Week 1 - Outline

Monday	Tuesday	Wednesday	Thursday	Friday
• Morning ○ Workbook 1 - a - From-10000-Feet ○ Exercise - Decimal/Binary Conversions • Afternoon ○ Workbook 1 - b - CLI and GIT ○ Module 1 - Working with CLI and GitBash ○ Module 2 - Working with git and GitHub	• Morning ○ Workbook 1 - c - Working with IntelliJ ○ Workbook 1 - Intro to Java ▪ Module 1: Introduction to Programming • Afternoon ○ Workbook 1 - Intro to Java ▪ Module 1: Introduction to Programming ○ Workbook 1 - Intro to Java ▪ Module 2: Java Syntax and Data Types	• Morning ○ Workbook 1 - Intro to Java: Module 3 ▪ Java Syntax ▪ Operators and Expressions • Afternoon ○ Workbook 1 - Intro to Java: Module 3 ▪ Printing to Screen ▪ Getting User Input ▪ Working with Methods ▪	• Morning ○ Workbook 1 - Intro to Java: Module 4 ▪ Conditions (if/else) • Afternoon ○ Workbook 1 - Intro to Java: Module 4 ▪ The switch statement ○ Workshop prep	• Workshop Project ○ Console application to solidify this weeks concepts

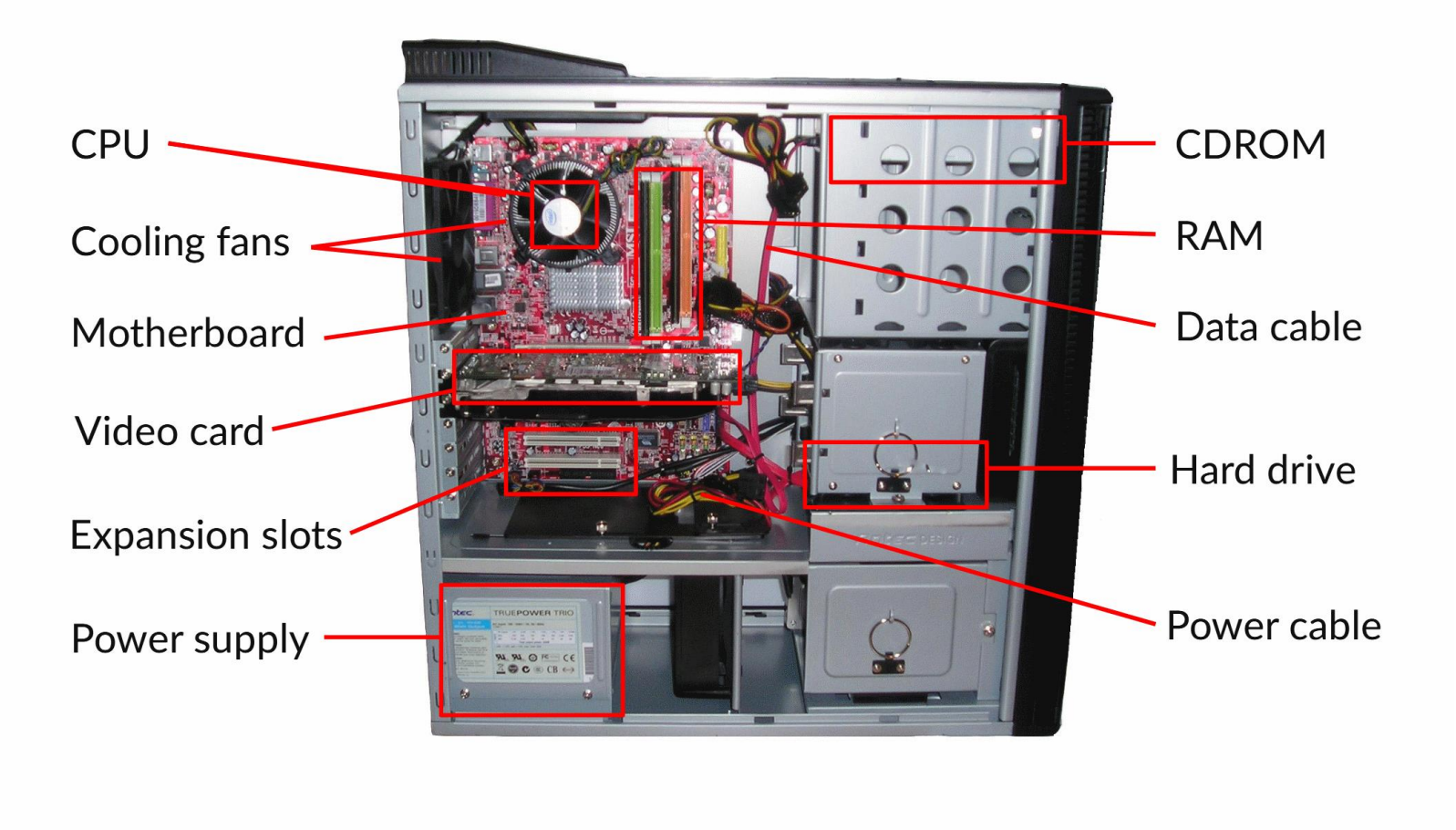
Module 1

Computers and the Internet (from 10k feet)

Section 1–1

Introduction to Computers

What's Inside a Computer?



