High-Level Program Design

Java Development Accelerator

GENERAL ASSEMBLY

Upskill the Broader Organization

Java Development Accelerator

Instructor-Led Onsite or Remote ~35 Hours

Overview:
Employees will learn foundational Java skills and bring them back to your business. By the end of the program participants will be able to read and write object-oriented code, debug, and run unit tests to ensure proper behavior in your Java applications.

Business Outcomes:.
- Expand technical competencies on and off your technical team.
- Deploy modern engineering practices into your organization.
- Use one of the world's most popular, secure and reliable coding language.

~35 Hours of Java Development Training

Introduction to Java	Object-Oriented Programming	Design Patterns	Debugging and Exception Handling	Test-Driven Development
Get to know the basic elements of Java such as data types, control flow, methods, and classes.	**Build on existing knowledge** to understand how OOP concepts are implemented in Java.	**Learn how and why the most important design patterns** are used to solve problems	**Explore how debugging and exception handling work** in order to build programs that recover on their own.	**Understand how to run unit tests to confidently change code** without breaking everything around it.

Why Java Development Accelerator?

- **Validated tools and approaches** for programming in Java, developed in partnership with top organizations.
- **Built with subject matter experts** with experience in Java development and object-oriented programming.
- 35 hours of expert-led, hands-on learning, including:
    - **Projects, labs, and assignments** that mimic real-world tasks and workflows.
    - **Case studies and examples** that demonstrate how businesses use the concepts.
    - **Regular feedback and touchpoints** with instructors and peers to ensure students meeting learning goals.

Learner Personas for Java Development Accelerator

This product is specifically designed for the following audience:

- **Junior- to mid-level developers** with some experience in another programming language (e.g., JavaScript, Python, Ruby).

Anatomy of Java Development Accelerator

Unit	What's Covered	Unit Lab
Unit 1: **Introduction to Java** **(12 hours)** Get to know the basic elements of the Java programming language.	● Benefits of Java ● Data types and variables ● Control flow and loops ● Methods and scope ● Arrays and ArrayLists	Write methods that incorporate control flow, conditional logic, and data collections to solve programming challenges in Java.
Unit 2: **Object-Oriented Programming** **(12 hours)** Understand OOP concepts and how they are implemented in Java.	● Principles of object-oriented programming (OOP) ● Objects and classes ● Subclasses ● Abstract classes and interfaces ● Inheritance and abstraction	Create a "zoo" program to create animal types using classes, subclasses, properties, getters, and setters. Build a "household" application that creates and maintains households using abstract classes and interfaces. Design a program that manages a school with abstract classes, interfaces, and subclasses.

Anatomy of Java Development Accelerator

Unit	What's Covered	Unit Lab
Unit 3: **Test-Driven Development** **(8 hours)** Use debugging, exception handling, and unit testing techniques to ensure proper behavior in Java applications.	● Debugging ● Exception handling ● jUnit ● Types of application testing	Use jUnit to write unit tests for previously created Java methods.
Unit 4: **Java Project** **(8 hours)** Create a version of rock–paper–scissors that allows users to play against the computer in the console.		The game consists of a few main features: ● Play rock–paper–scissors against a computer player. ● Play rock–paper–scissors against a human player. The game should include Java concepts covered in class: exception handling; unit tests; classes and objects; abstract classes; and more.

Tools Used in the Course

- JDK 8
- The course is written for MacOS; if students or instructors use a Windows or Linux system, some modifications to the curriculum may have to be made.
- Git and Github Enterprise
- IntelliJ IDEA Community Edition
- Zoom, Slack, and Google Chrome

