent the inner-code, in a Pythonic-style
for (int lcv = 0; lcv < 10; lcv++)
  System.out.println(lcv * 2);
```

If you need to iterate backward, there are two ways that work well; for example, printing the numbers from 10 to 0 (inclusive). First, the standard approach:

```
for (int i = 10; i >= 0; i--) {
  System.out.println(i);
}
```

In Unit 6, we *enhance* this syntax for usage with arrays to create the **for-each** loop.

Secondly, we can use the `--` operator in conjunction with the `>` operator to form a sort of *step-down* **for** loop:

```
for (int i = 11; i --> 0;) {
  System.out.println(i);
}
```

This form is much more useful for working with list-based data types such as **Strings** and **Arrays**:

```
// print a String in reverse
String text = "Hello, World!";
for (int lcv = text.length(); lcv --> 0;) {
  System.out.println(text.substring(lcv, lcv+1));
}
```

Iterating Through Strings

Iterating through each individual character in a **String** can be done in multiple ways (such as the `str.charAt(index)` method, but this is not on the AP subset), though we will only focus on the use of substrings. To select one character at a time, we can use `str.substring(index, index + 1)` in a **for** loop — remember that the `substring(start, end)` method does NOT include the *end* index in the slice of the String, so we can safely iterate from index 0 to the length of the String. For example:

```
// Print each letter of the text "Hello, world!" on a separate line
String hello = "Hello, world!";
for (int lcv = 0; lcv < hello.length(); lcv++)
  System.out.println(hello.substring(lcv, lcv + 1));
```

Iterating Through Files

While not on the AP subset, this is an extremely useful (and real-world) skill that happens to be easier in Java than in many other languages. However, this also requires that we understand basic exception handling, as working with files is a task prone to many errors. The `Scanner` class graciously provides us the ability to work with both user input AND files, so long as we pass in a `File` object rather than `System.in`.

For example, consider a data file named **mydatafile.txt** in a folder called **datafolder**, structured as multiple lines of 3 numbers (of varying types) separated by a space:

```
192 8 8125.00
203 8 3250.00
218 5 5000.00
235 5 5250.00
264 17 4150.00
```

Consider that each row in the data file contains 3 numbers: an **int**, an **int**, and a **double**. We can grab this data using a `Scanner` and a loop that scans `while (input.hasNext())` as follows:

```
try {
  Scanner input = new Scanner(new File("datafolder/mydatafile.txt"));
  while (input.hasNext()) {
    int num1 = input.nextInt();
    int num2 = input.nextInt();
    double num3 = input.nextDouble();
    // Do stuff with the data here
  }
} catch (IOException e) {
  System.out.println("Can't find data file!");
}
```

Exception Handling

The concept of **Exception Handling** refers to the process of responding to some event in the code that may be unintentional/unexpected, etc. Put simply, it is a way of preventing a program from crashing when a specified error occurs and instead allows us to decide what to do if an **Exception** is caught. **Again, this is not on the AP subset but is extremely useful for practical programs.** This is done using a ***try-catch*** statement, where `try` will attempt to run a block of (possibly) erroneous code, and `catch (SomeException ex)` tells us what to do if *SomeException* occurs while running the code in the `try` block. For example, catching a division by 0 error using the generic `Exception` type exception:

```java
try {
  // Run some code that may or may not throw an error
  int x = 5;
  int y = 0;
  int z = x / y;
  System.out.println(z);
} catch (Exception e) {
  System.out.println("Error: " + e.toString());
}
```

We typically use this when working with files to prevent an exception when a file is missing:

```java
try {
  // On some systems, may need to put "../data/prog285b.dat",
  // or use the absolute (exact) path if neither work
  Scanner input = new Scanner(new File("data/prog285b.dat"));

  while (input.hasNext()) {
    // ...
  }
} catch (IOException e) {
  System.out.println("Can't find data file!");
}
```

Unit 5 - Writing Classes

Classes are the heart of Object-Oriented Programming (OOP) — in languages like Java, we divide programs up into multiple components, called **Classes**, to keep code organized and emphasize code reusability. Put simply, a Class is a thing that contains other things — be it a set of functions (or *methods*), public/private variables, or simply our ***Main*** method (known as the *"Entry Point"*). Some classes may also be instantiated as *objects*, such as the **Scanner** class.

Writing Methods

A **method**, also known as a function or procedure, is a block of related lines of code that may contain any valid Java syntax. Methods may or may not also return a value (such as `Math.sin(x)` which returns a double, compared to `System.out.println()` which returns nothing). As such, we can flexibly use them as reusable snippets of code as desired, allowing us to make programs more efficient.

Methods generally follow the following format:

```
<access_level: public or private (or protected, but not on AP subset)>
<static or blank> <datatype (return type)>
<method_name>(<datatype arg1, datatype arg2, etc.>) {
  // ...
  // If not void: return datatype;
}

// or, simpler:
<access_level> <static/blank> <type> <name>(<type arg1, type arg2, ...>) {
  // Some code here
  // return varOfType;
}
```

The first line of every function is known as the **Method Header** or function signature since it tells us all the core details about the function (ideally). That is, consider the following method header:

```
public static int getRandomInt(int min, int max)
```

Clearly, this function can be accessed inside or outside of its defined class (*public*), it does not require its contained class to be instantiated first (*static*, just like we can call `Math.random()` from anywhere without making a **Math** class object), it returns an **int** value, the name is `getRandomInt`, and it takes two parameters: an `int` **min**, and an `int` **max**.

For example, a basic summation function (documented with docstrings/documentation comments):

```
/**
 * Return the sum of all numbers 1 to n, inclusive.
 * @param n The maximum range to sum up to
 * @return The summation of the range [1, n]
 */
public int summation(int n) {
  int sum = 0;
  for (int i = 1; i <= n; i++)
    sum += i;
  return sum;
}
```

If a method's return type is `void`, it simply means that we are not required to return anything — though, often it is used to exit the method early.

Documentation Comments

Comments are a fundamental need for large projects; unless you use extremely verbose and explicit variable, class, and method names, there will be parts of your code that do not make sense to other readers (including your future self!) especially. As well, we can use comments not only to explain what code does, but what unfinished code WILL do eventually (often marked with `// TODO: _____`) — for example, before even writing out any code, it is often useful to "sketch" out your program by breaking it down into smaller parts separated by comments describing what happens next.

Another standard in programming is the concept of **documentation comments**, which your text editor or development environment often uses to provide you with mouse-hover hints on what a *method* or *class* does and/or assists in code completion. These are multi-line comments that begin with `/**`, a `*` on each line, and end with `*/`. These can be as simple as describing who wrote the program and when, including HTML tags:

```
/**
* <h1>Hello, World!</h1>
* The HelloWorld program simply displays "Hello World!" to the console.
*
*
* @author  John Smith
* @version 1.0
* @since   2022-01-31
```

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

Often, especially on the AP exam, documentation comments are used to explain the conditions for a given function, as well as the expected conditions after a function has ran. These are typically defined as follows:

- **Precondition**: abc
- **Postcondition**: abc

For example, an AP-like question:

```
** Constructs a GameBoard object having numRows rows and numCols columns.
* Precondition: numRows > 0, numCols > 0
* Postcondition: each tile has a 50% probability of being set to on.
*/
public GameBoard(int numRows, int numCols)
{ /* to be implemented ... */ }
```

Javadoc Tags

The **javadoc** tool provides a large number of standard annotations for documentation comments, such as `@return`, `@param`, and `@throws`/`@exception`, among many others[16] such as the author/version/date annotations we saw previously. These allow us to specify things like descriptions of method parameters and/or what the method returns, if/what exceptions are thrown/what kind/when, deprecated methods, references to see something else, etc.

For example:

```
/**
 * This method is used to add two integers. It is
 * a simple class method used to demonstrate the
 * various common javadoc tags.
 * @param num1 The first integer parameter.
 * @param num2 The second integer parameter.
 * @return int The sum of num1 and num2.
 */
 public int addInt(int num1, int num2) {
   return num1 + num2;
 }
```

[16]https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html#CHDBEFIF

```
/**
 * This is the main method which calls the addInt method.
 * @param args Unused.
 * @return Nothing.
 * @exception IOException On input error.
 * @see IOException
 */
 public static void main(String args[]) throws IOException {
   int sum = addInt(10, 20);
   System.out.println("10 + 20 = " + sum);
 }
```

Class Design

A general-purpose Class (sometimes called a *data class* in other languages) typically consists of the following:

- **Private Data:** *instance* variables (i.e., variables specific to each object created of the class) that any method in the class can access but cannot be accessed outside the class directly; in good practice, we typically prefix these variable names with something like "my" or "_", such as `private int myAge` or `private int _age`.
- **Constructor(s):** sets up the private data (taking in arguments for some and setting the rest to some default value, like 0)
- **Mutator (Setter) Methods**: modify private data
- **Accessor (Getter) Methods:** return private data
- *Sometimes* **Override Methods (such as `toString()`):** overridden methods from the **Object** class — usually some accessor method(s)

The general layout for classes is as follows:

```
public class CLASS_NAME {
  // Private data
  private datatype myVar1;
  private datatype myVar2;
  private datatype myVar3;

  // Constructor
  public CLASS_NAME(datatype arg1, datatype arg2) {
    myVar1 = arg1;
    myVar2 = arg2;
  }

  // Calculation Function (Class Method and Mutator)
  public void calcVar3() {
    myVar3 = myVar1 * myVar2;
  }
  // Basic Class Method
  public datatype combineVar1AndVar2() {
    return myVar1 + myVar2;
  }

  // Getters (Accessor Methods)
  public datatype getMyVar1() { return myVar1; }
  public datatype getMyVar2() { return myVar2; }
  public datatype getMyVar3() { return myVar3; }

  // Setters (Mutator Methods)
  public datatype setMyVar1(datatype arg1) { myVar1 = arg1 };
  public datatype setMyVar1(datatype arg2) { myVar2 = arg2 };

  // Override Method
  public String toString() {
    return String.format("Var1: %FORMAT   Var2: %FORMAT   Var3: %FORMAT",
                  myVar1, myVar2, myVar3);
  }

  // Easy Print Method
  public void print() {
    System.out.println(this.toString());
  }
}
```

Finally, we can instantiate it as a new object anywhere as follows:

```
// In main, for example:
CLASS_NAME myClassObj = new CLASS_NAME(datatypev1, datatypev2);
myClassObj.calc();
System.out.println("Var3: " + myClassObj.getMyVar3());

// Modify the object's private values/fields/properties
myClassObj.setMyVar1(newdatatypev1);
myClassObj.setMyVar2(newdatatypev2);
myClassObj.calc();

// Print updated class
myClassObj.print();
// or, since the toString() method is implicitly called by print,
System.out.println(myClassObj);
```

We will break this class down to its components next.

Constructors

A **Constructor** is a simple function that sets up the private data for a Class. It is a special type of method called through the `new CLASSNAME()` keyword (rather than a direct function call) that is used to initialize an object. When an object is created, the constructor is called to initialize the object and allocate memory for it. The constructor sets the initial values for the object's instance variables (the typically-private class data) and performs any other necessary initialization tasks. Constructors are typically defined with the same name as the class, and they do not have a return type.

It is worth noting that we can have multiple constructors — a process called ***"overloading"*** — which allow us to take in varying arguments if any. Sometimes a constructor may not even have any arguments, typically known as the **Default Constructor** since it usually sets up *default* or placeholder values for the class.

For example:

```java
public class Dog {
  private int age;
  private String name;

  // Default constructor
  public Dog() {
    age = 0;
    name = "";
  }

  // Normal constructor
  public Dog(int dogAge, String dogName) {
    age = dogAge;
    name = dogName;
  }

  // Overload constructor
  public Dog(String dogName) {
    age = 0;
    name = dogName;
  }
}


// Now, construct the class in 3 possible ways
Dog dog1 = new Dog();
Dog dog2 = new Dog("Buddy");
Dog dog3 = new Dog(10, "Bucky");
```

Class Methods

Functions, or **Methods**, serve a variety of purposes. They may provide us with a way to modify some class data, perform some calculations independent of any class data (static methods), or simply return some class data. Just like with constructors, these can also be **overloaded** to consume different arguments. As such, methods in Java classes can generally be classified into one of three possible categories:

- **Accessor Methods**: return some private class variable (typically) or some result using them; may also be static
- **Mutator Methods**: modify some private class variable(s) and possibly return something; may also be static
- **Static Methods**: generally, perform some task independent of any class variables (that are not also static, at least)

Accessor Methods

In *Object-Oriented Programming*, **Accessor Methods** are functions that are used to retrieve the value of an object's instance variables using the `return` keyword, which sends the value outside of the function so we can store it in a variable. These methods, also known as *getters*, allow other objects to access the value of the instance variable without directly accessing the variable itself. This can be useful for enforcing ***encapsulation***, which is the practice of hiding the internal details of an object and exposing only the necessary information to other objects. Accessor methods are typically named using the "**get**" prefix followed by the variable name, and they return the value of the instance variable. For example, if an object has an instance variable named "*name*", the corresponding accessor method would be called `getName()`. As well, these may also be overloaded from a parent class, such as `Object` which all classes inherit from — providing us with the `toString()` method that we can customize the behavior of.

Let's look at a very simple example:

```java
public class Vector3 {
  private double _x;
  private double _y;
  private double _z;

  public Vector3(int x, int y, int z) {
    _x = x;
    _y = y;
    _z = z;
  }

  // Getters (accessor methods)
  public double getX() {
    return _x;
  }
  public double getY() {
    return _y;
  }

  public double getZ() {
    return _z;
  }

  // Overload accessor from Object class
  public String toString() {
    return String.format("X: %f\t Y: %f\t Z: %f",
                  this.getX(), this.getY(), this.getZ());
  }
}


// In main...
Vector3 position = new Vector3(0.5, 0.75, 3);

// Call the accessors and store their return value
double xPos = position.getX();
double yPos = position.getY();
double zPos = position.getZ();
System.out.println(position.toString());
// toString() is also called implicitly by println,
// so `System.out.println(position);` is sufficient