- **The *motherboard* is the big circuit board that holds all the other parts together**
  - It provides fast connectivity between the major components within the computer

CPU and RAM

- **Connected to the motherboard is a *CPU (central processing unit)***
  - Also called "the processor", it is "the brain" of the computer
  - It executes instructions that a programmer wrote to control the processing of data
  - It gets HOT when running, and is usually attached under a fan and a heat sink!
  - Most modern CPUs have more than one processor core (dual-core, quad-core, etc)

- **Also connected to the motherboard is *RAM (random access memory)***
  - RAM is very fast memory that holds instructions and data *while the computer is running*
    - It is *volatile storage* and does NOT provide long-term storage and files; if you turn the computer off, it's erased
  - Personal computers often have 8GB or 16GB of RAM, although there may be more or less

Persistent Storage

- ***Persistent* (or *non-volatile*) storage holds data even when the computer is turned off**

- ***Disk drives* provide non-volatile storage for files**
  - May be very large (125GB - 2TB or more!) and inexpensive, though slower than RAM
  - There are two basic types of disk drives
    - Hard disk drives (HDD) store data on a magnetically coated platter that spins at a high speed
    - Solid state drives (SSD) store data on non-volatile semiconductor chips that have no moving parts

- **Computers often provide some type of removable storage device as well, including:**
  - USB connections for reading/writing to flash drives
  - Optical disc drives for reading/writing to DVDs or CD-ROMs
  - Floppy disk drives for reading/writing to ancient floppy disks

Expansion Cards

- **Other important "cards" that plug into expansion slots on the motherboard may include:**
  - a NIC card (network interface card) that is an adapter that allows the computer to connect to a network
  - a wireless NIC card that lets a computer use radio signals to connect to a network wirelessly
  - a video or graphics card that is an adapter to let the computer output display on a monitor
  - a sound card to let the computer play sound from speakers

- **Finally, there are power-related items within the case**
  - a power supply
  - a heat sink and one or more fans to help dissipate the heat

How do you Connect Peripherals to a Computer?

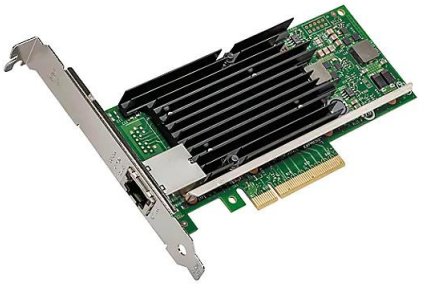
- **Connectors provide a way to connect a peripheral device to your computer**





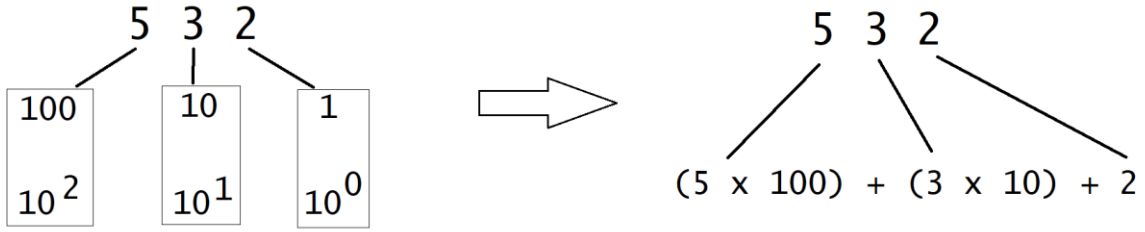
- **The main ones used today include:**
  - USB provides a hot-swappable connection for data transfer (includes USB 2.0, USB 3.0, USB-C)
  - HDMI carries high-speed video and sound
  - Firewire (mostly used by Apple)
  - PS/2 (older) often used for keyboards and mice
  - VGA and DVI used for video signals for monitors

Question - Identify These!



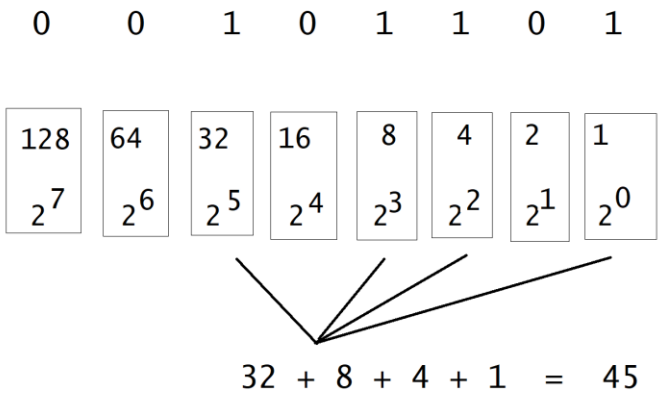
Counting Like a Computer

- **When we count as human beings, we use numbers that are powers of 10**
  - Every digit can be one of 10 different values 0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 (decimal numbers)



- **But we measure quantity in the computing world with numbers that are powers of 2**
  - Every digit can be one of 2 different values 0 - 1 (binary numbers)
  - Each *binary digit* (**bit**) represents an on / off state

What does the number 45 look like in binary?