Java Development Accelerator

What You’ll Learn

Module Overview

12 hours

Introduction to Java

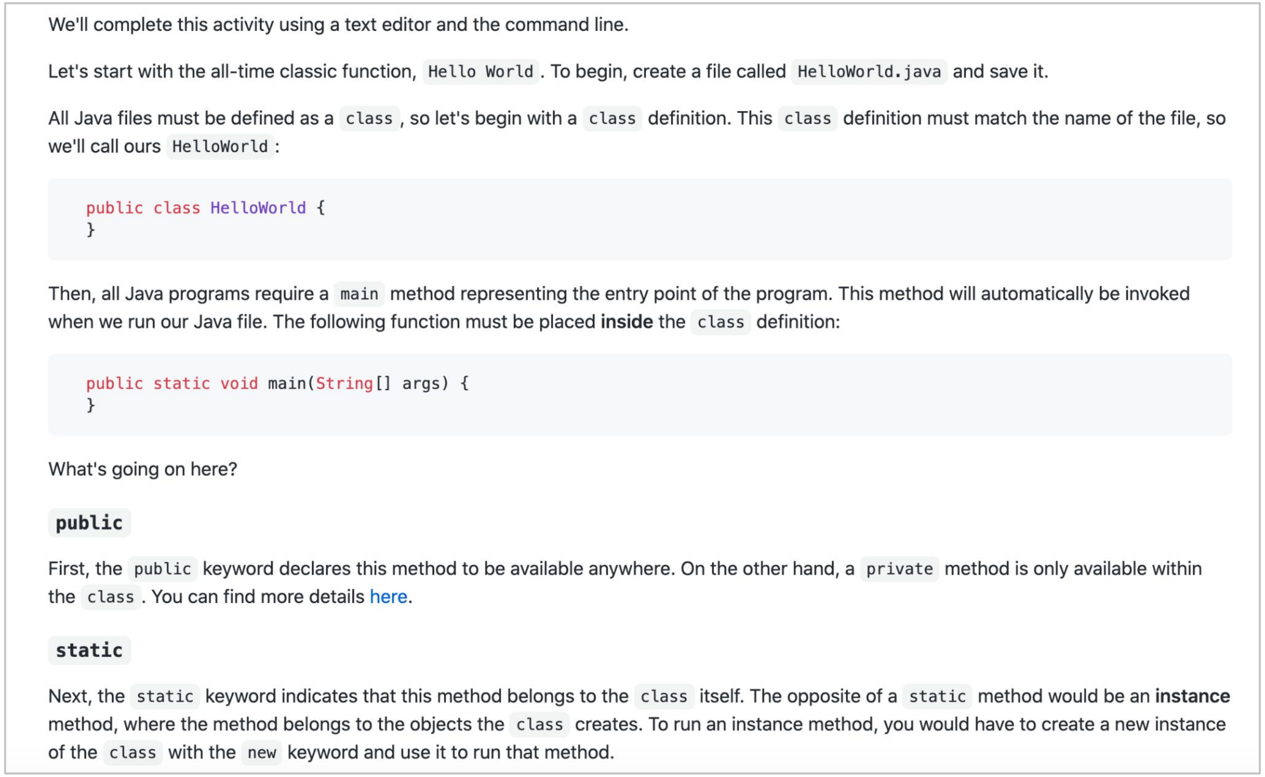
Why?

Java is the most popular coding language among engineers and is popular with enterprises due to its robustness, ease of use, multi-platform capabilities, and security features. In this unit, students will dive into the basics of the Java programming language.

Learning Objectives

- Write the main Java method.
- Identify and describe Java data types and their use cases.
- Describe the different types of variables and when to use them.
- Use if…else conditionals to control program flow.
- Use comparison operators to evaluate and compare statements.
- Compare types of variable scope.
- Identify the parts of a method.
- Create and manipulate arrays and array lists.

Hello World in Java



We'll complete this activity using a text editor and the command line.

Let's start with the all-time classic function, `Hello World`. To begin, create a file called `HelloWorld.java` and save it.

All Java files must be defined as a `class`, so let's begin with a `class` definition. This `class` definition must match the name of the file, so we'll call ours `HelloWorld`:

```java
public class HelloWorld {
}
```

Then, all Java programs require a `main` method representing the entry point of the program. This method will automatically be invoked when we run our Java file. The following function must be placed **inside** the `class` definition:

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

What's going on here?