// Or, call the accessors and directly utilize their return values
System.out.printf("X: %f\t Y: %f\t Z: %f\n",
      position.getX(), position.getY(), position.getZ());
```

Mutator Methods

Similar to *getters*, **Mutator Methods** are methods that are used to modify the value of an object's instance variables. These methods, also known as *setters*, allow other objects to change the value of the instance variable without directly accessing the variable itself, which can also be useful for enforcing encapsulation. Setters seldom return anything — as such, we have a special type known as `void` which specifies that it returns nothing; sometimes we may even simply `return;` to exit the function earlier, such as if some condition does not pass. Mutator methods are typically named using the **set** prefix followed by the variable name, and they take a parameter that specifies the new value for the instance variable. For example, if an object has an instance variable named "*name*", the corresponding mutator method would be called `setName(String name)`; however, they may be also named something as simple as `calculate()`, which modifies numerous class variables at once. Again, these may also be overloaded to allow for varying arguments.

Let's look at an example:

```java
public class SimpleShape {
  private int myLength;
  private int myWidth;
  private int myArea;

  public SimpleShape(int length, int width) {
    myLength = length;
    myWidth = width;
    myArea = 0;
  }

  // Mutators
  public void calcArea() {
    myArea = myLength * myWidth;
  }

  public void setLength(int length) {
    myLength = length;
  }

  public void setWidth(int width) {
    myWidth = width;
  }

  // We can also combine these methods
  public void setLengthAndRecalculate(int length) {
    myLength = length;
    calcArea();
  }

  public void setWidthAndRecalculate(int width) {
    myWidth = width;
    calcArea();
  }

  // Overload mutator
  public void calcArea(int length, int width) {
    myLength = length;
    myWidth = width;
    myArea = length * width;
  }
  // Accessor
  public int getArea() {
    return myArea;
  }
}


// In main...
SimpleShape shape = new SimpleShape(5, 10);

// Call the mutators
shape.calcArea();
System.out.println(shape.getArea());

shape.setLength(3);
shape.setWidth(8);
shape.calcArea();
System.out.println(shape.getArea());

shape.setWidthAndRecalculate(16));
System.out.println(shape.getArea());

shape.setWidth(9);
shape.setLengthAndRecalculate(20);
System.out.println(shape.getArea());

shape.calcArea(15, 35);
System.out.println(shape.getArea());
```

Static Variables and Methods

The **static** keyword is used to define a static member of a class. A static member is a member of a class that belongs to the class itself rather than to any instance of the class. This means that a static member can be accessed directly on the class without the need to create an object of the class — i.e., it is *ready at compile-time*. More simply, the `static` keyword implies that the method/variable belongs to the **Class** itself, rather than a specific *object* or *instance* of that class, meaning that the class does not have to be instantiated for us to use it.

The `static` keyword can be used to define static variables, static methods, and static inner classes:

- **Static Variables**: variables that are shared by all instances of a class. They are typically used to store values that are common to all objects of the class, such as constants
- **Static Methods**: methods that can be called directly on the class, without the need to create an object of the class. They are typically used to implement utility functions or to create methods that can be shared by all instances of a class
- **Static Inner Classes**: classes that are defined within another class, and are marked with the "static" keyword. These classes are associated with the outer class, rather than with any specific instance of the outer class

In general, the `static` keyword is used to define members of a class that can be accessed without an instance of the class. It is often used to create utility classes (as part of a library/package/module) or to implement methods that are common to all objects of a class.

For example:

```java
public class Counter {
  // Static variable
  public static int count = 0;

  // Instance variable
  private String name;

  // Static method
  public static void incrementCount() {
    count++;
  }

  // Constructor
  public Counter(String name) {
    this.name = name;
    // Increment the static count variable
    incrementCount();
  }
}


// In main...
// Print the initial value of the count variable
System.out.println(Counter.count); // Output: 0
// Create an instance of the MyClass class
Counter counter = new Counter("John");
// Print the value of the count variable again
System.out.println(Counter.count); // Output: 1
```

Static Methods

A static method is a method that belongs to a class rather than an instance of the class. This means that a static method can be called directly on the class itself, without the need to create an object of the class. Static methods are typically used to implement utility functions or to create methods that can be shared by all instances of a class.

To define a static method, the `static` keyword is used before the return type in the method declaration. For example:

```java
public static void printHello() {
  System.out.println("Hello!");
}
```

This method can then be called directly on the class using `MyClass.printHello();`.

Static methods can only access static variables and other static methods. They cannot access instance variables or instance methods because they do not have access to a specific instance of the class. This is the cause of the infamous *"Non-static variable cannot be referenced from a static context"* error — which we often run into in our favorite `public static void main(String[] args) {}` method.

Scope and Access

Access refers to the level of accessibility of a variable or method within a class. If a method or class variable is `public`, then it can be accessed from anywhere — within or outside of the class, including in other classes/methods. Likewise, a method/variable marked as `private` can only be accessed within its containing class.

Similarly, ***scope*** refers to where exactly in the code something (usually a *variable*) is accessible. Typically, this refers to the set of curly braces that contain the variable. Consider the following example:

```java
public static void main(String[] args) {
  int lcv = 0;
  while (lcv < 10) {
    System.out.println(lcv);
    lcv++;
  }

  // Versus
  for (int i = 0; i < 10; i++) {
    System.out.println(i);
    if (i > 8) {
      int temp = i * 2;
      System.out.println(temp);
    }
  }
}
```

In the preceding code, we have three different variables, each with different *scopes*:

- `lcv` can be accessed from anywhere within `main` because it is declared outside of any block statements, such as a loop or condition (i.e, its "parent curly braces" are the entirety of `main`)
- `i` can only be accessed within the `for` loop because it was declared inside of the loop — so, anything inside the `for` loop can also access `i`
- `temp` can only be accessed within the `if` statement inside the `for` loop — its "parent curly-braces" are the `if` statement, whose "parent curly-braces" are the `for` loop; so, every time the loop starts a new iteration, a new `int temp` will be made inside the `if` statement

The *this* Keyword

The `this` keyword simply refers to the current class when working inside a constructor or method, typically. While it lacks any specific functionality, it is useful for eliminating any possible confusion when variable names are the same between a class variable and a method argument. For example:

```java
public class MyClass {
  private int x;

  public MyClass(int x) {
    this.x = x;
  }

  public static void main(String[] args) {
    MyClass thing = new MyClass(5);
    System.out.println("Value of x = " + thing.x);
  }
}
```

If `this` was not used in the example above, the program would print `0` instead of `5`, as the **argument** `x` would be assigned to itself, rather than assigning the **argument** `x` to the **class variable** `x`.

The `this` keyword can be used to:

- Invoke the current class constructor
- Invoke the current class method
- Pass an argument in the constructor call
- Pass an argument in the method call
- Return the current class object

Unit 6 - Array

Sometimes called a 1D Array or Vector, an **array** is simply a list of items (typically) of the same data type. Each item is assigned a unique **index**, or *slot*, acting as a sort-of "lookup table" for the items — however, indices begin counting at 0 in Java, rather than 1 in other languages such as Lua, R, Julia, and MATLAB/Octave.

Arrays are essentially lists of **objects** — they allow us to store multiple variables inside of one variable (i.e., an object that holds more objects). Hence, they can hold either primitive types/objects (`int`, `double`, `char`, etc.) or objects of classes (**String** and any other class we define).

Their mathematical equivalent, vectors, are typically denoted with a lowercase letter either with a right-pointing arrow above them or bolded (if the arrow sign is unavailable), such as $\vec{v}$ or $\mathbf{v}$. While in math these are typically seen as *column vectors* (where index 1 is the top of the vector and increasing indices are stacked below), we express these as *row vectors* (where index 1 starts from the left and increasing indices are stacked to the right).

Row vector (typical array):

$$\begin{bmatrix} 1 & \dots & m \end{bmatrix}$$

Column vector (most common in math):

$$\begin{bmatrix} 1 \\ \vdots \\ m \end{bmatrix}$$

Array/Vector Creation and Access

To declare an array, we use the `new` keyword just as if we were declaring a new Class object (like **Scanner**) along with the array *"bracket operator"* `[]`. However, we must also have a capacity in mind — regardless of if all slots are full, we need to specify a minimum size, which **CANNOT be modified** later. This is similar to a bookshelf or a permit-only parking lot: the parking lot has 100 spots, each numbered uniquely from 0 to 99; spots may not always be full, but we cannot add or remove spots without rebuilding the lot (theoretically).

Whenever a spot in the parking lot is empty, it has the value `null` meaning *nothing*, which we will look at a solution for below. ***For primitive types, Java will fill the empty slots with "0".***

The following syntax is used to declare an array:

```
DATATYPE[] arrname = new DATATYPE[arrLength];
```

We can also declare them **explicitly** using `{}`'s:

```
DATATYPE[] arrname2 = { var1, var2, var3, var4, var5 };
```

Creating an array with the explicit notation is a bit unusual unless the array is intended to be very short, as Java will make the capacity of the array the same as the number of items you provide.

Typically, we use a **for** loop to initialize an empty array. For example, an array of `int`:

```
int[] nums = new int[10];
for (int i = 0; i < nums.length; i++) {
  nums[i] = i;
}
```

Note the usage of `.length` to find the size of the array, and the `[index]` operator to assign and retrieve data from an array.

There are numerous ways to approach looping through arrays, either to **populate** or "fill" the list or print one or multiple arrays (of the **same size**) at once. For example:

```
// Make two arrays that store all numbers [1-5]
int[] list1 = {1, 2, 3, 4, 5};
// or...
int[] list2 = new int[5];
for (int lcv = 1; lcv <= 5; lcv++)
  list2[lcv - 1] = lcv;

// Print the arrays
for (int i = 0; i < 5; i++)
  System.out.printf("list1[%d] = %d,\tlist2[%d] = %d\n",
        i, list1[i], i, list2[i]);
```

Array Traversal

The term ***traversal*** refers to the act of *iterating* through an array — that is, checking multiple spots in an array using a loop, typically. This can of course be done with the `.length` property, or using a variable that is keeping track of the current size of the array (so as to not go into the "empty"/*null* slots, or some form of `0` for **primitive types**), but Java also provides us with a simpler syntax.

The *for-each* Loop (Enhanced For)

The **Enhanced For** loop, or **for-each** loop, is a simpler syntax for a **for** loop that iterates through an entire array without the need for an index. Instead, it gives us direct access to each object in the array sequentially.

A **for-each** loop uses the following syntax:

```
for (DATATYPE tempvarname : myarray) {
  // ...
}
```

The `:` operator equates to the word ***in*** (i.e., `for 'each' (DATATYPE currentNum in DataTypeArray)`); if we wanted to loop through all ints in an integer array, for example:

```
int[] mynums = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
for (int temp : mynums) {
  System.out.println(temp);
}
```

Consider the following approach to printing out the "name" and "age" properties from some **Person** class for all objects inside an array of **Person objects**:

```
Person[] people = new Person[50];
/* read in data file and populate array with 50 "Person" objects */

for (int slot = 0; slot < people.length; slot++) {
  Person currentperson = people[slot];
  System.out.println("Name: " + currentperson.getName());
  System.out.println("Age: " + currentperson.getAge());
}
```

In this case, we do not need the index beyond grabbing the current **Person** object that we want, so having the index variable is essentially useless. Instead, we can reduce the first two lines of code in the **for** loop into the *signature* for a **for-each** loop:

```
for (Person currentperson : people) {
  System.out.println("Name: " + currentperson.getName());
  System.out.println("Age: " + currentperson.getAge());
}
```

Null-Safe Traversal

One issue we often encounter with arrays is that we may not always have all slots of an array filled, just like in our parking lot example. When working with data files, we typically will keep a *counter* variable to track how many items are ACTUALLY in the array, rather than using the `length` property (which should really be called `capacity` instead). If we read in 50 objects from a data file, but our array has a capacity of 75, then the last 25 slots will all be `null`. If we try to call methods on these `null` objects, the program will crash — this is a huge problem with us using a **for-each** loop instead of a normal **for** loop with our *counter* variable.

Consider a data file (named **mycars.dat**, for example) that looks similar to the following, containing the ID number, make, model, and year for 50 cars:

```
1   Pontiac   Trans Sport   1995
2   Chevrolet   Impala    2007
.
.
.
50    Buick   Rendesvous    2003
```

Now, we can make a simple car class to contain those four properties:

```java
public class Car {
  private int myID;
  private String myMake;
  private String myModel;
  private int myYear;

  public Car(int id, String make, String model, int year) {
    myID = id;
    myMake = make;
    myModel = model;
    myYear = year;
  }

  public int getID() { return myID; }
  public String getMake() { return myMake; }
  public String getModel() { return myModel; }
  public int getYear() { return myYear };
}
```

Once we have our **data class**, we can read in all of the files from the data file. However, in case the size of the data file changes, we need to provide a buffer space (i.e., extra slots in the array) instead of counting the number of cars by hand and updating our code every time it changes — good luck trying that with a database! Let's make an array of **Car** objects that can hold up to 100 cars, similar to our parking lot example above:

```java
Scanner input = new Scanner(new File("mycars.dat"));
Car[] parkinglot = new Car[100];
int numcars = 0;
```

```java
while (input.hasNext()) {
  // Read in the information on each car
  int id = input.nextInt();
  String make = input.next();
  String model = input.next();
  int year = input.nextInt();

  // Make a new "Car object" using the information
  // of the current car we just read in
  Car parkedcar = new Car(id, make, model, year);

  // Increment the number of parked cars in the parking lot;
  numcars++;
}
```

Now, we have two options for safely printing out the information on every car without overflowing into our *buffer* space (the `null` slots): either use a **for** loop with our *counter* variable as the maxima or use a **for-each** loop but check to make sure that the car is not equal to `null` before attempting to call methods on it:

```java
// Approach 1 (for loop)
for (int spot = 0; spot < numcars; spot++) {
  // Grab the car parked at the current spot so we do not
  // have to write parkinglot[spot].getSomething() every time
  Car currentcar = parkinglot[spot];
  System.out.println("ID: " + currentcar.getID());
  System.out.println("Make: " + currentcar.getMake());
  System.out.println("Model: " + currentcar.getModel());
  System.out.println("Year: " + currentcar.getYear());
}

// Approach 2 (for-each loop with null-check)
for (Car currentcar : parkinglot) {
  if (currentcar != null) {
    System.out.println("ID: " + currentcar.getID());
    System.out.println("Make: " + currentcar.getMake());
    System.out.println("Model: " + currentcar.getModel());
    System.out.println("Year: " + currentcar.getYear());
  }
  // else { currentcar is null, so don't print or call anything }
}
```

Typically, for regular arrays, it's safer to just use a *counter* variable so we know that we won't overflow into the empty slots. However, what if we intentionally set a slot in the middle of the array to `null` like `parkinglot[5] = null;` (i.e., a parked car leaves the lot at random)? Then we have to account for null-checking even inside of our regular **for** loop! So, there are many factors we can take into account, but unless we know that we will not be removing anything from the array, it's a bit more flexible to just use a normal **for** loop with a *counter* — then we also have the index on hand, in case we need to modify our code to use it at some point later on.

Array Algorithms

Given the amount of data that exists, it is only sensible that computer scientists design standard algorithms for many of the most common needs and operations on such data. The purpose of these algorithms can generally be classified as follows:

- **Statistical or Property Analysis**: determining/computing the max, min, sum, average, mode, etc., checking for duplicates, or finding items with a particular property or that meet certain criteria
- **Searching**: efficiently finding an item in an array if it exists (linear search, binary search)
- **Sorting**: efficiently sorting an array in some order, usually ascending or descending (bubble sort, insertion sort, selection sort, quick sort, etc.)

Although technically part of Unit 7, searching and sorting algorithms (which we generally wait to learn until after the normal AP units are complete) are much more applicable to arrays than **ArrayLists**. Unit 10 also discusses recursive array algorithms briefly.

Array Searching Algorithms

Search algorithms are simply methods that provide us with different ways to find an item in an array if it exists. There are two common algorithms we typically use, either **Linear Search** for unsorted data or **Binary Search** for sorted data. You can find visualizations of these algorithms here[17].

Linear Search

Linear search is our typical sequential array iteration that many jump to for a simple search function. We simply iterate through the entire array from start to finish, checking to see if we find the index where the item is stored — otherwise, we usually return -1 or `null` (depending on the type).

```java
public static int linearSearch(int[] array, int x) {
  for (int i = 0; i < array.length; i++) {
    if (array[i] == x) return i;
  }
  return -1;
}
```

[17]https://www.cs.usfca.edu/~galles/visualization/Search.html

Binary Search

Binary search is a slightly more complicated search algorithm that excels when the list is already sorted. The idea is much like that of finding a page in a textbook — start by checking the middle of the list, compare the item to be found to see if we need to look on the right or left half of the list, then traverse down that half onward. Continuously check the middle item and traverse down the "hot" half until the item is found. While this algorithm is often implemented recursively (a function that calls itself; see Unit 10), we will see its iterative implementation for now.

```java
public static int binarySearch(int[] array, int x) {
  int low = 0;
  int high = array.length - 1;
  while (low <= high) {
    int mid = low + (high - low) / 2;
    if (array[mid] == x) return mid;
    if (array[mid] < x) low = mid + 1;
    else high = mid - 1;
  }
  return -1;
}
```

Array Sorting Algorithms

Sorting algorithms provide us with different approaches to sorting an array — of which there are MANY different possible choices. For now, we will learn strictly **comparison-based sorting** methods; i.e., comparing numbers and placing them in either *ascending* (least to greatest) or *descending* (greatest to least) order. Again, many of these algorithms can be performed recursively, but we will focus on their iterative implementation for now. You can find visualizations of these algorithms here[18] or here (alongside the running code)[19].

Let's look at a few of the most elementary sorting algorithms — bubble sort, insertion sort, and selection sort. These implementations are all in ascending order, but you can easily change them to descending order by changing the comparison operator inside the conditional statement from `>` to `<`.

If you want to truly understand when and why we use certain algorithms over others, you'll need to learn a topic called Asymptotic Analysis[20], or Big-O Notation[21], which provides us a means of mathematically analyzing how functions grow over time (and space) as the amount of data increases. This is typically the first true Computer Science topic that college majors learn in an infamous class known as ***Data Structures and Algorithms***.

[18] https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html
[19] https://visualgo.net/en/sorting
[20] https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation
[21] http://web.mit.edu/16.070/www/lecture/big_o.pdf

Bubble Sort

Bubble sort is the crudest array sorting algorithm; it is impractical for large datasets and generally slow even for small ones. However, its steps are so simple that it provides a good baseline for understanding the principles of designing sorting algorithms. The algorithm simply compares two adjacent values and swaps them until the entire list is sorted, *bubbling* them to the end of the list in sorted order.

The steps are as follows:

1. Compare the first two items in the list. If the first item is greater than the second item, swap them
2. Compare the second and third items in the list. If the second item is greater than the third item, swap them
3. Continue comparing and swapping pairs of items until you reach the end of the list
4. After reaching the end of the list, start again at the beginning and repeat the process until the list is sorted

```java
public static void bubbleSort(int[] array) {
  for (int i = 0; i < array.length - 1; i++)
    for (int j = 0; j < array.length - i - 1; j++)
      if (array[j] > array[j + 1]) {
        int temp = array[j];
        array[j] = array[j + 1];
        array[j + 1] = temp;
      }
}
```

Insertion Sort

Insertion sort is a simple but usable sorting algorithm that *inserts* values into their sorted location during each iteration, similar to how we might sort a deck of cards. We continuously place items into their correct position until the list is sorted.

The steps are as follows:

1. Start by considering the first element in the list as the sorted portion and the rest of the elements as the unsorted portion
2. Pick the second element in the list and compare it to the first element. If the second element is smaller than the first, swap them; otherwise, leave them as they are
3. Pick the third element in the list and compare it to the first and second elements. If it is smaller than either of them, insert it in the correct position relative to them; otherwise, leave it as it is
4. Continue this process, taking one element at a time and inserting it into the correct position in the sorted portion of the list until you reach the end of the list

```
public static void insertionSort(int[] array) {
  for (int step = 1; step < array.length; step++) {
    int key = array[step];
    int j = step - 1;
    while (j >= 0 && key < array[j]) {
      array[j + 1] = array[j];
      --j;
    }
    array[j + 1] = key;
  }
}
```

Selection Sort

Selection sort is another one of the simplest sorting algorithms that sorts values by repeatedly finding (*selecting*) the minimum (or maximum) element from the unsorted portion of the list and placing it at the beginning (or end) of the sorted portion of the list. We repeat this process until the list is sorted. It is very similar to bubble sort, but with only one exchange in each pass through the list.

The steps are as follows:

1. Start by considering the first element in the list as the sorted portion and the rest of the elements as the unsorted portion
2. Find the minimum element in the unsorted portion of the list and swap it with the first element in the list
3. Consider the first two elements in the list as the sorted portion and the remaining elements as the unsorted portion
4. Find the minimum element in the unsorted portion of the list and swap it with the second element in the list
5. Continue this process, taking one element at a time and placing it at the end of the sorted portion of the list, until you reach the end of the list

```
public static void selectionSort(int[] array) {
  for (int step = 0; step < array.length - 1; step++) {
    int minIndex = step;
    for (int i = step + 1; i < array.length; i++) {
      if (array[i] < array[minIndex]) {
        minIndex = i;
      }
    }
    int temp = array[step];
    array[step] = array[minIndex];
    array[minIndex] = temp;
  }
}
```

Vector Operations

Below is a library of matrix methods based on operations found in calculus and linear algebra, which are extremely useful for statistics, machine learning, and deep learning algorithms. While these are not on the AP exam (at least explicitly), they can be very helpful for understanding the purpose of arrays in various contexts (i.e., physics, image processing, game design, etc.) especially coming from a mathematical background.

```java
/* Vector Library by Daniel Szelogowski, 2022 */
public class VectorLib {

  public static double[] add(double[] a, double[] b) {
    double[] c = new double[a.length];
    for (int i = 0; i < a.length; i++)
      c[i] = a[i] + b[i];
    return c;
  }

  public static double[] sub(double[] a, double[] b) {
    double[] c = new double[a.length];
    for (int i = 0; i < a.length; i++)
      c[i] = a[i] - b[i];
    return c;
  }

  public static double[] scalarMult(double[] a, double b) {
    double[] c = new double[a.length];
    for (int i = 0; i < a.length; i++)
      c[i] = a[i] * b;
    return c;
  }

  public static double dot(double[] a, double[] b) {
    double c = 0;
    for (int i = 0; i < a.length; i++)
      c += a[i] * b[i];
    return c;
  }

  public static double[] cross(double[] a, double[] b) {
    double[] c = new double[a.length];
    c[0] = a[1] * b[2] - a[2] * b[1];
    c[1] = a[2] * b[0] - a[0] * b[2];
    c[2] = a[0] * b[1] - a[1] * b[0];
    return c;
  }
  public static double[] normalize(double[] a) {
    double[] c = new double[a.length];
    double mag = 0;
    for (int i = 0; i < a.length; i++)
      mag += a[i] * a[i];
    mag = (double) Math.sqrt(mag);
    for (int i = 0; i < a.length; i++)
      c[i] = a[i] / mag;
    return c;
  }

  public static double magnitude(double[] a) {
    double mag = 0;
    for (int i = 0; i < a.length; i++)
      mag += a[i] * a[i];
    return (double) Math.sqrt(mag);
  }

  public static double angle(double[] a, double[] b) {
    double dot = dot(a, b);
    double magA = magnitude(a);
    double magB = magnitude(b);
    return (double) Math.acos(dot / (magA * magB));
  }

  public static double[] convolve(double[] a, double[] b) {
    double[] c = new double[a.length + b.length - 1];
    for (int i = 0; i < c.length; i++) {
      for (int j = 0; j < a.length; j++) {
        if (i - j >= 0 && i - j < b.length)
          c[i] += a[j] * b[i - j];
      }
    }
    return c;
  }

  public static int argmax(double[] a) {
    int max = 0;
    for (int i = 1; i < a.length; i++)
      if (a[i] > a[max]) max = i;
    return max;
  }
}
```

Unit 7 - ArrayList

Also known as Dynamic Arrays (or erroneously, *vector* in C++), the **ArrayList** class provides us with a simplified *abstraction* of a normal array where the list has no specific capacity. The class Instead, ArrayLists automatically resize *"under the hood"* whenever they reach capacity (usually by some rules such as doubling when the array is full or halving when the array is less than 25% full) by making a temporary array of the new capacity, copying all the items over from the class' private array, and reassigning the private array to the temporary one (though some implementations simply use a **Doubly-Linked List** structure). This also allows us to interface with the list much easier using a large number of provided methods from the ArrayList class, such as using `add(item)` and `get(index)` methods rather than the index operator `[]`, searching for items using `contains(item)` or `indexOf(item)`, deleting items and (automagically) shifting the rest of the items down one index using `remove(item)`, and many others.

Just like with the **Scanner** class, we need to first import the **ArrayList** class from the **java.util** library, either using the entire library with `import java.util.*;` or just the ArrayList class with `import java.util.ArrayList;`.

The syntax for an ArrayList is as follows:

```java
// First, import from java.util;
import java.util.ArrayList;
// ...
ArrayList<DATATYPE> varname = new ArrayList<DATATYPE>();
```

We can also instantiate an ArrayList as an instance of the **List** interface, which is more common for code compatibility:

```java
List<DATATYPE> listname = new ArrayList<DATATYPE>();
```

Like arrays, ArrayLists can store any singular type of data or class. However, the one disadvantage with this class is that **to use an ArrayList with a primitive type, you MUST use the wrapper class for that type**; i.e., **Integer** for `int`, **Double** for `double`, **Character** for `char`, etc. When we add and retrieve these primitive values back from the list, Java will automatically **autobox** and **unbox** the objects to their primitive form. For everything else, like **String**, the name of the class is sufficient.

Like a normal array, we can easily populate an ArrayList using a **for** loop and the `add()` method (however, we need to specify the end number of elements, since an ArrayList's size is unrelated to its current capacity):

```
List<Integer> nums = new ArrayList<Integer>();
for (int i = 0; i < 100; i++) {
  nums.add(i);
}
```

ArrayLists provide us with a great number of built-in methods[22] for working with lists of data, though we only really need to know the following:

- `x.size()`: return the number of values/objects stored in the list
- `x.add(obj)`: appends an object to the end of the list
- `x.get(index)`: returns the item stored at the specified index.
- `x.set(index, obj)`: replace the item at the specified index with the provided object
- `x.remove(index)`: remove the item at the specified index and shift all proceeding elements down 1 position in the list

Other useful methods include `contains(object)`, `indexOf(object)`, `toArray()`, and `addAll()`. **Take careful note that ArrayLists do not support the array `[]` operators — you have to use `.get(index)` instead.**

Let's look at an example of these various method calls:

```
import java.util.ArrayList;

public class ArrayListExamples {
  public static void main(String[] args) {
    // Create an ArrayList of Strings
    List<String> list = new ArrayList<String>();

    // Add elements to the list
    list.add("apple");
    list.add("banana");
    list.add("cherry");
    list.add("blueberry");
    list.add("grape");

    // Print the size of the list
    System.out.println("Size of list: " + list.size());

    // Get the element at index 2
    String element = list.get(2);
    System.out.println("Element at index 2: " + element);

    // Set the element at index 3 to "pineapple"
    list.set(3, "pineapple");
    System.out.println("Updated list: " + list);
```

[22]https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

```
    // Remove the element at index 1
    list.remove(1);
    System.out.println("Updated list: " + list);