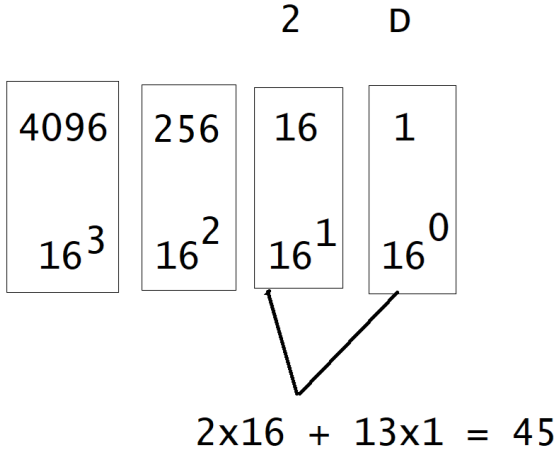
Word Sizes

- **When you bought a computer, do you remember if the marketing material said it was a 32-bit or 64-bit CPU processor?**
  - A 32-bit computer has a "word size" of 32 bits
    - This means the biggest signed whole number that it can fit in a word is 2,147,483,647 which is determined by the binary number 01111111111111111111111111111111
    - Note: Why does the binary number above start with a 0? Signed binary numbers uses the high-order bit to indicate the number is negative!
  - A 64-bit computer has a "word size" of 64 bits
    - This means the biggest signed whole number that it can fit in a word is 9,223,372,036,854,775,807

Hexadecimal (Base 16) numbers

- **Because base 2 numbers can consist of many, many digits… programmers often use another numbering system called hexadecimal (base 16)**
  - Each digit can be one of 16 different values: 0 - 9 and A - F (where A = 10, B = 11, etc)

What does the number 45 look like in hexadecimal?



- **You will rarely see binary as a programmer, but a web programmer often uses hex numbers for color (RGB) values**
  - We will see examples soon
  - For example, the color red can be represented by the hex number FF0000 in CSS; the first two digits are a hex value for red, the next two are for green, and the last 2 are for blue

Exercise - How Old Are You?

- **Let's figure out how old you are using alternate bases…**
- **in Base 2?**

```
___  ___  ___  ___  ___  ___  ___  ___
128   64   32   16    8    4    2    1
```

digits are 0 - 1

- **in Base 10 (this should be easy)?**

```
___  ___  ___
100   10    1
```

digits are 0 - 9

- **in Base 16?**

```
___  ___  ___
256   64    1
```

digits are 0 - 9 and A - F (representing 10 - 15)

What does KB, MB, GB and TB Actually Mean?

- **The smallest "unit" of memory you can access is 8 bits… which is called a** *byte*

- **Programmers use the term KB for** ***kilobyte***
  - When describing file sizes, a kilobyte is 1024 bytes

- **Not 1000?**
  - The International System of Units defines 1K as 1000
  - Software makers use base 2 numbers and define 1K as 1024

- **Usually, programmers use the term MB for megabyte…. 1024 units of 1KB**
  - That means 1024 x 1024, or 1,048,576 bytes

UNIT	SIZE	BYTES
1 Bit	Single 0 or 1 binary digit	--
1 Byte	8 bits	1 byte
1 Kilobyte (KB)	1024 bytes	1024 bytes
1 Megabyte (MB)	1024 KB	1,048,576 bytes
1 Gigabyte (GB)	1024 MB	1,073,741,824 bytes
1 Terabyte (TB)	1024 GB	1,099,511,627,776 bytes

Storage Capacity and Network Speeds

- **It gets more confusing when you introduce hardware**
  - According to software manufacturers, 1 KB = 1024 bytes
    - Which means 1 TB = 1024 * 1024 * 1024 * 1024 = 1,099,511,627,776 bytes.
  - But hard disk makers don't think average consumers can understand this issue about "kilo" so they use 1 KB = 1000 bytes and 1 TB = 1,000,000,000,000 bytes
  - Either way you look at it, 1 TB is a lot of capacity!

- **When we specify internet speeds, we specify in *bits per second***
  - What download speed do you get from your internet provider?
  - Today I got 222 **Mbps** (megabits per second)
    - That is 232,783,872 **bits** per second or 29,097,984 **bytes** per second

Which Holds More Data?

Floppy disk - 1.44 MB

USB thumb drives … It depends on the size you buy! 8GB? 256GB?

Internal HDD - 10TB drive

Exercise – Decimal to Binary

- **How would you write the following Base10 (Decimal) numbers in Binary**
  - 15

    ——————————
  - 32

    ——————————
  - 117

    ——————————
  - 946

    —————————— ——————————

Exercise – Binary to Decimal

- **How would you write the following Binary numbers in Decimal (Base10)**
  - 00000100