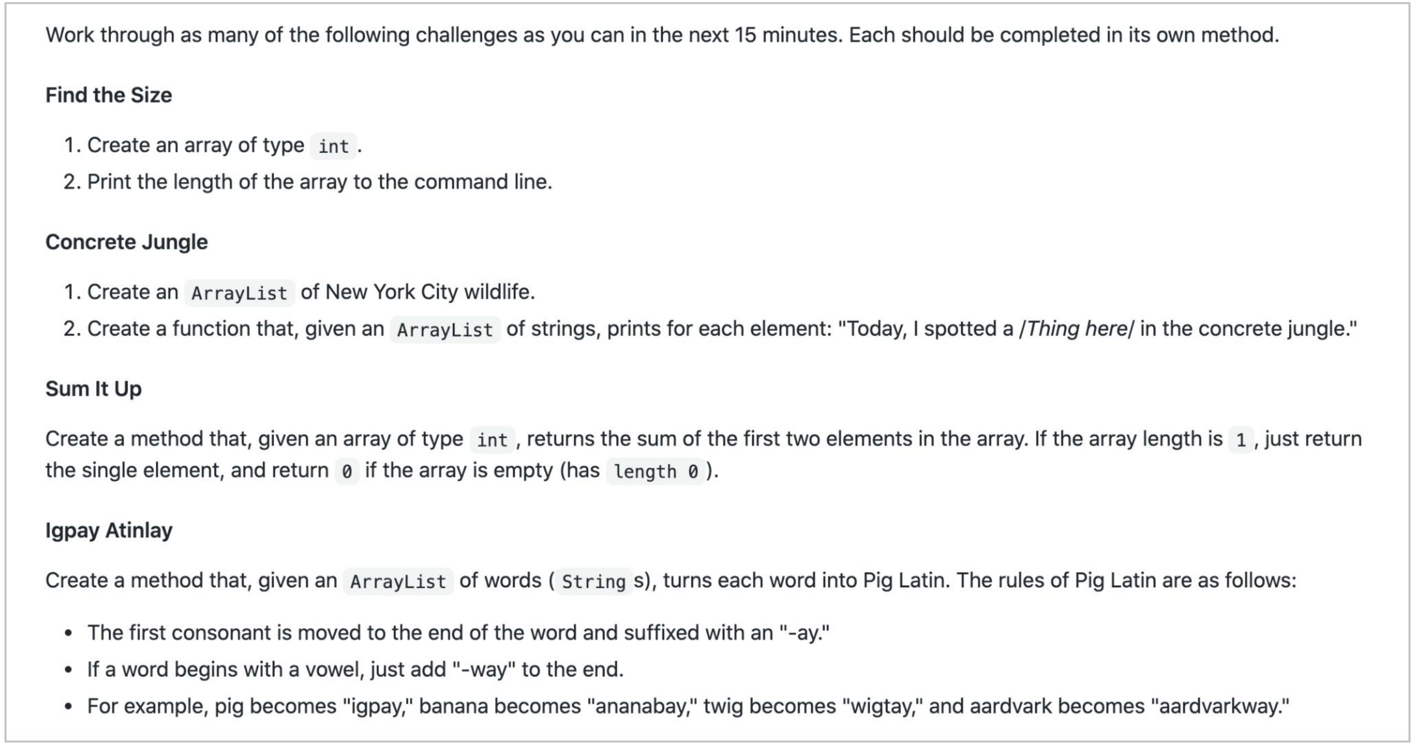
`public`

First, the `public` keyword declares this method to be available anywhere. On the other hand, a `private` method is only available within the `class`. You can find more details here.

`static`

Next, the `static` keyword indicates that this method belongs to the `class` itself. The opposite of a `static` method would be an **instance** method, where the method belongs to the objects the `class` creates. To run an instance method, you would have to create a new instance of the `class` with the `new` keyword and use it to run that method.

Creating Java Collections



Work through as many of the following challenges as you can in the next 15 minutes. Each should be completed in its own method.

Find the Size

1. Create an array of type `int`.
2. Print the length of the array to the command line.

Concrete Jungle

1. Create an `ArrayList` of New York City wildlife.
2. Create a function that, given an `ArrayList` of strings, prints for each element: "Today, I spotted a /*Thing here*/ in the concrete jungle."

Sum It Up

Create a method that, given an array of type `int`, returns the sum of the first two elements in the array. If the array length is `1`, just return the single element, and return `0` if the array is empty (has `length 0`).