    // Find the index of the "cherry" element
    int index = list.indexOf("cherry");
    System.out.println("Index of cherry: " + index);
  }
}
```

The **ArrayList** class is a type of ***generic class*** (sometimes ***template class***, such as in C++), meaning it works with any possible type of data. Generic typing is not on the AP subset, but it is extremely useful and practical (see the Java documentation on Generic Types[23]). For reference, the class signature might look something like `public class ArrayList<T> { ... }` and contain a private array of the generic type `T` like `private T[] myArray`.

Traversing ArrayLists

At its simplest, traversing an ArrayList is no different than traversing a normal array. Either start a loop at some index and increment/decrement to some end index, then *get* the item at that index to use it, or simply use a **for-each** loop if the index does not matter.

For example, to retrieve items, we could use either a normal **for** loop with the `get()` method or a **for-each** loop:

```
// Print out all the items in the "nums" ArrayList
for (int i = 0; i < nums.size(); i++)
  System.out.print(nums.get(i) + " ";
System.out.println();

for (int n : nums)
  System.out.print(n + " ");
System.out.println();
```

ArrayList Algorithms

ArrayLists often utilize the same algorithms that we learn with *arrays* in Unit 6 — statistical or property analysis (max, min, average, duplicates, etc.), linear/binary search, and all of the many sorting algorithms still apply. However, it is much less common to see examples of searching and sorting algorithms being applied to ArrayLists as opposed to arrays, since we often perform those algorithms on intentionally-capped lists of data and/or learn them using arrays that are entirely full of data. Regardless, if you can write an algorithm for an array, it is nearly identical to writing the same algorithm for an ArrayList and vice versa.

[23]https://docs.oracle.com/javase/tutorial/java/generics/types.html

Unit 8 - 2D Array

Often called a Matrix from its mathematical derivative, a **2D Array** is an array containing two **dimensions** or *features* of data (here, the term "dimension" is borrowed from dimensions in a *Hilbert Space* from Calculus, i.e., the "size" of a set of vectors, rather than a physical *Euclidian Space* dimension). In simpler terms, it is an array of arrays, which are typically all the same length. Here, we think of each array as being a **row** in the matrix, and each *feature* (position) refers to the **columns** in the matrix. Their mathematical equivalent, matrices, are denoted using italicized uppercase letters, such as A.

$$\begin{bmatrix} 1 & 2 & 3 \\ a & b & c \end{bmatrix}$$

We can declare a 2D Array using the same syntax as a normal array (with `new` keyword), but by attaching an additional set of array braces `[size]`, like so:

```
DATATYPE[][] multiDimArr = new DATATYPE[numRows][numCols];
// However, the number of columns is technically optional,
// since we may have jagged arrays
```

Just like with any other data standard data structure, this works with any data type:

```
// Create a matrix with 5 rows, 6 columns
// i.e., 5 internal array of length 6
int[][] matrix = new int[5][6];
double[][] matrix2 = new double[5][6];
```

However, we can also use the *explicit* notation, either by building each row array individually and making an array of row arrays, or by defining the entire matrix together:

```
// Matrix-style (very common)
int[][] mat1 = {
  {0, 1, 2},
  {3, 4, 5},
  {6, 7, 8}
};
```

```
// Array of arrays (more confusing to read)
int[] row1 = {0, 1, 2};
int[] row2 = {3, 4, 5};
int[] row3 = {6, 7, 8};
int[][] mat2 = {row1, row2, row3};
```

Then, we can access items based on their `[row][column]` index, much like a 1D array: `int rowOneColTwo = mat1[1][2];` Likewise, we can apply many of the same algorithms from 1D arrays to 2D arrays, but there are many mathematical operations specific to matrices that are also useful, such as computing things like the determinant, eigenvalues, eigenvectors, and many more (check out finite mathematics and linear algebra!).

Multidimensional and Jagged Arrays

2D Arrays are often classified into two different categories:

- **Multidimensional Array (Matrix)**: an $n \times m$ matrix (i.e., n rows and m columns) where all rows have the exact same length; typically declared using the explicit matrix notation or `new DATATYPE[numRows][numCols]`
- **Jagged Array**: an array of arrays, where the inner arrays may possibly vary in length; typically declared using the explicit jagged array notation or `new DATATYPE[numRows][]`

Let's compare the two:

```
// Matrix Notation
int[][] mat1 = new int[2][3];
int[][] mat2 = {
  {0, 1, 2},
  {3, 4, 5}
};

// Jagged Array Notation
int[][] ja1 = new int[3][];
ja1[0] = new int[]{0, 1};
ja1[1] = new int[]{2, 3, 4};
ja1[2] = new int[]{5};

int[][] ja2 = {
  {0, 1},
  {2, 3, 4},
  {5}
};
```

Nested Iteration for Matrix Traversal

To both populate and traverse matrices and jagged arrays, we must use a ***for loop inside another for loop*** for simplicity. For a traditional matrix (multidimensional array) where the number of columns are the same for all rows, we can loop using `matrixName.length` for the rows and `matrixName[0].length` for the columns; otherwise, we might need to use our row counter to find the length of the current row (i.e., `multiDimArrName[row].length` if we have a jagged array). We typically loop through 2D arrays in a style known as ***Row-Major Order***, where we start at the first item of the first row, traverse to the end of the row, then start the next row at its first item (i.e., nest the *column* loop inside the *row* loop). If we nested our *row* loop inside the *column* loop instead, this would become ***Column-Major Order***.

For example:

```
// Create and populate a matrix of ints
int[][] powers = new int[5][3];
for (int row = 0; row < powers.length; row++) {
  for (int col = 0; col < powers[0].length; col++) {
    powers[row][col] = (int)Math.pow(col + 1, row + 1);
  }
}

// Print out the matrix in Row-Major Order
for (int r = 0; r < powers.length; r++) {
  for (int c = 0; c < powers[0].length; c++) {
    System.out.print(powers[r][c] + " ");
  }
  System.out.println();
}

// Print out the matrix in Column-Major Order
for (int c = 0; c < powers[0].length; c++) {
  for (int r = 0; r < powers.length; r++) {
    System.out.print(powers[r][c] + " ");
  }
  System.out.println();
}
```

However, with a jagged array, we need to use `jaggedArr[row].length` when looping through the columns. Or, more commonly, we can use a **for-each** (or *Enhanced for*) loop to grab each inner array and loop through each of its values:

```
int[][] jagArr = {
  {0, 1, 2},
  {3, 4, 5, 6, 7},
  {8, 9},
  {10}
};
```

```
// Option 1: double-for
for (int r = 0; r < jagArr.length; r++) {
  for (int c = 0; c < jagArr[r].length; c++) {
    System.out.print(jagArr[r][c] + " ");
  }
  System.out.println();
}

// Option 2: double-for-each
for (int[] row : jagArr) {
  for (int x : row) {
    System.out.print(x + " ");
  }
  System.out.println();
}
```

nD Arrays/Tensors (3D and Beyond)

Technically, we can extend the multidimensional array syntax to any number of dimensions, i.e., an nD array, often referred to as a **Tensor** (which can technically be any number of dimensions but is often used when $n > 2$). For data scientists and machine learning engineers, tensors are one of the most important data structures that exists — especially because of its use in deep learning/neural networks. Simply add an additional set of `[]` to your array initialization, or painstakingly define a jagged array of arrays explicitly:

```
DATATYPE[][][] tensor = new DATATYPE[numRows][numCols][numLayers];

// Or...
int[][][] jaggedTensor = {
  {
    {0, 1},
    {2, 3, 4},
    {5}
  },
  {
    {6, 7},
    {8}
  },
  {
    {9, 10}
  }
};

// Or, a 5D Array
DATATYPE[][][][][] hugeTensor = new DATATYPE[x][y][z][w][alpha];
```

Likewise, to traverse a tensor (and/or populate it), add on an additional loop for every dimension:

```java
// Create a random 5x5x5 tensor
int[][][] tens = new int[5][5][5];
for (int x = 0; x < tens.length; x++) {
  for (int y = 0; y < tens[x].length; y++) {
    for (int z = 0; z < tens[x][y].length; z++) {
      tens[x][y][z] = (int)(Math.random() * 9 + 1);
    }
  }
}

// Option 1: triple-for
for (int x = 0; x < tens.length; x++) {
  for (int y = 0; y < tens[x].length; y++) {
    for (int z = 0; z < tens[x][y].length; z++) {
      System.out.print(tens[x][y][z] + " ");
    }
    System.out.print("\t");
  }
  System.out.println();
}

// Option 2: triple-for-each
for (int[][] mat : tens) {
  for (int[] vec : mat) {
    for (int scalar : vec) {
      System.out.print(scalar + " ");
    }
    System.out.print("\t");
  }
  System.out.println();
}
```

Matrix Operations

Below is a library of matrix methods based on operations found in calculus and linear algebra, which are extremely useful for statistics, machine learning, and deep learning algorithms. Like with vector operations, while these are not on the AP exam, they can be very helpful for understanding the purpose of arrays in various contexts (i.e., physics, image processing, quantum computing, artificial intelligence, etc.) especially coming from a mathematical background.

```java
/* Matrix Library by Daniel Szelogowski, 2022 */
public class MatrixLib {

  public static double[][] add(double[][] a, double[][] b) {
    double[][] c = new double[a.length][a[0].length];
    for (int i = 0; i < a.length; i++)
      for (int j = 0; j < a[0].length; j++)
        c[i][j] = a[i][j] + b[i][j];
    return c;
  }

  public static double[][] sub(double[][] a, double[][] b) {
    double[][] c = new double[a.length][a[0].length];
    for (int i = 0; i < a.length; i++)
      for (int j = 0; j < a[0].length; j++)
        c[i][j] = a[i][j] - b[i][j];
    return c;
  }

  public static double[][] scalarMult(double[][] a, double b) {
    double[][] c = new double[a.length][a[0].length];
    for (int i = 0; i < a.length; i++)
      for (int j = 0; j < a[0].length; j++)
        c[i][j] = a[i][j] * b;
    return c;
  }

  public static double[][] mult(double[][] a, double[][] b) {
    double[][] c = new double[a.length][b[0].length];
    for (int i = 0; i < a.length; i++)
      for (int j = 0; j < b[0].length; j++)
        for (int k = 0; k < a[0].length; k++)
          c[i][j] += a[i][k] * b[k][j];
    return c;
  }

  public static double[][] transpose(double[][] a) {
    double[][] c = new double[a[0].length][a.length];
    for (int i = 0; i < a.length; i++)
      for (int j = 0; j < a[0].length; j++)
        c[j][i] = a[i][j];
    return c;
  }
```

```
public static double[][] id(double[][] a) {
  double[][] c = new double[a.length][a[0].length];
  for (int i = 0; i < a.length; i++)
    for (int j = 0; j < a[0].length; j++)
      if (i == j) c[i][j] = 1;
      else c[i][j] = 0;
  return c;
}

public static double determinant(double[][] a) {
  double det = 0;
  if (a.length == 1) return a[0][0];
  for (int i = 0; i < a.length; i++) {
    double[][] sub = new double[a.length - 1][a.length - 1];
    for (int j = 1; j < a.length; j++)
      for (int k = 0; k < a.length; k++)
        if (k < i) sub[j - 1][k] = a[j][k];
        else if (k > i)
          sub[j - 1][k - 1] = a[j][k];
    det += a[0][i] * Math.pow(-1, i) * determinant(sub);
  }
  return det;
}

public static double[][] inverse(double[][] a) {
  double[][] c = new double[a.length][a[0].length];
  double det = determinant(a);
  if (det == 0) return null;
  for (int i = 0; i < a.length; i++)
    for (int j = 0; j < a[0].length; j++) {
      double[][] sub = new double[a.length - 1][a.length - 1];
      for (int k = 0; k < a.length; k++)
        for (int l = 0; l < a.length; l++)
          if (k < i && l < j)
            sub[k][l] = a[k][l];
          else if (k < i && l > j)
            sub[k][l - 1] = a[k][l];
          else if (k > i && l < j)
            sub[k - 1][l] = a[k][l];
          else if (k > i && l > j)
            sub[k - 1][l - 1] = a[k][l];
      c[i][j] = Math.pow(-1, i + j) * determinant(sub) / det;
    }
  return transpose(c);
}
```

```
  public static double[][] convolve(double[][] a, double[][] b) {
    double[][] c = new double[a.length][a[0].length];
    for (int i = 0; i < a.length; i++)
      for (int j = 0; j < a[0].length; j++)
        for (int k = 0; k < b.length; k++)
          for (int l = 0; l < b[0].length; l++)
            if (i + k < a.length && j + l < a[0].length)
              c[i + k][j + l] += a[i][j] * b[k][l];
    return c;
  }

  public static double[][] eigenvectors(double[][] a) {
    double[][] c = new double[a.length][a[0].length];
    double[][] id = id(a);
    double[][] sub = sub(a, id);
    double[][] inverse = inverse(sub);
    for (int i = 0; i < a.length; i++)
      for (int j = 0; j < a[0].length; j++)
        c[i][j] = inverse[i][j] / inverse[i][j];
    return c;
  }

  public static double[] eigenvalues(double[][] a) {
    double[] c = new double[a.length];
    double[][] id = id(a);
    double[][] sub = sub(a, id);
    double[][] inverse = inverse(sub);
    for (int i = 0; i < a.length; i++)
      c[i] = 1 / inverse[i][i];
    return c;
  }
}
```

Unit 9 - Inheritance

Inheritance is a fundamental concept in **Object-Oriented Programming (OOP)** languages, such as Java, C++, and Python. It allows one class (also known as a **"parent"** or *"super"* class) to pass on its attributes and behaviors to another class (known as a **"child"** or *"sub"* class). This allows the child class to reuse the functionality of the parent class while also adding its own unique features, as well as more efficient and organized code because it allows for code reuse and reduces duplication. It also provides a better organization of related classes and makes it easier to create and maintain large programs.

Put simply, a **Dog** class could inherit from a more general **Animal** class. The Animal class might have a `move()` method that defines the general behavior for moving, while the Dog class could override this method to include specific behavior for how a dog moves. For example:

```java
public class Animal {
  protected String name;
  protected int numLegs;

  public Animal(String name, int numLegs) {
    this.name = name;
    this.numLegs = numLegs;
  }

  public void move() {
    System.out.println("Animal is moving");
  }
}

public class Dog extends Animal {
  public Dog(String name) {
    // Use the super keyword to call the constructor of the parent class
    super(name, 4);
  }
  // Override the move() method from the parent class
  @Override
  public void move() {
    System.out.println(name + " is running");
  }
}

public class InheritanceTest {
  public static void main(String[] args) {
    Animal animal = new Animal("animal", 4);
    animal.move();  // prints "Animal is moving"
    Dog dog = new Dog("dog");
    dog.move();  // prints "dog is running"
  }
}
```

Note that the `protected` keyword is NOT on the AP subset, but it is useful; it simply implies that ***the method/variable is private to everyone except any child classes***.

Creating Superclasses and Subclasses

A **Superclass** is a parent class that defines common attributes and methods that are inherited by subclasses. A **Subclass**, on the other hand is a child class that inherits the attributes and methods of the superclass and can also have its own unique attributes and methods.

Let's look at an example; first, let's define a superclass called **Vehicle** that has two attributes: `make` and `model` and one method called `startEngine`:

```
public class Vehicle {
  String make;
  String model;

  public void startEngine() {
    System.out.println("Starting engine...");
  }
}
```

Next, let's create a subclass called **Car** that *inherits* the attributes and method of the Vehicle superclass and also has its own unique attribute called `numDoors` and method called `honkHorn`:

```
public class Car extends Vehicle {
  int numDoors;

  public void honkHorn() {
    System.out.println("Honk honk!");
  }
}
```

To use the **Car** class, we can create an object of type `Car` and call the inherited and unique methods as follows:

```
Car myCar = new Car();
myCar.make = "Toyota";
myCar.model = "Camry";
myCar.numDoors = 4;
myCar.startEngine();  // Output: "Starting engine..."
myCar.honkHorn();     // Output: "Honk honk!"
```

To summarize, the **Car** class is a *subclass* of the **Vehicle** *superclass* because it inherits the attributes and methods of the Vehicle class but also has its own unique attributes and methods.

Writing Constructors for Subclasses

Recall that a constructor is a special method that is used to initialize an object's private data. When creating a subclass, it is often necessary to create a constructor for the subclass. This is done by defining a constructor method in the subclass that has the same name as the subclass. The constructor can then be used to initialize the properties ***specific to that subclass***.

For example, consider a simple class hierarchy with a base class called **Shape** and two subclasses called **Circle** and **Rectangle**:

```
public class Shape {
  // Base class for shapes
}

public class Circle extends Shape {
  private int radius;

  public Circle(int radius) {
    this.radius = radius;
  }
}

public class Rectangle extends Shape {
  private int width;
  private int height;

  public Rectangle(int width, int height) {
    this.width = width;
    this.height = height;
  }
}
```

However, another extremely important component of writing subclasses is the `super` keyword, which allows us to pass values into the superclass' constructor.

Overriding Methods

Method **Overriding** is a feature in Object-Oriented Programming that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes. This allows for subclasses to have their own unique behavior, while still retaining some of the basic behavior of the parent class.

Let's look at an **Animal** class with a `makeNoise()` method that simply outputs the string `"Making noise"`. If you have a **Dog** class that *extends* **Animal**, you could **override** the `makeNoise()` method in the Dog class to output the string `"Woof!"` instead. This allows the Dog class to have its own unique behavior, while still retaining the basic behavior of the Animal class.

In order to override a method in Java, the following conditions must be met:

- The method in the subclass must have the same name and parameter list as the method in the superclass
- The method in the subclass must have the same return type (or a subtype) as the method in the superclass
- The method in the subclass SHOULD be marked with the `@Override` annotation. This tells the compiler that the method is intended to override a method in the superclass

Consider the following example:

```java
class Animal {
  public void makeNoise() {
    System.out.println("Making noise");
  }
}

class Dog extends Animal {
  @Override
  public void makeNoise() {
    System.out.println("Woof!");
  }
}
```

Here, the **Dog** class overrides the `makeNoise()` method from the **Animal** class and provides its own implementation. When `makeNoise()` is called on an instance of the Dog class, it will output the string `"Woof!"` instead of `"Making noise"`.

Overloading Methods

It is very important to note that while some people use the terms "Overloading" and "Overriding" synonymously, they are two VERY different concepts in programming. Method

Overloading refers to the ability of a class to ***have multiple methods with the same name, but with different parameters***. This allows for the creation of methods that can perform similar tasks, but with varying numbers and/or types of arguments.
For example:

```
public class Calculator {
  // Method that takes two integers and returns their sum
  public int add(int a, int b) {
    return a + b;
  }

  // Method that takes three integers and returns their sum
  public int add(int a, int b, int c) {
    return a + b + c;
  }

  // Method that takes a list of integers and returns their sum
  public int add(List<Integer> numbers) {
    int sum = 0;
    for (int num : numbers)
      sum += num;
    return sum;
  }
}
```

Here, the **Calculator** class has three methods named `add()`, but each method takes a different number and type of arguments.

Again, method overloading is different from method overriding, which refers to the ability of a subclass to provide a different implementation of a method that is already defined in the superclass. In method overriding, the method signature (i.e., the name and the number and type of arguments) must be the same in both the superclass and the subclass.

For example:

```
public class Shape {
  // Method that calculates the area of the shape
  public double getArea() {
    // Method implementation goes here
  }
}

public class Circle extends Shape {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }
  // Overridden method that calculates the area of a circle
  @Override
  public double getArea() {
    return Math.PI * radius * radius;
  }
}
```

We technically have two implementations of `getArea()`, but since they share the same arguments (none, in this case) and it is inherited, we *Override* the original implementation from the parent class.

The *super* Keyword

The `super` keyword acts as a reference variable referring to the immediate parent class object. It is mainly used to access the members (either methods or variables) of a parent class that has been hidden by a child class.

For example, if class **A** is the parent of class **B**, and both classes have a method named `printMessage()`, the `printMessage()` method in class B can call the `printMessage()` method of class A using the `super` keyword:

```
class A {
   public void printMessage() {
       System.out.println("This is a message from class A");
   }
}

class B extends A {
   public void printMessage() {
       // Call the printMessage() method of class A
       super.printMessage();
       System.out.println("This is a message from class B");
   }
}
```

Here the `printMessage()` method of class **B** calls the method of the same name from class A using the `super` keyword and then prints its own message. This allows the `printMessage()` method of class B to both inherit and extend the behavior of the original method from class A.

Super Constructors

A **Super Constructor** is a constructor that is used to create an instance of a *subclass* while setting up the inherited class variables from the *superclass*. It is called using the `super` keyword just like any other inherited member accesses. For example:

```java
class MyAnimal {
  private String name;
  private int age;

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

class Cat extends MyAnimal {
  private String color;

  public Cat(String name, int age, String color) {
    super(name, age);
    this.color = color;
  }
}
```

The **MyAnimal** class has a constructor that takes two arguments, `name` and `age`, and uses them to initialize the corresponding instance variables. The **Cat** class extends MyAnimal, so it inherits the name and age instance variables. In the Cat class, we have added a new instance variable, `color`, which is not present in the MyAnimal class. To initialize the `name` and `age` variables, which are inherited from MyAnimal, we use the `super` keyword to call the *superclass constructor*. ***The super keyword must be used as the first statement in the subclass constructor, and it must pass the values for the name and age variables as arguments***. This ensures that `name` and `age` are properly initialized before the Cat constructor proceeds to initialize the `color` variable.

The *instanceof* Operator

In Java, the `instanceof` operator is a conditional operator used to determine if an object is an instance of a particular class or if it implements a particular interface. It returns a boolean value indicating whether or not the object is an instance of the specified class or interface.

Consider the following example:

```java
// Create an object of the String class
String str = "Hello, World!";

// Check if str is an instance of the String class
if (str instanceof String) {
  System.out.println("str is an instance of String");
} else {
  System.out.println("str is not an instance of String");
}
```

Here, `instanceof` is used to check if the `str` object is an instance of the **String** class. Since this is true, the code will print *"str is an instance of String"* to the console.

You can also use the `instanceof` operator to check if an object implements a particular interface. For example:

```
// Create an object of the ArrayList class
List<String> list = new ArrayList<>();

// Check if list implements the List interface
if (list instanceof List) {
  System.out.println("list implements the List interface");
} else {
  System.out.println("list does not implement the List interface");
}
```

Here, we check if the list object implements the List interface. Since the **ArrayList** class implements the **List** interface, the code will print *"list implements the List interface"*. Also, note that ***when making an instance of a class that implements an interface or inherits from a superclass, we often declare the inherited or implemented class on the left-hand side and specify the object type with the `new` keyword***. For example:

```
class Animal { }
class Dog extends Animal { }
class Cat extends Animal { }

public class CheckSuperInstance {
  public static void main(String[] args) {
    // Create a Dog object
    Animal animal = new Dog();

    // Check if animal is an instance of the Dog class
    if (animal instanceof Dog)
      System.out.println("animal is an instance of Dog");
    else
      System.out.println("animal is not an instance of Dog");

    // Check if animal is an instance of the Cat class
    if (animal instanceof Cat)
      System.out.println("animal is an instance of Cat");
    else
      System.out.println("animal is not an instance of Cat");
  }
}
```

Inheritance Hierarchies - The Theory of Object-Oriented Programming

In Object-Oriented Programming, an **Inheritance Hierarchy** refers to the way that classes inherit attributes and behaviors from parent classes. This allows for a hierarchical relationship between classes, where child classes inherit the attributes and behaviors of their parent classes. Inheritance hierarchies can also have multiple levels of inheritance, where a child class inherits from a parent class, which itself inherits from a grandparent class, and so on, enabling a flexible and reusable code structure where common attributes and behaviors can be defined in parent classes and then inherited by child classes.

Encapsulation

The idea of **Encapsulation** is one of the most fundamental concepts in Object-Oriented Programming. It is a technique for ensuring that the internal workings of a class are hidden from other classes and external users of an object. In other words, it is the process of enclosing all of the data and behavior of an object within a single, self-contained unit.

In Java (and most C-based programming languages like C++), encapsulation is implemented through the use of access modifiers, such as `private`, `protected`, and `public`. These modifiers determine the visibility and accessibility of the members of a class, such as its fields and methods. For example, a **private field** can only be accessed by the methods of the same class, while a **public field** can be accessed by any other class. This allows the developer to control how the internal data of an object is accessed and modified and ensures that the object's internal state remains consistent and correct.

Encapsulation has several benefits. It promotes modularity and code reusability, as it allows developers to create self-contained objects that can be easily reused in other parts of the program. It also increases code maintainability, as it allows us to make changes to the internal workings of an object without affecting other parts of the program. Finally, it enhances security, as it prevents external code from directly accessing or modifying the internal data of an object, which can help protect against malicious attacks.

Polymorphism

Another important core OOP concept is **Polymorphism** (literally *"many forms"*), which refers to the ability of a variable, object, or function to take on multiple forms or behaviors. In the context of Java, polymorphism refers to the ability of an object to behave differently based on the current context in which it is used. This is achieved through the use of *inheritance*, *interfaces*, and *overridden* methods, and is extremely valuable in writing flexible and reusable code.

Consider inheritance, which allows a child class to inherit the methods and properties of a parent class. This means that a child class can have its own unique behavior and attributes, while also being able to use the methods and properties of the parent class. For example, a parent class called **Animal** could have a `speak()` method that describes the general behavior of moving, while a child class called **Cat** could inherit the `speak()` method from the Animal class and also have its own unique `purr()` method.

Another way that polymorphism is achieved is through the use of **Interfaces**. An interface defines a set of methods that a class must implement, but it does not provide any implementation for those methods. This means that a class can implement multiple interfaces, each of which defines a different set of methods, and the class can provide its own unique implementation for each of those methods. This allows a single class to have multiple behaviors depending on which interface is being used to access its methods.

Finally, polymorphism can also be achieved through the use of overridden methods, meaning that a child class can provide its own implementation for a method that it has inherited from a parent class, which allows the child class to have its own unique behavior for that method. This is often used to provide more specific or specialized behavior for a particular child class, while still being able to use the more general behavior defined by the parent class.

The *Object* Superclass

In Java (and many other OOP languages), the **Object** class is the superclass of all other classes. This means that ***every class in Java inherits the methods and fields defined in the Object class***. The class provides a set of methods that are available to all objects regardless of their specific type, including `equals()`, which allows you to compare two objects to see if they are equal, and `toString()`, which returns a String representation of the object. The **default value** of an object reference (i.e., a variable that refers to an object) is `null`, which means that it doesn't refer to any object.

To override the `toString()` method, you would declare it as a method in your class that returns a string representation of the object. This method would have the following signature:

```
public String toString() {
  // ...
}
```

Likewise, to override the `equals()` method, you would declare it as a method in your class that takes another object as a parameter and returns a **boolean** indicating whether the two objects are equal. This method would have the following signature:

```
public boolean equals(Object other) {
  // ...
}
```

When overriding these methods, it is important to follow the general contract specified in the Object class. For example, the `equals()` method should return `true` if and only if the two objects are equal, and `toString()` should return a string representation of the object that is consistent with its `equals()` method.

Consider the following class definition that **explicitly** extends the Object class:

```
public class SomeClass extends Object {
  // ...
}
```

Since SomeClass extends the Object class, it automatically has access to the methods defined in the Object class, including `equals()` and `toString()`.

Consider the following example:

```
public class MyClass extends Object {
  private int x;
  private int y;

  public MyClass(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public String toString() {
    return "MyClass[x=" + x + ",y=" + y + "]";
  }
}
```

Here, the `toString()` method returns a string representation of a MyClass object that includes the values of the `x` and `y` fields.

The **Object** class and its methods are an important part of the object-oriented nature of Java. They provide a foundation for working with objects in the language and allow you to create classes that can be used in a consistent and predictable way.

Interfaces and Abstract Classes

While not currently in the AP subset, two important concepts that extend the concept of inheritance are **Interfaces** and **Abstract Classes**. In Object-Oriented Programming, an **Interface** is a blueprint for a class that specifies the behavior that a class must implement. It defines a set of methods that a class MUST implement, but it does not provide any implementation for those methods. Thus, when a class *implements* an interface, it must implement all of the methods that the interface defined.

An **Abstract Class** is a class that cannot be instantiated but can be extended by other classes. Abstract classes can contain both concrete (normal) and *abstract* methods (marked with the `abstract` keyword), which are methods that lack implementation. Unlike interfaces, abstract classes can provide *some* implementations for their methods.

In Java, a class can implement multiple interfaces, but can only extend a single abstract class. This is because Java (unfortunately) does not support multiple inheritance, which is the ability of a class to inherit from multiple classes. However, ***it is very important to***

note that you cannot make an instance of an abstract class or an interface, since abstract classes are meant to be a superclass from which other classes can inherit, and interfaces provide no functionality. Since abstract classes can have abstract and non-abstract methods, the subclasses MUST provide implementations for the abstract methods.

Interfaces

In Java, an **Interface** is a reference type that is similar to a **Class**. Put simply, it is a collection of abstract methods and constant values. ***A class implements an interface by providing implementations for all of the interface's methods***.

Consider the following example:

```java
public interface Shape {
  double getArea();
  double getPerimeter();
}
```

A class that implements the Shape interface (literally, using the `implements` keyword) would need to provide implementations for the `getArea()` and `getPerimeter()` methods — however, they must also be marked as overridden methods using `@Override`.

For example:

```java
public class Circle implements Shape {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  @Override
  public double getArea() {
    return Math.PI * radius * radius;
  }

  @Override
  public double getPerimeter() {
    return 2 * Math.PI * radius;
  }
}
```

Here, the **Circle** class implements the **Shape** interface by providing implementations for the methods that the interface defines.

Another advantage of using interfaces is that a class can implement *multiple interfaces*. For example, a **Square** class that implements the methods of the **Shape** interface, as well as an interface named **Printable**:

```java
public interface Printable {
  void print();
}

public class Square implements Shape, Printable {
  private double sideLength;

  public Square(double sideLength) {
    this.sideLength = sideLength;
  }

  @Override
  public double getArea() {
    return sideLength * sideLength;
  }

  @Override
  public double getPerimeter() {
    return 4 * sideLength;
  }

  @Override
  public void print() {
    System.out.println("Side length: " + sideLength);
  }
}
```

Abstract Classes and Methods

In OOP, an **Abstract Class** is a class that contains one or more abstract methods and possibly some concrete methods. An **Abstract Method** is a method that has a declaration but does not have an implementation. This means that the method does not have a body – it just specifies the method's *signature*, including the name of the method, the return type, and the list of parameters.

For example:

```java
public abstract class Animal {
  // Abstract method
  public abstract void makeSound();

  // Concrete method
  public void eat() {
    System.out.println("Eating...");
  }
}
```

In this example, the **Animal** class contains one abstract method called `makeSound()`, as well as one **Concrete (normal) Method** called `eat()`. The `makeSound()` is abstract because it does not have an implementation — it simply specifies that the method will take no parameters and return no value, whereas the `eat()` method is concrete since it has a complete implementation, including a method body.

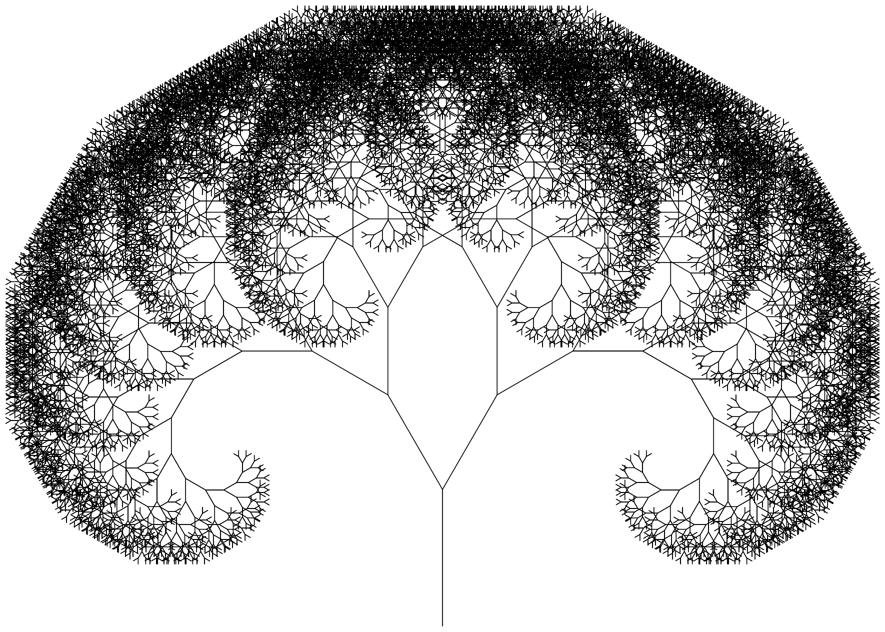
To use an abstract class, you need to create a subclass that *extends* (like inheritance) the abstract class and provides implementations for all of the abstract methods. For example:

```java
public class Dog extends Animal {
  // Implement the abstract method
  @Override
  public void makeSound() {
    System.out.println("Woof!");
  }
}
```

Here, the **Dog** class extends the **Animal** class and provides an implementation for the `makeSound()` method, meaning we can make instances of dogs and `makeSound()` will have the desired behavior when it is called.

Unit 10 - Recursion

The concept of **Recursion** can be seen in countless real-life examples; simply put, it is a function that calls itself. Many everyday concepts are *recursive* — think of Russian nesting dolls (a doll inside a doll inside a doll...), Ouroboros (the serpent eating its own tail), sourdough starter (which can be kept forever so long as you continue to add more over time), two mirrors facing each other, and in particular, **fractals**. For one of the most famous examples (often a programming interview question), check out the **Tower of Hanoi**[24] problem.



*A **Fractal Canopy**[25] drawn using recursion*

Consider the simplest example, a basic summation:

```java
public static int sum(int x) {
  int total = 0;
  for (int i = 1; i <= x; i++)
    total += i;
  return total;
}
```

[24] https://www.digitalocean.com/community/tutorials/tower-of-hanoi

[25] https://en.wikipedia.org/wiki/Fractal_canopy

In mathematics (especially Calculus and **Discrete Mathematics**), a **Summation** is represented by the *Sigma* (or Σ) operator, which adds all the numbers in a series together:

$$\text{total} = \sum_{i=1}^{x} i$$

This is our standard **Iterative** approach. To make it recursive, we first need to imagine running the summation backward from the number down to 1 instead of the opposite:

```java
public static int sumBackward(int x) {
  int total = 0;
  for (int i = x; i > 0; i--)
    total += i;
  return total;
}
```

Given this code, it is clear that our ending condition occurs when `x == 0`; this is known as the **Base Case**. All of our iterations will become our **Recursive Case** (the point that the method calls itself), like so:

```java
public static int sumRecursive(int x) {
  if (x == 0) return 0;           // Base case
  return x + sumRecursive(x-1);   // Recursive case
}
```

As you can see, our `sumRecursive` method follows the same principle as `sumBackward`: make sure we haven't reached 0; otherwise, add `x-1` to `x`. Hence, sumRecursive(5) $= 5+4+3+2+1 \rightarrow 15$.

Practical Uses of Recursion

Recursion is still (typically) iterative, but there are many approaches to recursion that can drastically increase the speed of an algorithm by breaking it down into smaller sub-problems and solving those problems through sub-problems and combining their solutions — a concept known as the **Divide-and-Conquer Method**[26], which is extended to even more practical concepts, such as **Dynamic Programming**[27] and the **Greedy Approach**[28]. As well, computer scientists study *(Abstract) Data Structures* that are built from recursions, such as the **Linked List** or **Binary Search Tree**, or their core methods rely on recursions such as **Graphs** and their graph search algorithms. These concepts are beyond the scope of this course but are extremely practical with real-world programs and massive datasets.

One great example of recursion is the **factorial** operator ($n!$ in mathematics), which returns $n! = n*(n-1)*(n-2)*(n-3)*\cdots*1$. For example, $5! = 5*4*3*2*1 \rightarrow 120$. Let's look at the code for this, both iteratively and recursively:

[26]https://www.programiz.com/dsa/divide-and-conquer
[27]https://www.programiz.com/dsa/dynamic-programming
[28]https://www.programiz.com/dsa/greedy-algorithm

```java
// Iterative
public static int fact(int n) {
  int product = 1;
  for (int i = 1; i <= n; i++)
    product *= i;
  return product;
}
// Recursive
public static int factRec(int n) {
  if (n == 1) return 1;
  return n * factRec(n-1);
}
```

Like summation, we have an operator for a series of multiplicands as well — the *Uppercase Pi* (or Π for **Product**) operator, which multiplies all the numbers in a series together:

$$\text{product} = n! = \prod_{i=1}^{n} i$$

Also, see the **Gamma Function**[29] $\Gamma(n)$, which extends the factorial function to complex numbers as well!

Another extremely popular example of recursion is the **Fibonacci Sequence** — a sequence of numbers that form the *Fibonacci spiral* and the *Golden Ratio*[30].

The Fibonacci numbers F_n are defined by the following recurrence relation (the ∀ symbol means *"for all"*; the opposite is ∃ meaning *"there exists"*):

$$F_0 = 0, \quad F_1 = 1, \quad \text{and} \quad F_n = F_{n-1} + F_{n-2} \quad \forall n > 1$$

For example, the first 10 Fibonacci numbers:

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9
0	1	1	2	3	5	8	13	21	34

We can easily implement this using recursion as follows:

```java
public static int fib(int n) {
  if (n <= 1) return n;
  return fib(n-1) + fib(n-2);
}
```

This series can also be represented as a fraction (much like most sums of a series), the *Golden Ratio*, which is famously represented symbolically by the letter ϕ:

$$\phi = \frac{1+\sqrt{5}}{2}$$

[29]https://en.wikipedia.org/wiki/Gamma_function
[30]https://en.wikipedia.org/wiki/Fibonacci_number

If you find the mathematical notation interesting, we could have also written the factorial function as being the product of the set of all numbers from [1, n] using **Set Notation**[31]:

$$\text{let } \mathbb{Z}_n^+ = 1, 2, 3, \ldots, n$$
$$n! = \prod_{i=1}^{n} x_i \quad \forall x \in \mathbb{Z}_n^+$$

Written out, this says that the set of positive integers[32] (known as $\mathbb{Z}^+$) subscript n equals the set of positive numbers from 1 to n. Hence, $n!$ is equal to the product Π of each number in the set multiplied together, where x_i is the value x at index i in the set (think of Π as a **for-each loop** basically). Also, check out **Ring Theory**[33]!

Recursive Searching and Improved Binary Search

Using recursion, we can apply the *divide-and-conquer* approach to improve upon search algorithms like **Binary Search** (along with search algorithms for abstract data structures like *Graphs*).

For example, we can restructure Binary Search to use recursion:

```
public static int binarySearchRec(int[] array, int x, int low, int high) {
  if (low > high) return -1;
  int mid = (low + high) / 2;
  if (array[mid] == x) return mid;
  if (array[mid] > x) return binarySearchRec(array, x, low, mid - 1);
  return binarySearchRec(array, x, mid + 1, high);
}

// Call with binarySearchRec(array, num, 0, array.length)
// or make a helper method that does this for you
```

Though they use two different approaches to the same implementation, both *iterative* and *recursive* Binary Search methods share the same **Time Complexity** ($O(\log n)$ to be precise), so there is not much of a difference in performance. However, the recursive call could also be combined with some **Parallel Processing** method such as *Multithreading*[34] or *Parallelization*[35] to allow each recursive call to be performed **concurrently**, rather than **iteratively** — although linear search could also be parallelized, removing the need for the array to be pre-sorted.

[31]https://www.mathsisfun.com/sets/symbols.html
[32]https://en.wikipedia.org/wiki/Integer
[33]https://en.wikipedia.org/wiki/Ring_theory
[34]https://www.geeksforgeeks.org/multithreading-in-java/
[35]https://livebook.manning.com/book/java-8-in-action/chapter-7/79

Recursive Sorting and Divide-and-Conquer Algorithms

While not necessary for the AP exam, the **Quick Sort**[36] and **Merge Sort**[37] algorithms are extremely important real-world sorting algorithms. There are also sorting algorithms that do not require numeric comparisons, such as **Counting Sort**[38] and **Radix Sort**[39].

Quicksort

Quicksort is a popular, very efficient sorting algorithm that operates by dividing a list of items into two smaller sub-lists, sorting those sub-lists, and then merging the sorted sub-lists back together to form a final, sorted list. The key to quicksort is dividing the list into sub-lists — a process known as partitioning. This approach to sorting through solving smaller sub-problems is known as the ***divide-and-conquer*** method. Its efficiency makes it extremely *quick* even for large datasets, making it very applicable to real-world data.

The steps are as follows:

1. Choose an element from the list, known as the pivot. This element will be used to divide the list into sub-lists
2. Divide the list into two sub-lists: one that contains all of the elements that are less than or equal to the pivot and one that contains all of the elements that are greater than the pivot. These sub-lists are known as the left and right partitions, respectively
3. Sort the left and right partitions by recursively applying the quicksort algorithm to each of them. This means that we will repeat steps 1-3 on each of the sub-lists until they are each sorted
4. Once the left and right partitions are sorted, merge them together to form a final, sorted list. To do this, we simply combine the left partition, the pivot, and the right partition in that order
5. The final, sorted list is returned

Quicksort is broken down into two methods, one for finding a partition and the other for performing the sort:

```java
private static int partition(int arr[], int low, int high) {
  int pivot = arr[high];
  int i = low - 1;
  for (int j = low; j < high; j++) {
    if (arr[j] <= pivot) {
      i++;
      int temp = arr[i];
      arr[i] = arr[j];
      arr[j] = temp;
    }
  }
```

[36]https://www.programiz.com/dsa/quick-sort
[37]https://www.programiz.com/dsa/merge-sort
[38]https://www.programiz.com/dsa/counting-sort
[39]https://www.programiz.com/dsa/radix-sort

```
  int temp = arr[i + 1];
  arr[i + 1] = arr[high];
  arr[high] = temp;
  return i + 1;
}

public static void quickSort(int array[], int low, int high) {
  if (low < high) {
    int pivot = partition(array, low, high);
    quickSort(array, low, pivot - 1);
    quickSort(array, pivot + 1, high);
  }
}

// Call using quickSort(array, 0, array.length-1) or make a helper
// method that does this for you, then rename the above function
// to "quickSortRecursive" and modify its recursive call
```

Merge Sort

Merge sort is another popular and very efficient sorting algorithm that works by dividing a list of items into two smaller sub-lists, sorting those sub-lists, and then *merging* the sorted sub-lists back together to form a final, sorted list. This process of dividing and merging is repeated until the entire list is sorted, making this another ***divide-and-conquer*** algorithm.

The steps are as follows:

1. Divide the list of items into two smaller sub-lists. This is typically done by splitting the list in half, although other methods of dividing the list are also possible
2. Sort each of the two sub-lists by recursively applying the merge sort algorithm to each of them. This means that we will repeat steps one and Two on each of the sub-lists until they are each sorted
3. Once the two sub-lists are sorted, we can merge them together to form a final, sorted list. This is done by comparing the first element of each sub-list and choosing the smaller of the two as the first element of the final list. We then repeat this process, comparing the second element of each sub-list until all of the elements from both sub-lists have been added to the final list
4. The final, sorted list is returned

Merge Sort is also broken down into two methods, one for merging two subarrays and the other for performing the sort:

```
private static void merge(int arr[], int p, int q, int r) {
  int n1 = q - p + 1;
  int n2 = r - q;
  int L[] = new int[n1];
  int M[] = new int[n2];
```

```java
  for (int i = 0; i < n1; i++)
    L[i] = arr[p + i];
  for (int j = 0; j < n2; j++)
    M[j] = arr[q + 1 + j];

  int i = 0;
  int j = 0;
  int k = p;

  while (i < n1 && j < n2) {
    if (L[i] <= M[j]) {
      arr[k] = L[i];
      i++;
    } else {
      arr[k] = M[j];
      j++;
    }
    k++;
  }

  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }

  while (j < n2) {
    arr[k] = M[j];
    j++;
    k++;
  }
}

public static void mergeSort(int array[], int l, int r) {
  if (l < r) {
    int m = (l + r) / 2;
    mergeSort(array, l, m);
    mergeSort(array, m + 1, r);
    merge(array, l, m, r);
  }
}

// Call using mergeSort(array, 0, array.length-1) or make a helper
// method that does this for you, then rename the above function
// to "mergeSortRecursive" and modify its recursive call
```

Unit 11 - Useful Non-AP Java Concepts for After the Exam

There are many programming and Computer Science concepts that the AP exam leaves out that, while not necessary for the test, are extremely useful in real-world software. The most important of these is the concept of **Number Bases**, which describe how numbers are represented in various contexts within computing systems. As well, there are many types of statements and keywords, such as **switch** statements, generic typing, and **Functional Programming** concepts that are used constantly in real applications but are left out of the exam. We will survey many of the most common ones here; though while not discussed, additional concepts to know include sockets and networking, cryptography, graphics libraries (such as JavaFX and Swing), game development, and web applications.

Number Bases

In a number system, the **Base** (or *Radix*) is the number of digits or distinct symbols that are used to represent numbers. For example, in the **Decimal** number system, which is the most commonly used system, the base is 10 since there are 10 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9) used to represent numbers.

Binary Encoding

Binary is a number system that uses only two digits, 0 and 1, to represent numbers. It is often used in computing and digital systems because it is a simple and efficient way to represent and manipulate data.

To represent a number in binary, you can use a series of digits to represent the value of each power of 2. For example, the binary number 1011 represents the decimal number 11.

To convert a number from **decimal to binary**, you can divide the number by 2 and keep track of the remainder until the result is 0. For example, to convert the decimal number 11 to binary:

$$\begin{aligned} 11/2 &= 5 \text{ remainder } 1 \\ 5/2 &= 2 \text{ remainder } 1 \\ 2/2 &= 1 \text{ remainder } 0 \\ 1/2 &= 0 \text{ remainder } 1 \end{aligned}$$

Hence, the binary representation of 11 is 1011.

To convert a number from **binary to decimal**, you can multiply each digit by the corresponding power of 2 and add the results. Let's convert the binary number 1011 back to decimal:

$$1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 8 + 0 + 2 + 1 = 11$$

Hexidecimal Encoding

Hexadecimal is one of the most important number systems that is used in computing. It uses a base of 16 with the digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F) to represent numbers (where A=10, B=11, etc.), allowing extremely compact number storage for large amounts of data.

To convert a number from **decimal to hexadecimal**, you can divide the number by 16 and keep track of the remainder until the result is 0. For example, to convert the decimal number 28 to hexadecimal:

$$\begin{aligned} 28/16 &= 1 \text{ remainder } 12 \\ 1/16 &= 0 \text{ remainder } 1 \end{aligned}$$

Hence, the hexadecimal representation of 28 is 1C.

To convert a number from **hexadecimal to decimal**, you can multiply each digit by the corresponding power of 16 and add the results. Let's convert the hexadecimal number 1C back to decimal:

$$1 * 16^1 + 12 * 16^0 = 16 + 12 = 28$$

The *do-while* Loop

A **do-while** loop is a simple looping structure that runs a block of code *at least once*, then repeatedly executes the block as long as a condition is true. The syntax is simply a **while** loop upside-down, where the condition is checked at the end of the loop (after iterating):

```
do {
  // code block to be executed
} while (condition);
```

The code block inside the loop will be executed at least once, even if the condition is initially false. Consider the following example:

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

Here, the code inside the loop will be executed 10 times because the condition `count > 0` is initially true. On each iteration, the value of `count` is decreased by 1 until it becomes 0, at which point the loop will terminate; hence, the loop prints out all the numbers from 10 to 1 (inclusive).

The *break* and *continue* Keywords

The `break` and `continue` keywords are simple but powerful statements used to alter the flow of a loop. Put simply, `break` will immediately exit the current loop (not all if it is nested within two loops, though) and `continue` skips all remaining code in the loop and starts the next iteration. We can either *break* to exit a loop early or *continue* and skip the current iteration of a loop to move on to the next one.

For example, using `break`:

```
for (int i = 0; i < 10; i++) {
  if (i == 5) {
    break;
  }
  System.out.println(i);
}
```

The loop will iterate from 0 to 9, but it will break out of the loop when `i` is equal to 5; hence, the output will be "0, 1, 2, 3, 4".

Likewise, `continue` can be used in a very similar manner:

```
for (int i = 0; i < 10; i++) {
  if (i % 2 == 0) {
    continue;
  }
  System.out.println(i);
}
```

Here, the loop iterates from 0 to 9, but it skips printing the value of `i` when it is even. Thus, the output of this loop will be "1, 3, 5, 7, 9".

The *switch* Statement

One very useful statement Java inherits from C, the **switch** statement, is used to execute a block of code based on the value of a given expression. It is similar to having multiple **if/else-if/else** statements, but it is often more efficient when you have a large number of possible conditions for one variable. We simply make a `switch` on the variable we want to check, and make a `case` for all possible values for that variable — however, we also need to `break` at the end of the case, or we can actually fall into the next case down. As well, we can also provide an optional `default` case which acts as our **else** condition if no other cases match.

For example:

```java
int day = 3;

switch (day) {
  case 1:
    System.out.println("Monday");
    break;
  case 2:
    System.out.println("Tuesday");
    break;
  case 3:
    System.out.println("Wednesday");
    break;
  case 4:
    System.out.println("Thursday");
    break;
  case 5:
    System.out.println("Friday");
    break;
  case 6:
    System.out.println("Saturday");
    break;
  case 7:
    System.out.println("Sunday");
    break;
  default:
    System.out.println("Invalid choice");
}
```

Here, the `day` variable is 3, so the code inside the `case 3:` block will be executed. Breaking at the end of each case exits the **switch** statement and prevents the code in the following case blocks from being executed, just like breaking inside a loop.

The *var* Type Name

In many programming languages, the `var` type name is used to declare a variable with an inferred type. It was introduced as a way to reduce the verbosity of code by allowing the compiler to implicitly determine the type of a variable based on the initializer expression.

For example:

```
public class MyClass {
  public static void main(String[] args) {
    var myStr = "Hello, world!";

    // The type of 'myStr' is inferred to be 'String' by the compiler
    System.out.println(myStr);
  }
}
```

It's important to note that `var` can ***only be used to declare local variables and cannot be used to declare class-level variables or method parameters***. Additionally, the type must be inferable from the initializer expression, so it cannot be used to declare variables with an initial value of null. While a matter of preference, the most appropriate use case for `var` would be for variables that are instances of classes or variables returned from methods, especially with extremely long names — however, ***almost NEVER should you use this with primitive types or inherited, abstract, or interface classes***, as the behavior may not be what you expect. As such, you should avoid using `var` until you have a strong grasp of inheritance and mental type inference (or look at the documentation!) especially.

Consider the following *appropriate* example:

```
var url = new URL("https://google.com/");
var con = url.openConnection();
var rdr = new BufferedReader(new InputStreamReader(con.getInputStream()));
```

Here, it is much clearer and easier to read our code if the left-hand side is less cluttered, which would normally look as follows:

```
URL url = new URL("https://google.com/");
URLConnection con = url.openConnection();
Reader rdr = new BufferedReader(new InputStreamReader(con.getInputStream()));
```

The Ternary Operator

The **Ternary Operator** is a conditional operator that can be used to assign a value to a variable based on a *boolean condition*. It uses the following syntax:

```
variable = condition ? value1 : value2;
```

In this format, `condition` is a boolean expression, and `value1` and `value2` are values/expressions of any type. If the condition is true, the ternary operator will evaluate to `value1`, otherwise, it will evaluate to `value2`.

Consider the following example:

```
int x = 10;
int y = 20;

int max = (x > y) ? x : y;
System.out.println(max);  // Output: "20"
```

Here, the boolean expression `x > y` is false, so the ternary operator evaluates to `y` and thus `max` is assigned as 20.

The ternary operator is often used as a shorthand way to write simple **if-else** statements. For example:

```
int x = 10;
int y = 20;

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

System.out.println(max);  // Output: "20"
```

is equivalent to the previous code using the ternary operator.

Enumerations

In Java, an `enum` (or **enumeration**) is a special data type that represents a fixed set of values. It is used to define a list of predefined constants, which can be useful when you want to define a set of values that a variable can take on and you want to ensure that the variable can only be assigned one of those specific values. One such example is the set of possible keys a user may press in a video game — which is extremely useful when paired with a `switch` statement.

For example:

```
public enum Day {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

// In main...
Day today = Day.MONDAY;
System.out.println("Today is " + today);
// Output: "Today is MONDAY"
```

Enums can also have additional methods and fields, and you can use them in a similar way to classes. For example:

```
public enum Day {
  MONDAY("Monday"), TUESDAY("Tuesday"), WEDNESDAY("Wednesday"),
  THURSDAY("Thursday"), FRIDAY("Friday"), SATURDAY("Saturday"),
  SUNDAY("Sunday");

  private final String fullName;

  private Day(String fullName) {
    this.fullName = fullName;
  }

  public String getFullName() {
    return fullName;
  }
}

Day today = Day.MONDAY;
System.out.println("Today is " + today.getFullName());
// Output: "Today is Monday"
```

Now we have added a field fullName and a constructor to the enum **Day**, as well as a method `getFullName()` that returns the full name of the day in title case instead of all uppercase.

Generic Typing

In Java, **Generics** (sometimes called *Templates* from C++) allow you to write code that can work with multiple types while still providing type safety. This is extremely useful and powerful when you want to create a class or method that can work with multiple types, but want to ensure that the types used are compatible. Generally, we represent this by appending the suffix `<T>` (or any other name like `<TYPE>` or multiple using `<T, U>` etc.) to the end of the class name. We have seen this exact syntax when using the **ArrayList** class, which we noted requires us to use the wrapper class for primitive types.

Consider the following simple example:

```
public class MyClass<T> {
  private T value;

  public MyClass(T value) {
    this.value = value;
  }

  public T getValue() {
    return value;
  }
}
```

Here, the class MyClass has a *generic type* `T`. The `T` can be replaced with any type when the class is instantiated. For example:

```
MyClass<String> stringClass = new MyClass<>("hello");
MyClass<Integer> intClass = new MyClass<>(123);
// Note that with generic classes, we can technically drop including
// the type on the right-hand side as long as we define it
// on the left - i.e., if we are not using `var`
```

The `T` type is used in the method `getValue()`, which returns an object of type `T`.

We can also make generic methods:

```
public static <T>T getMiddleElement(T[] array) {
  return array[array.length / 2];
}
```

The `getMiddleElement()` method has a generic type `T`, which represents the type of elements in the array. The method can be called with any type of array, given that the type is compatible with the return type of the method:

```
String[] strArr = {"a", "b", "c"};
String sMid = getMiddleElement(strArr);
Integer[] intArr = {1, 2, 3};
Integer iMid = getMiddleElement(intArr);
```

Functional Programming

The concept of **Functional Programming (FP)** is a programming paradigm that emphasizes the use of functions to model and solve problems. In functional programming, functions are treated as ***first-class citizens***, meaning that they can be passed as arguments to other functions, returned as values from functions, and assigned to variables.

Functional programming has a number of benefits, including:

1. **Simplicity**: By focusing on functions and immutable data, functional programming can help keep code simple and easy to reason about
2. **Modularity**: Functions can be easily composed and reused, making it easy to build modular, maintainable code
3. **Concurrency**: Because functional programming relies on immutable data and side-effect-free functions, it can make it easier to write concurrent and parallel code

Functional programming is a powerful tool for solving a wide range of problems and can be used in many different programming languages. In Java, FP can be achieved using **lambda expressions** especially, which allow you to create **anonymous functions** that can be passed as arguments to other functions.

One of the main benefits of using functional programming is that it allows you to write concise, expressive code that is easy to read and understand. This can make it easier to write and maintain code, particularly in large projects with many developers. It is especially useful in Java when working with data streams and collections, as it allows you to easily apply transformations and filters to data in a declarative way, as well as when working with concurrency and parallelism.

Function Pointers or Functions as Objects

One extremely useful concept of FP is the idea of **Function Pointers**, also known as function references or function handles, which are references to functions that can be stored, passed around, and invoked. To use them, we can import the `java.util.function.Function` interface to ***define or store functions as variables***.

For example, you can use the **Function** interface to define a function pointer that takes an integer as input and returns a string:

```java
import java.util.function.Function;

public class FuncPointerExamples {
  public static void main(String[] args) {
    // Define a function pointer that takes
    // an integer argument and returns a string
    Function<Integer, String> intToStr = i -> String.valueOf(i);

    // Use the function pointer to convert an integer to a string
    String str = intToStr.apply(123);
    System.out.println(str);
  }
}
```

You can also use *Method References* to create function pointers using the `::` operator in the form `CLASSNAME::METHODNAME`:

```java
import java.util.function.Function;

public class MethodRefPointers {
  public static double Exp(double n) {
    return Math.exp(n);
  }

  public static void main(String[] args) {
    Function<String, Integer> strLen = String::length;
    int length = strLen.apply("hello");
    System.out.println(length);

    Function<Double, Double> myExp = MethodRefPointers::Exp;
    double result = myExp.apply(3.0);
    System.out.println(result);
  }
}
```

Definitions of Derivative and Integral

One of the most practical ways we can use function pointers is to make methods that take functions as arguments. For example, consider the Definition of the **Derivative**[40] from Calculus:

$$f'(x) = \frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

This definition provides an approach to evaluating the slope (or rate of change) at any point within a function $f(x)$. Using function pointers, we can make a method and use the pointer to that method to calculate its derivative at some point using an arbitrarily small number $h \approx 0.000...1 \equiv 10^{-\varepsilon}$ for the limit. Likewise, we can find the nth derivative (written $f^{(n)}(x)$ or $\frac{d^n f}{dx^n}$) by recursively taking the derivative of the derivative n times and approximating a good h value as $h \approx \frac{1}{10^n}$.

Consider the following example using $x = 5$, $f(x) = x^3$, and by the *Power Rule*, $f'(x) = 3x^2$, $f''(x) = 6x$, and $f'''(x) = 6$:

```java
import java.util.function.Function;

public class Derivatives {
  public static double f(double x) {
    return Math.pow(x, 3);
  }

  public static double derive(Function<Double, Double> f, double x) {
    final double h = 0.0000001;
    return (f(x + h) - f(x)) / h;
  }

  public static double
  derive(Function<Double, Double> f, double x, double h) {
    return (f(x + h) - f(x)) / h;
  }

  public static double
  deriveN(Function<Double, Double> f, double x, int n) {
    final double h = 1/Math.pow(10, n);
    if (n == 1) return derive(f, x);
    else return (deriveN(f, x + h, n - 1) - deriveN(f, x, n - 1)) / h;
  }
```

[40] https://tutorial.math.lamar.edu/classes/calci/defnofderivative.aspx

```
  public static void main(String[] args) {
    double x = 5;
    System.out.println("x = " + x);
    System.out.println("Does derive(f,x) match the power rule? "
          + "f(x)=x^3 so f'(x)=3x^2: " + (3 * Math.pow(x, 2)));
    System.out.println("f(x) = " + f(x));
    System.out.println("f'(x) = " + derive(Derivatives::f, x));
    System.out.println("The second derivative using the power rule "
          + "yields f''(x)=6x: " + (6 * x));
    System.out.println("f''(x) = " + deriveN(Derivatives::f, x, 2));
    System.out.println("The third derivative using the power rule "
          + "yields f'''(x)=6: " + 6);
    System.out.println("f'''(x) = " + deriveN(Derivatives::f, x, 3));
  }
}

/* Displays the following:
   x = 5.0
   Does derive(f,x) match the power rule? f(x)=x^3 so f'(x)=3x^2: 75.0
   f(x) = 125.0
   f'(x) = 75.00000165805432
   The second derivative using the power rule yields f''(x)=6x: 30.0
   f''(x) = 30.029994491087564
   The third derivative using the power rule yields f'''(x)=6: 6
   f'''(x) = 6.011191544530448
*/
```

Here we overload the `derive` method to either accept a value for h or use the arbitrarily small default value provided in the first definition, which can be useful if a certain function necessitates a different value.

We can also apply this same concept to the Definition of the **Definite Integral** as the Limit of a Riemann Sum[41] (also known as the *Antiderivative*) using the *Midpoint Rule* and some arbitrarily large number of divisions $n = 10^{\varepsilon}$ to approximate:

$$F(x) = \int_a^b f(x)dx = \lim_{n\to\infty} \sum_{i=1}^{n} f(c_i)\Delta x$$

where

$$\Delta x = \frac{b-a}{n}$$
$$x_i \approx a + i * \Delta x$$
$$c_i = \frac{x_i + x_{i+1}}{2}$$

[41] https://www.sfu.ca/math-coursenotes/Math%20158%20Course%20Notes/sec_riemann.html

This definition provides an approach to evaluating the area under a curve between two interval points $[a, b]$ below a function $f(x)$.

Consider the following example using $a = 1$, $b = 5$, $f(x) = x^3$, and by the *Power Rule* and *the Fundamental Theorem of Calculus*, $F(x) = \int_a^b x^3 dx = [\frac{x^4}{4}]_a^b = \frac{b^4}{4} - \frac{a^4}{4}$:

```java
import java.util.function.Function;

public class Integrals {
  public static double f(double x) {
    return Math.pow(x, 3);
  }

  public static double
  integrate(Function<Double, Double> f, double a, double b, int n) {
    double sum = 0;
    double deltaX = (b - a) / n;
    for (int i = 0; i < n; i++)
      sum += f(((a + i * deltaX) + (a + (i + 1) * deltaX)) / 2) * deltaX;
    return sum;
  }

  public static void main(String[] args) {
    double a = 1;
    double b = 5;
    int n = 100000000;
    System.out.printf("a = %f\tb = %f\n", a, b);
    System.out.println("Does integrate(f,x) match the power rule? "
          + "By the Fundamental Theorem,");
    System.out.println("\tf(x)=x^3 so F(x)[a,b]=(b^4/4)-(a^4/4): "
          + ((Math.pow(b, 4) / 4) - (Math.pow(a, 4) / 4)));
    System.out.println("f(x) = " + f(b - a));
    System.out.println("F(x) = " + integrate(Integrals::f, a, b, n));
  }
}
/* Displays the following:
   a = 1.000000    b = 5.000000
   Does integrate(f,x) match the power rule? By the Fundamental Theorem,
           f(x)=x^3 so F(x)[a,b]=(b^4/4)-(a^4/4): 156.0
   f(x) = 64.0
   F(x) = 156.0000000000034
*/
```

Note that you could modify the Riemann estimation to use different approximation rules by simply changing the line inside the *for* loop and modifying *n*:

- **Left Hand Rule**: `sum += f(a + i * deltaX) * deltaX;`
- **Right Hand Rule**: `sum += f(a + (i+1) * deltaX) * deltaX;`
- **Midpoint Rule**: `sum += f(((a + i * deltaX) + (a + (i + 1) * deltaX)) / 2) * deltaX;`

Anonymous Functions and Lambda Expressions

An **Anonymous Function** is simply a function without a name. It is defined and used in a single statement, typically as an argument to a method or constructor. These are also known as **Lambda Expressions**, coming from their origin of *Lambda Calculus*[42].

Consider the following anonymous function that defines a function to add two integers and returns the result:

```
((int x, int y) -> x + y);
```

This function takes two integer arguments `x` and `y` and returns their sum.

We can also use this with the `Collections.sort()` method to define a custom sorting behavior for a list or array, such as in descending order:

```
List<String> words = Arrays.asList("apple", "banana", "cherry");
Collections.sort(words, (a, b) -> b.compareTo(a));
System.out.println(words); // prints "[cherry, banana, apple]"
```

Here, the anonymous function takes two strings `a` and `b` and compares them using the `compareTo()` method. The `Collections.sort()` method thus sorts the list of strings in descending order by passing the lambda expression as the comparator.

Anonymous functions can be used in any context where a **Functional Interface** is expected, which is an interface with a single abstract method. For example, the `Comparator` interface is a functional interface because it has a single abstract method `int compare(T o1, T o2)`.

Here are a few additional examples:

1. Defining a function to add two integers and return the result:

```
((int x, int y) -> x + y);
```

2. Defining a function to check if a string is empty:

```
(String s) -> s.isEmpty();
```

3. Defining a function to compare two strings based on their length:

```
(String s1, String s2) -> s1.length() - s2.length();
```

4. Defining a function to print a message:

```
() -> System.out.println("Hello, world!");
```

5. Defining a function to convert a string to uppercase:

```
(String s) -> s.toUpperCase();
```

[42] https://en.wikipedia.org/wiki/Lambda_calculus

These lambda expressions can be passed as arguments to methods or used to define functional interfaces. For example, the following code uses a lambda expression to define a **Runnable** object, which is a functional interface with a single abstract method `void run()`:

```
Runnable r = () -> System.out.println("Running!");
```

Map, Reduce, and Filter

In functional programming, **Map**, **Reduce**, and **Filter** are ***Higher-Order Functions*** (functions that take another function as an argument) that allow you to apply a function to a collection of elements and produce a new collection as a result. These three functions are the pillars of FP — their proper usage enables very powerful, concise code, especially when applied to large collections of data.

- **Map** applies a function to each element in a collection and returns a new collection with the results (coming from the *map* operation[43] in Discrete Math denoted by the $\mapsto$ operator). For example, given a list of numbers, you can use `map` to apply a function that increases each number by 1:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
List<Integer> incremented = numbers.stream()
                                   .map(x -> x + 1)
                                   .collect(Collectors.toList());
// incremented = [2, 3, 4, 5]
```

- **Reduce** combines all the elements in a collection into a single value by applying a function to the elements in a specific order. For example, given a list of numbers, you can use `reduce` to sum them up:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
int sum = numbers.stream().reduce(0, (x, y) -> x + y);
// sum = 10
```

- **Filter** returns a new collection that includes only the elements that meet a certain condition. For example, given a list of numbers, you can use `filter` to select only the even numbers:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
List<Integer> evens = numbers.stream()
                             .filter(x -> x % 2 == 0)
                             .collect(Collectors.toList());
// evens = [2, 4]
```

These functions can even be composed together to perform more complex operations on collections. You can use `filter` to select a subset of a collection, then `map` to transform the elements in that subset, and finally `reduce` to combine the transformed elements into a single value.

[43] https://en.wikipedia.org/wiki/Map_(mathematics)

Miscellaneous

If you want to go even further into computer science, I have additional lesson plans on my website,[44] including:

- **AI and Machine Learning for AP:** http://danielszelogowski.com/resources/apcsamachinelearning/
- **Python Crash Course for AP:** http://danielszelogowski.com/resources/PythonCrashCourseForAP.html
- **Taking Programming Language Notes using Jupyter Notebooks:** http://danielszelogowski.com/resources/JupyterLanguageNotebooks.pdf

[44] http://danielszelogowski.com/education.php

Appendix

This appendix contains a quick reference for the major keywords, operators, escape sequences, and format specifiers in the Java programming language used in the AP subset.

A - Operators

Arithmetic Operators:

- `+`: plus
- `-`: minus
- `*`: times
- `/`: divide
- `%`: modulus (or MOD) — returns the remainder from long (integer) division

Assignment Operators:

- `x = y`: set *variable* on left side equal to *value* (or variable) on the right side
- `x += y`: shorthand for `x = x + y`; add `y` to the current value of `x`
- `x -= y`: shorthand for `x = x - y`
- `x *= y`: shorthand for `x = x * y`
- `x /= y`: shorthand for `x = x / y`
- `x %= y`: shorthand for `x = x % y`
- `x++`: shorthand for `x = x + 1` or `x += 1`
- `x--`: shorthand for `x = x - 1` or `x -= 1`

Unary Operators:

- `()`: parenthesis
- `[]`: array subscript (index operator)
- `.`: object member selection
- `+`: unary plus (represent a number as positive `+2`)
- `-`: unary minus (represent a number as negative `-2`)
- `(type)`: type cast

Conditional Operators:

- `==`: exactly equal to
- `!=`: not exactly equal to
- `>`: greater than
- `<`: less than
- `>=`: greater than or equal to
- `<=`: less than or equal to
- `instanceOf`: class membership

Logical Operators:

- `&&`: AND
- `||`: OR
- `!`: NOT

B - Keywords

Reserved Keywords:

abstract	boolean	catch	char	class	double	else
extends	final	for	if	implements	import	instanceof
int	interface	new	package	private	protected	public
return	static	super	this	try	void	while

Literals:

- `true`
- `false`
- `null`

C - Escape Sequences and Format Specifiers

Escape Characters:

- `\n`: new line (line break)
- `\t`: tab space
- `\'`: single quote
- `\"`: double quote
- `\\`: backslash

Format Specifiers:

- `%d`: int
- `%f`: double (`%.#f` to round to # decimal places)
- `%c`: char
- `%s`: String

Good luck on the AP exam!