    ——————————
  - 00001101

    ——————————
  - 00110101

    ——————————
  - 01010101 10101010

    —————————— ——————————

Learn to Code

Command Line and Git

Workbook 1-b

Version 6.0 Y

Table of Contents

Module 1

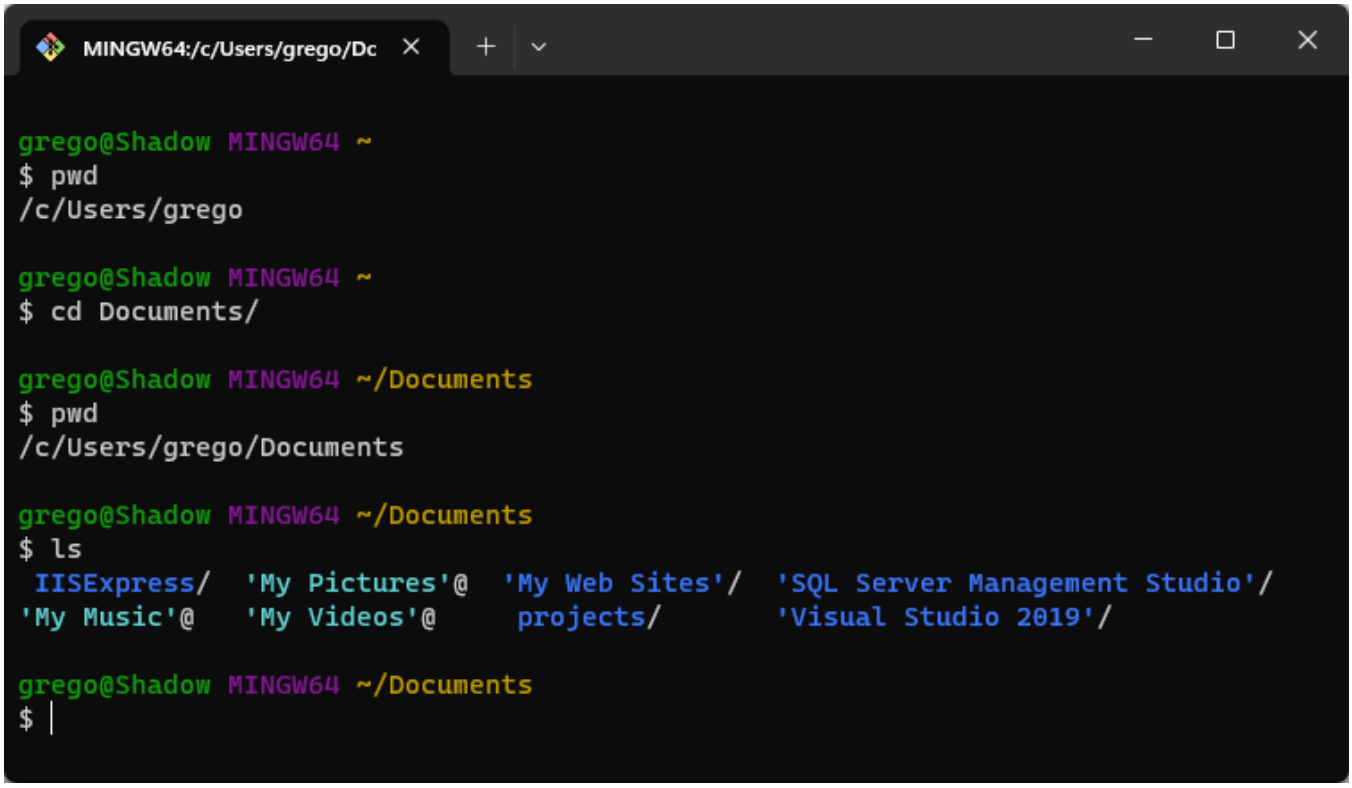
The Command Line

Section 1–1

The Command Line

The Command Line

- **The command line is a powerful, text-based interface that helps developers run programs and control the computer**
  - Before operating systems had sophisticated user interfaces like those in Windows or macOS, the command line was all we had
    - Think of old-school DOS and early Linux systems



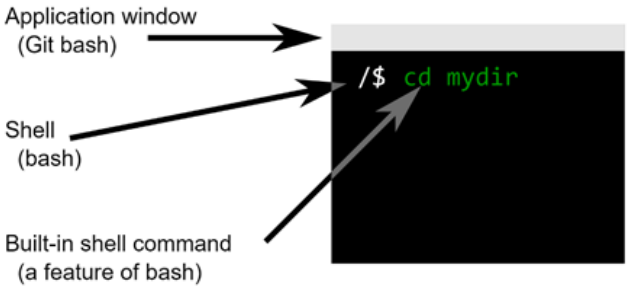
- **There are *still* things you can do from the command line that you *can't* do with point-and-click user interfaces**
  - As a programmer, it is a skill that you eventually *must* conquer
  - It takes a while to get used to, but with practice you catch on

CLIs and IDEs

- **If you watch old sci-fi from the ’80s or a modern show with hackers working, you see:**
  - a command line presented as a black window with white text that scrolls by quickly
  - computer experts typing cryptic commands
- **The program that processes these cryptic commands is called a *Command-Line Interface (CLI)***
- **If you are a programmer in the 2020s, you have more friendly software development tools to get the job done**
- **Powerful tools like Visual Studio Code (Web programming), Visual Studio (C#/.NET), and IntelliJ (Java) allow programmers to write and run their code using a single tool**
  - These tools are called Integrated Development Environments (IDE)
- **If an IDE doesn’t allow us to perform the task we need to, or it doesn’t make it easy, we can leave the IDE and use the CLI**

Anatomy of a Command Line

- **A program called the *shell* processes the commands we enter in the command line interface**
  - The Windows operating system has two native command shells: The Command Shell (`cmd`) and PowerShell
  - Plus, we've installed another ***different command shell*** (using Git Bash) named `bash`



- **We can type the name of a *command* (or program) to run it in the terminal window**

- **The `date` command prints out the current date**

Example

```
$ date
Thu Apr  6 21:22:08 EDT 2023
```

Section 1–2

Built-In Shell Commands for Files and Directories

The File System

- **In persistent storage devices, a chunk of related data is stored in a *file***
  - A file holds bytes
  - Has a specific location on the drive and a size (in bytes)
    - May be large or small (even 0 bytes), depending on contents
  - Has a name (hopefully, one that makes sense to a person)

- **A file usually represents something important to the user**
  - PDF document
  - Photograph
  - Animated GIF
  - Address book
  - A Java source file (!)