Igpay Atinlay

Create a method that, given an `ArrayList` of words (`String`s), turns each word into Pig Latin. The rules of Pig Latin are as follows:

- The first consonant is moved to the end of the word and suffixed with an "-ay."
- If a word begins with a vowel, just add "-way" to the end.
- For example, pig becomes "igpay," banana becomes "ananabay," twig becomes "wigtay," and aardvark becomes "aardvarkway."

Module Overview

12 hours

Object-Oriented Programming

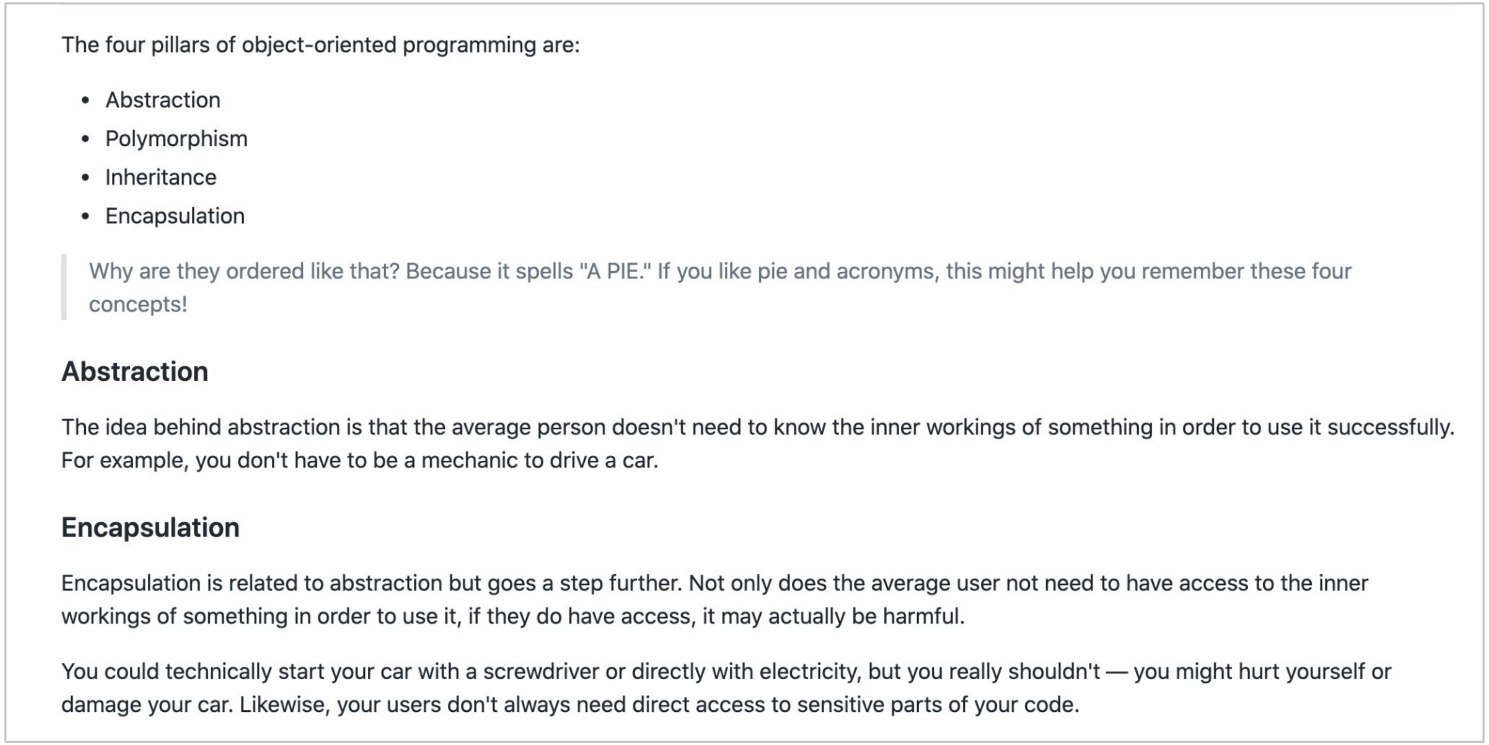
Why?

Object-oriented programming (OOP) helps developers write reusable, flexible, and modular code, allowing for easy troubleshooting, debugging, and maintenance. In this unit, students will explore how the principles of OOP are applied in Java.

Learning Objectives

- Define the four principles of object-oriented programming.
- Instantiate an object from a class.
- Add fields and methods to a class.
- Write constructors (including ones that accept arguments).
- Create a subclass.
- Use a subclass to augment the functionality of the base class.
- Write an abstract class.
- Define an interface in Java.
- Understand when to use an abstract class versus an interface.

Introduction to OOP



The four pillars of object-oriented programming are:

- Abstraction
- Polymorphism
- Inheritance
- Encapsulation

> Why are they ordered like that? Because it spells "A PIE." If you like pie and acronyms, this might help you remember these four concepts!

Abstraction

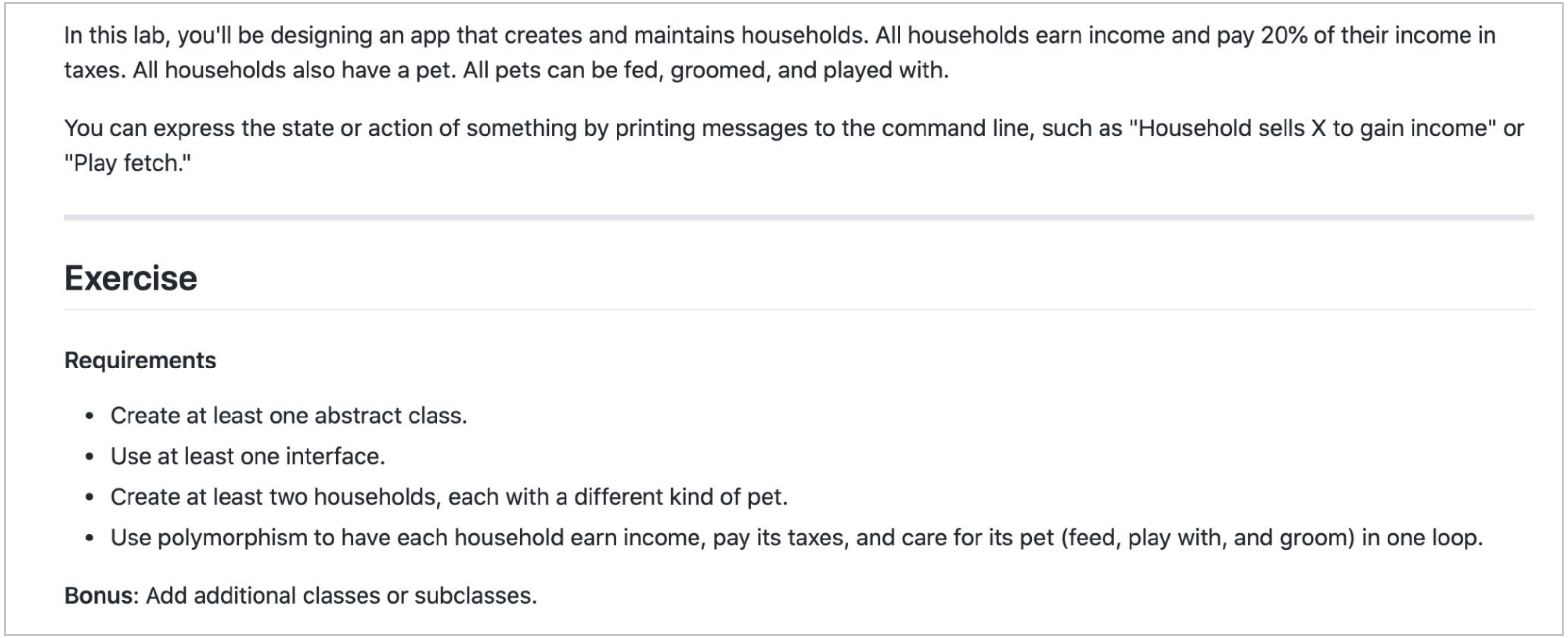
The idea behind abstraction is that the average person doesn't need to know the inner workings of something in order to use it successfully. For example, you don't have to be a mechanic to drive a car.