- **Some files store parts of the operating system itself**
  - Without them, your computer is a brick!

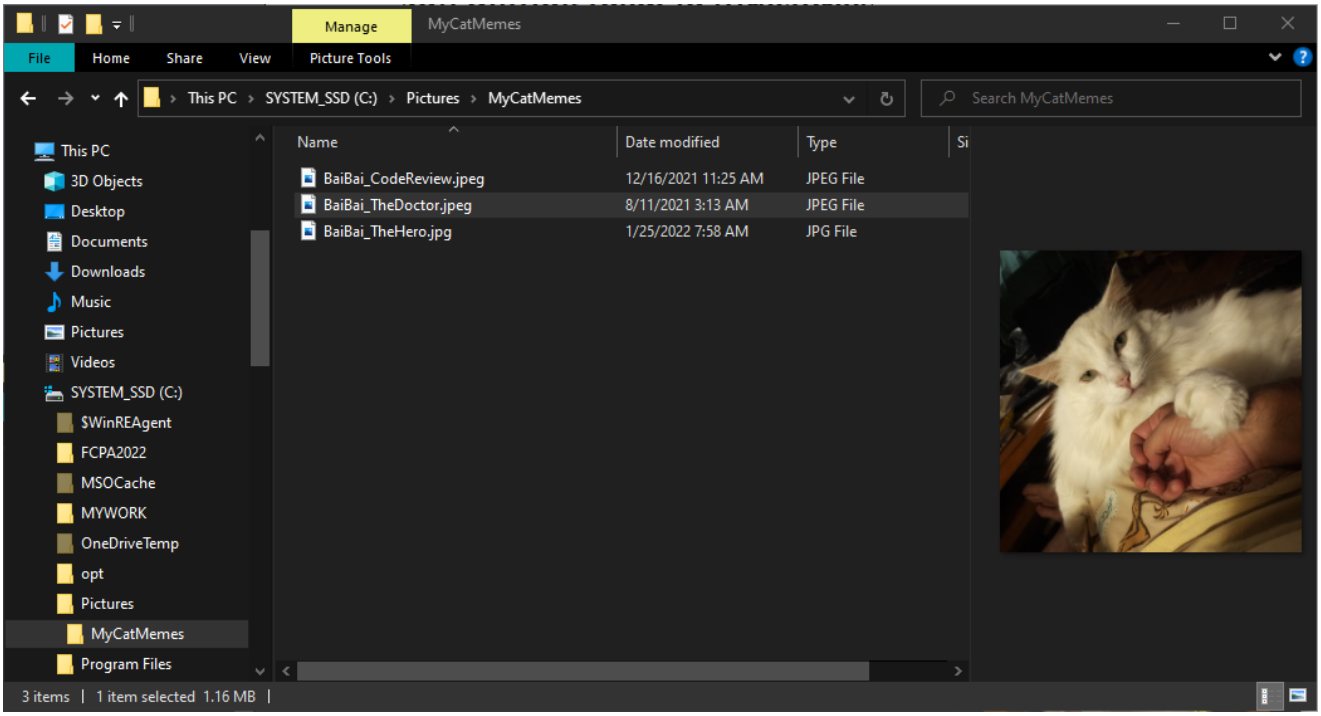
- ***We would hate to lose any of our files accidentally, so two things are really important:***
  - Organizing files so you can find them when you want them
  - Backing them up to more than one persistent storage location

Directories (Folders)

- **A set of related files is indexed by a *directory***
  - A directory is just a special kind of file that is only used to organize other files
- **Logically, a *directory* holds files, so you can also call it a *folder***
  - Has a name (again, hopefully, one that makes sense to a human)
  - Has zero or more files called "children"
  - Usually, a child is an "ordinary" file full of bytes, however...
- **A directory can be a child of another directory, called its "parent"**
  - Directories can organize other directories, in a family tree (called a *directory tree*)
  - A child directory is called a *subdirectory* or a *subfolder*
  - Only a "root" directory (e.g. the folder named "/") has no parent
- **A root directory, together with all of its children, grandchildren, great-grandchildren, etc) is called a *File System***

File Names

- **We organize our files by giving them (meaningful) names, and putting them in directories**



- **When you use the File Explorer, it shows you a visual representation of the file system**

  – That's nice, but...