Encapsulation

Encapsulation is related to abstraction but goes a step further. Not only does the average user not need to have access to the inner workings of something in order to use it, if they do have access, it may actually be harmful.

You could technically start your car with a screwdriver or directly with electricity, but you really shouldn't — you might hurt yourself or damage your car. Likewise, your users don't always need direct access to sensitive parts of your code.

Applying Inheritance



In this lab, you'll be designing an app that creates and maintains households. All households earn income and pay 20% of their income in taxes. All households also have a pet. All pets can be fed, groomed, and played with.

You can express the state or action of something by printing messages to the command line, such as "Household sells X to gain income" or "Play fetch."

Exercise

Requirements

- Create at least one abstract class.
- Use at least one interface.
- Create at least two households, each with a different kind of pet.
- Use polymorphism to have each household earn income, pay its taxes, and care for its pet (feed, play with, and groom) in one loop.

Bonus: Add additional classes or subclasses.

Module Overview

Test-Driven Development

8 hours

Why?

The goal of test-driven development (TDD) is to deliver high-quality code as quickly and efficiently as possible. In this unit, students will learn about debugging, exception handling, and unit testing in Java — tools that will help them to live up to the goals of TDD.

Learning Objectives

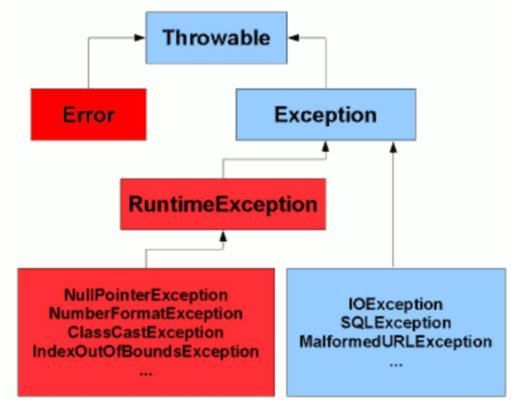
- Set a breakpoint and look at the current state given sample code.
- Identify which lines are broken in an application given a stack trace.
- Describe what a stack trace is saying.
- Describe an exception.
- Describe the purpose of try-catch blocks.
- Implement try-catch blocks.
- Add conditional methods to a test case.
- Perform a JUnit test.

The Basics of Exceptions

Before we start talking about the `try-catch` block, we need to talk about exceptions. **Exceptions** are events that occur while a program is running and interrupt the normal flow of the code. These can be null-pointer exceptions, divide-by-zero exceptions, index-out-of-bounds exceptions, and more. You can review many of the built-in exceptions in the Java documentation.

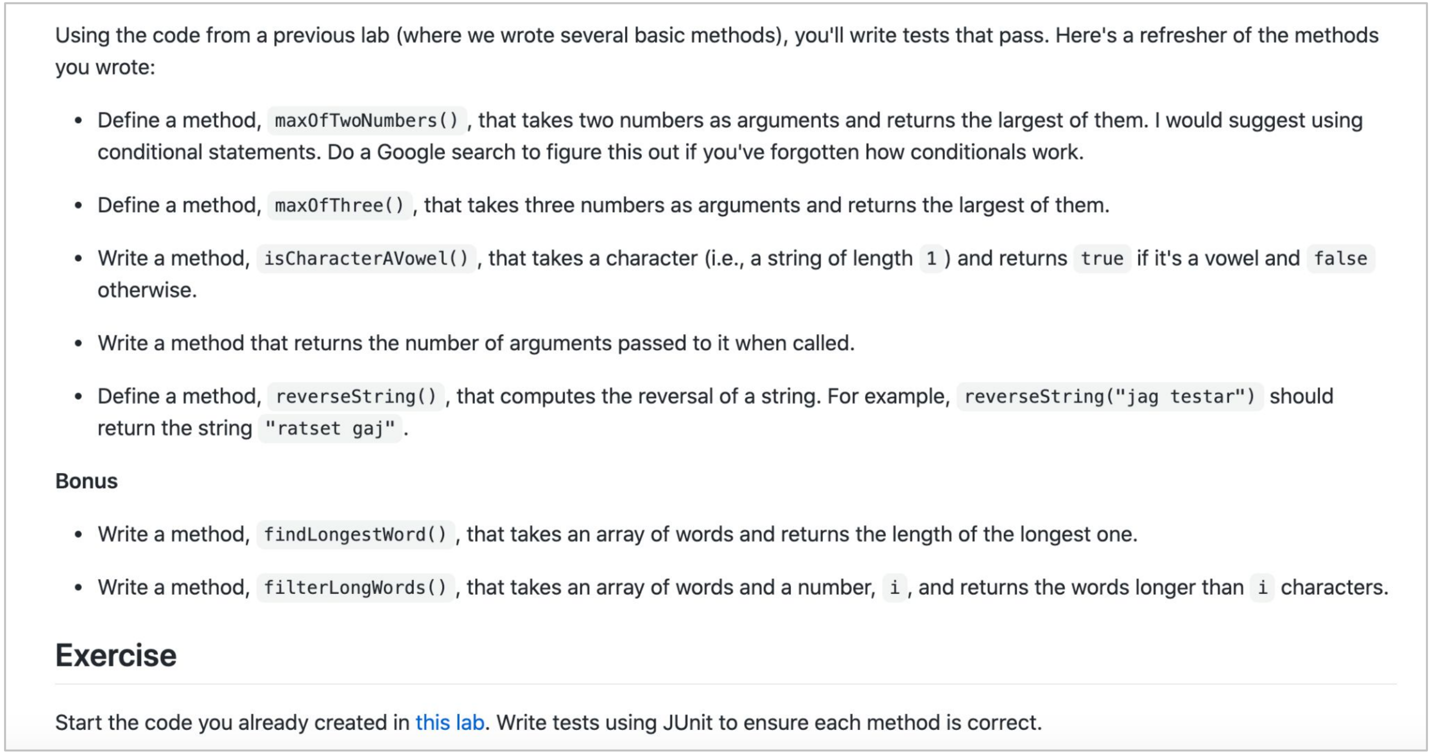
There are two types of exceptions: checked and unchecked. A **checked exception** occurs at compile time, which means a programmer is forced to handle these exceptions; otherwise, the program won't compile. Checked exceptions are subclasses of the `Exception` class. An **unchecked exception** — also known as a **runtime exception** — occurs at the time a program executes. You don't have to handle them, but you can if you'd like.

Below is an illustration of the exception hierarchy. Red denotes unchecked exceptions, while blue denotes checked ones:



Source

Testing: Man vs. Machine



Using the code from a previous lab (where we wrote several basic methods), you'll write tests that pass. Here's a refresher of the methods you wrote:

- Define a method, `maxOfTwoNumbers()`, that takes two numbers as arguments and returns the largest of them. I would suggest using conditional statements. Do a Google search to figure this out if you've forgotten how conditionals work.
- Define a method, `maxOfThree()`, that takes three numbers as arguments and returns the largest of them.
- Write a method, `isCharacterAVowel()`, that takes a character (i.e., a string of length `1`) and returns `true` if it's a vowel and `false` otherwise.
- Write a method that returns the number of arguments passed to it when called.
- Define a method, `reverseString()`, that computes the reversal of a string. For example, `reverseString("jag testar")` should return the string `"ratset gaj"`.

Bonus

- Write a method, `findLongestWord()`, that takes an array of words and returns the length of the longest one.
- Write a method, `filterLongWords()`, that takes an array of words and a number, `i`, and returns the words longer than `i` characters.

Exercise

Start the code you already created in this lab. Write tests using JUnit to ensure each method is correct.

Module Overview

8 hours

Java Project

Why?