- **When you use the CLI, your commands may affect files in your current working directory, but you just have to remember where you are in the file system (!)**

`pwd` - Where am I? (print working directory)

- As a CLI programmer, you learn to build a visual tree in your mind of the file system

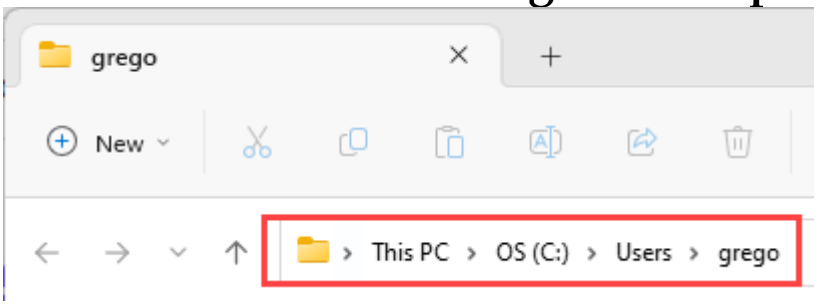
- At any point, if you wonder, "Where am I?", you can enter the `pwd` command

Example

Find out what directory you are in

```
$ pwd
/c/Users/grego
$ cd documents
$ pwd
/c/Users/grego/Documents
```

- This is similar to looking at the explorer navigation bar



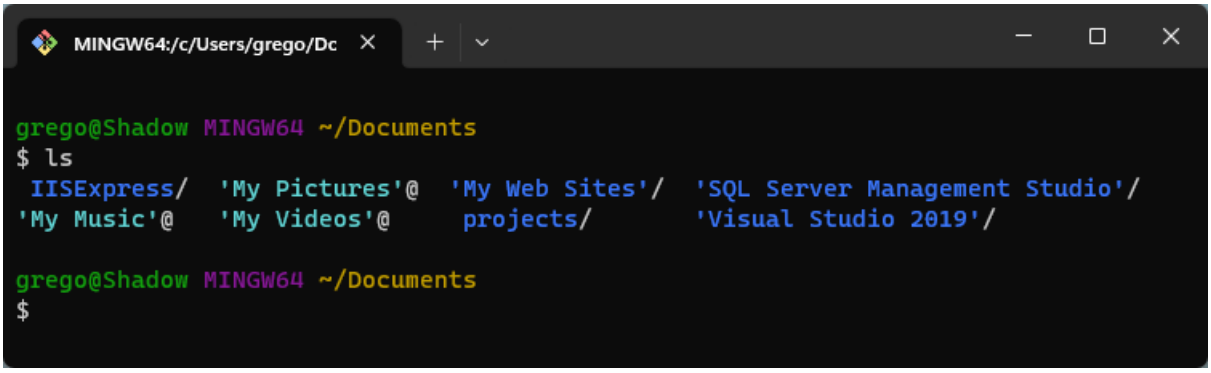
- We will explore the `cd` command soon

`ls` - List Files and Subdirectories

- **To list the files and directories of the current working directory, use the ls command**

Example

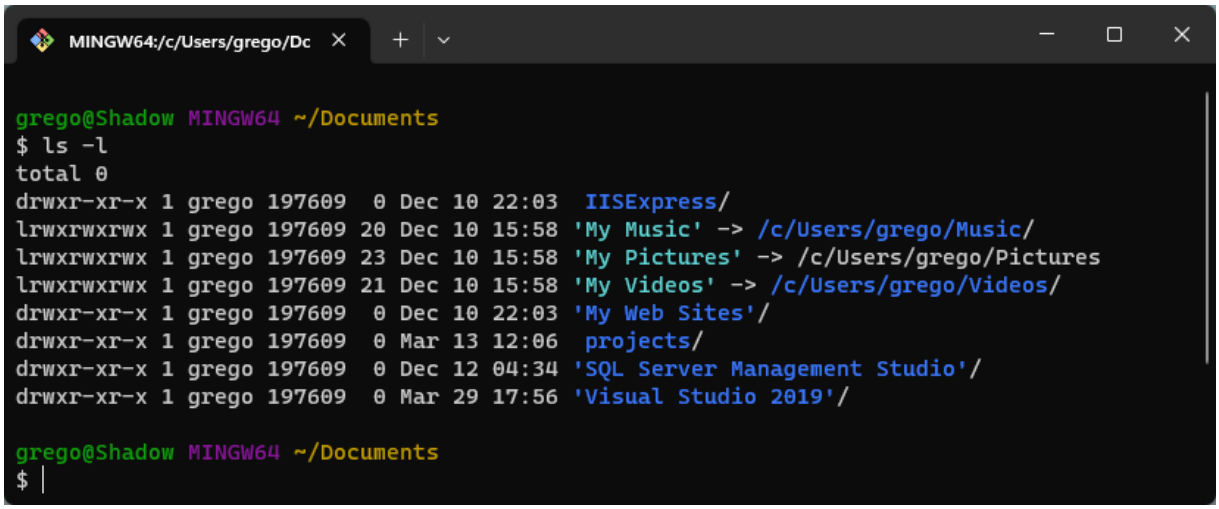
List files and subdirectories (doesn't show hidden files/directories)



- **If you want more information, specify the `-l` option (the lower case letter L -- not a 1)**

Example

List details of files and subdirectories (doesn't show hidden files/directories)