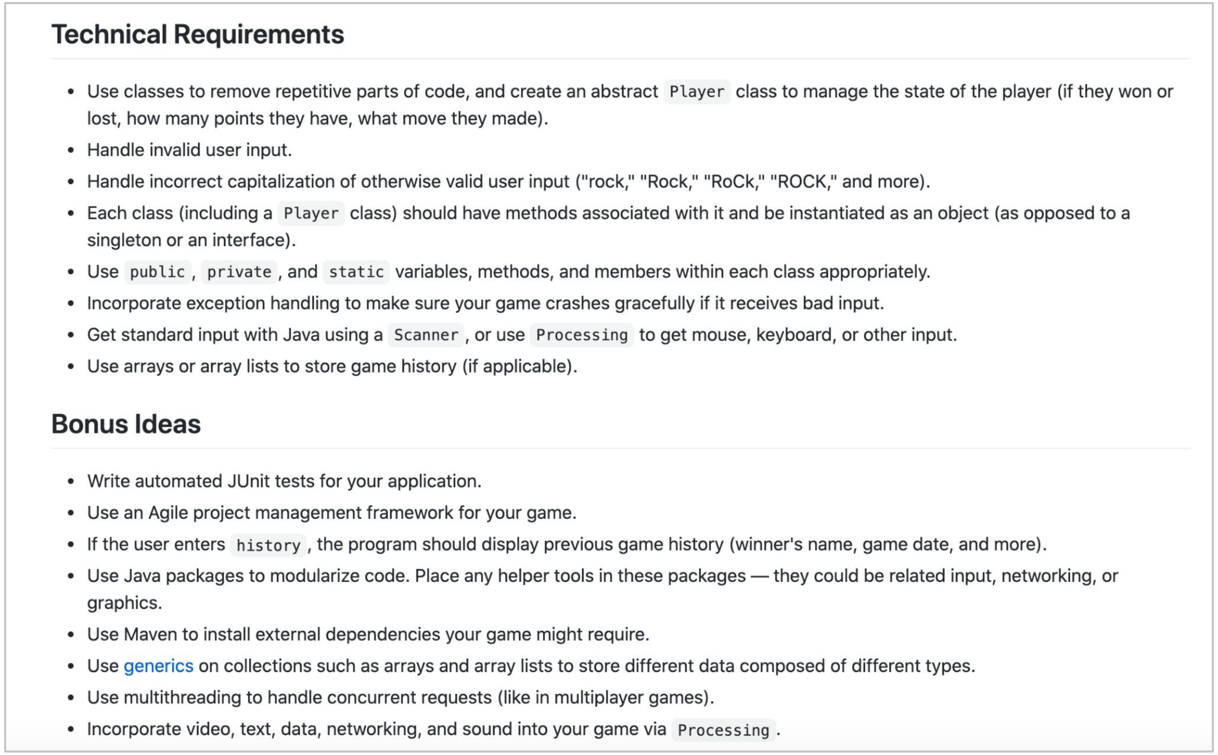
It's time to put students' Java knowledge to use. They will create a basic version of rock–paper–scissors that allows users to play against the computer in the console. The game consists of a few main features:

- Play rock–paper–scissors against a computer player.
- Play rock–paper–scissors against a human player.

The game must:

- Have a main menu with options to enter two players or vs. computer.
- If the user enters two players, they are able to play against a human.
- If the user enters vs. computer, they are able to play against the computer.
- When the game is over, the winner is declared.

Creating a Game in Java



Technical Requirements

- Use classes to remove repetitive parts of code, and create an abstract `Player` class to manage the state of the player (if they won or lost, how many points they have, what move they made).
- Handle invalid user input.
- Handle incorrect capitalization of otherwise valid user input ("rock," "Rock," "RoCk," "ROCK," and more).
- Each class (including a `Player` class) should have methods associated with it and be instantiated as an object (as opposed to a singleton or an interface).
- Use `public`, `private`, and `static` variables, methods, and members within each class appropriately.
- Incorporate exception handling to make sure your game crashes gracefully if it receives bad input.
- Get standard input with Java using a `Scanner`, or use `Processing` to get mouse, keyboard, or other input.
- Use arrays or array lists to store game history (if applicable).

Bonus Ideas

- Write automated JUnit tests for your application.
- Use an Agile project management framework for your game.
- If the user enters `history`, the program should display previous game history (winner's name, game date, and more).
- Use Java packages to modularize code. Place any helper tools in these packages — they could be related input, networking, or graphics.
- Use Maven to install external dependencies your game might require.
- Use generics on collections such as arrays and array lists to store different data composed of different types.
- Use multithreading to handle concurrent requests (like in multiplayer games).
- Incorporate video, text, data, networking, and sound into your game via `Processing`.