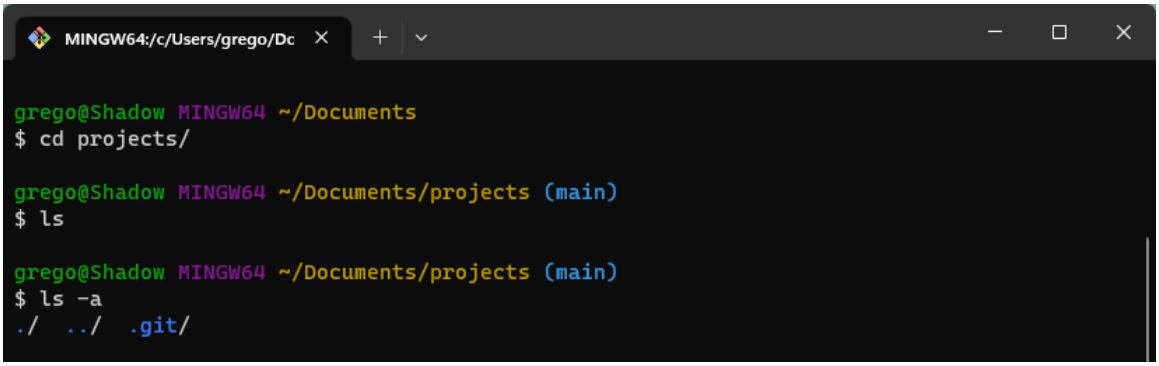
"Hidden" Files

- **In many shells, file names that begin with a period "." are *hidden* and won't appear in `ls` listings**

- **If you want to see *all* files, add the `-a` option (the 'all' flag)**

Example

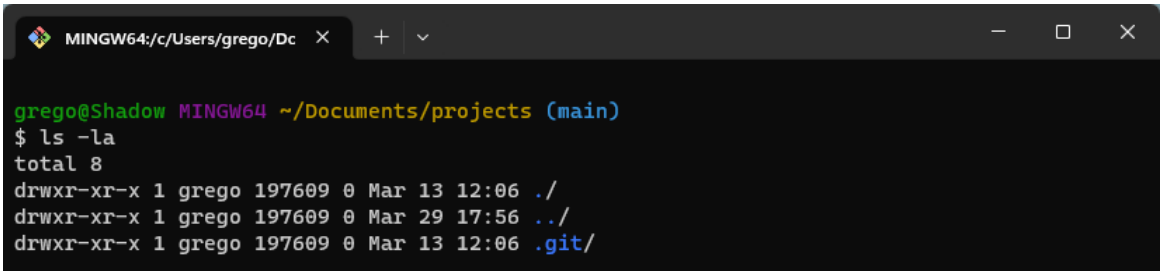
List all files and subdirectories. In the `projects` folder, the `ls` command shows no files, but the `ls -a` command shows a hidden `.git` folder



- **You can combine options to see all of the files and their details**
  - You might see this written as `-la` or `-al`

Example

List details of all files and subdirectories



`cd` - Change the Working Directory

- **In the following examples, assume you have the following directory tree:**

```
projects/            <== You start out here, in the projects directory
└── projects.txt
└── kitchen-remodel/
|   └── description.txt
|   └── budget/
|         └── remodel-budget.csv
└── photography/
    └── food/
    |   └── apples.jpg
    └── nature/
        └── trees.jpg
```

- **To change your current directory, use `cd`**

Example

Go down into the `photography` directory from the project directory

```
$ cd photography
```

- **You can change directories one at a time, or combine directory names into a single *relative path***

Example

Go down into the `photography/nature` directory in two steps:

```
$ cd photography
$ cd nature
```

or, do it all at once

```
$ cd photography/nature
```

Going up to the Parent Directory

- **To change the directory up one level, use `cd ..`**

  Example

  Go up one level in the directory tree

```
$ cd ..
```

  NOTE: `..` refers to the parent directory and `.` refers to the current directory

- **If you are in the directory `photography/nature` and want to go back to the `food` directory, you can use a path**

  Example

  Go up one level in the directory tree and then down into another subdirectory

```
$ cd ../food
```

- **If you are in the `nature` directory and want to go back to the top level of the project, use this**

  Example

  Go up two levels in the directory tree

```
$ cd ../..
```

Go Home!

- **Your "home" directory is the place where your user's files are stored**
  - On modern Windows, that's normally at the path `/c/Users/«your user name»`
    * For example, `/c/Users/grego`

- **You can use the bash shortcut tilde `~` to cd to your "home" directory**

  Example

  Go to your home directory

```
$ cd ~
```

- **Or you could just type `cd`**

  Example

  Go to your home directory

```
$ cd
```

Exercise

In this first exercise you will create the folder structure that you will use to do your work throughout this coding academy. Each week you will receive a new workbook which will contain your lab exercises. All of your exercises should be organized into the appropriate workbook and module folder. For example all of the work in these exercises should go into the `pluralsight/` folder.

```
C:/
└── pluralsight/
    └── command-line/
        └── VirtualWorld/
```

EXERCISE 1

Create a directory at the root of drive `C:\` named `pluralsight`. You will do your work in this directory, unless otherwise specified.

Add a new directory named `command-line` and unzip the `VirtualWorld.zip` file to that directory. After unzipping your folder structure should match the diagram above.

Note: We will use the words *directory* and *folder* interchangeably throughout this course; they mean the same thing.

DO NOT OPEN THE `VirtualWorld` DIRECTORY USING FILE EXPLORER!

Launch Git Bash. Use a **`cd`** command to navigate to `C:\pluralsight\command-line\VirtualWorld`. Use the **`pwd`** command to make sure you are in the correct directory.

Using the **`ls`** and **`cd`** commands, answer the following questions:

1. What are the names of the subfolders under `VirtualWorld`?
2. What types of food are there in this virtual world? Hint: find the Foods folder and look inside of it.
3. What kinds of pets are there? Hint: find the Pets folder and look inside of it.
4. What parks are in California?
5. What kinds of BBQ are there?
6. How many different farm animals are there?
7. What Mexican foods are there?
8. What parks are in Texas?

Now, go back to your HOME directory. What files and folders are located there?

Section 1–3

Creating and Deleting Files and Directories

`mkdir` - Create a Directory

- **You can create a directory using the `mkdir` command, followed by the name of the directory that you want to create**
  - NOTE: The directory is created as a subdirectory of wherever you are (!)

Example

Create a new directory

```
$ mkdir landscape-yard
```

`touch` - Create a File

- **The `touch` command is the easiest way to create a new, empty file**

  Example

  Create a new file

```
$ touch description.txt
```

  Example

  Create three new files

```
$ touch description.txt budget.csv tasks.csv
```

- **`touch` can also be used to change the timestamps on existing files and directories**
  - A timestamp is the recording of the date/time of the most recent access or modification on a file or directory

  Example

  Update the timestamp on a file if it already exists

```
$ touch description.txt   <----- since the file exists, it updates the timestamp
```

`rm` and `rmdir` - Delete a File or Directory

- **The `rm` command is used to delete one or more files or directories**

  Example

  Remove a single file

```
$ rm test.txt         <---- you can remove any file
```

  Example

  Remove an empty directory

```
$ mkdir test-project
$ rmdir test-project    <---- the directory must be empty for this to work
```

- **The `rm` command has the powerful (*dangerous?*) option `-r`**
  - It's the recursive option that says to delete that directory, any files it contains, any subdirectories it contains, and any files or directories in those subdirectories, all the way down

  Example

  Remove a directory and all of its contents

```
$ rm -r landscape-yard
```

Exercise

EXERCISE 2

Continue exploring the `VirtualWorld`.

Using just the commands **`mkdir, touch, rm`** and **`rmdir`** commands, perform the tasks below:

1. Create a new state (folder) to hold parks in Ohio
2. Add 2 new files named `Adams Lake State Park.txt` and `Caesar Creek State Park.`txt to Ohio
3. Add a new state (folder) to hold parks in New York
4. Add three new parks in New York. Remember, they are just files with the park name and a `.txt` file extension
5. Add an Iguana as a new type of pet
6. Add a new Mexican meal with the name `Enchilada.txt`
7. Add a new state to hold parks in Iowa
8. Add three new parks from Iowa
9. Remove goldfish as a pet
10. Remove all parks in Ohio
11. List the parks in Iowa
12. Remove all parks in Iowa

Section 1–4

Copying and Moving Files and Directories

`cp` - Copy a File or Directory

- **Use the `cp` command to copy files or directories**

  Example

  Copy the file `budget.csv` to a new directory (`Desktop`) in my home directory (~) and rename it to `landscape_budget.csv`

```
$ cp budget.csv ~/Desktop/landscape_budget.csv
```

- **You must use the `-r` flag if you want to include all of the contents of the directory you are copying**

  Example

```
$ cp kitchen-remodel bathroom-remodel
```

  This will create a copy of the `kitchen-remodel` directory but the new `bathroom-remodel` directory will be empty

  Example

```
$ cp kitchen-remodel bathroom-remodel -r
```

  This will create a copy of the `kitchen-remodel` directory and because of the **-r** flag (recursive) all of the contents of the original folder will also be copied to the `bathroom-remodel` directory

`mv` - Move Files and Directories

- **Use the `mv` command to move a file or directory to a new location**
  - In many ways, this is like rename… except that it can also change the location of where the newly renamed file or directory resides

Example

Rename a file

```
$ mv description.txt landscape_project_description.txt
```

Example

Move the file `budget.csv` to a new directory (`Documents` directory) in my home directory (~)

```
$ mv budget.csv ~/Documents
```

`cat` - View the Contents of a File (and other things)

- The `cat` (concatenate) command has several uses, but one of the most common is to display the contents of a file in the terminal window

  Example

  View the contents of a file

```
$ cat HelloWorld.java


public class HelloWorld {

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

}
```

- More examples of the `cat` command can be found here: `https://en.wikipedia.org/wiki/Cat_(Unix)#Examples`

Exercise

EXERCISE 3

Continue working in the `VirtualWorld` directory.

Using just the commands **`cp`** and **`mv`** commands, perform the tasks below:

1. Create a new `American` cuisine in the `Foods` directory by copying the BBQ folder and all of its contents

2. In the `American` cuisine, rename the following files:
   ```
   Brisket.txt  -> Hamburgers.txt
   Ribs.txt     -> Ribeye.txt
   Sausage.txt  -> BLT.txt
   ```

3. Create a new meal by copying the `Hamburgers.txt` file. Name the new file `Fries.txt`

EXERCISE 4

Create a new directory in the `C:/pluralsight/command-line` directory. Name it `first-java-app`.

Using GitBash commands create the following folder structure:

```
first-java-app/
├── src/
│   └── main/
│   │   └── java/
│   │       └── HelloWorld.java
│   └── test/
│       └── java/
│           └── .gitkeep
└── pom.xml
```

1. Using Notepad open `pom.xml` file and add the following text.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.pluralsight</groupId>
    <artifactId>first-java-app</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

</project>
```

2. Close Notepad and using the `cat` command in GitBash, view the contents of the `pom.xml` file.

Module 2

Version Control and Git Basics

Section 2–1

Overview of Git

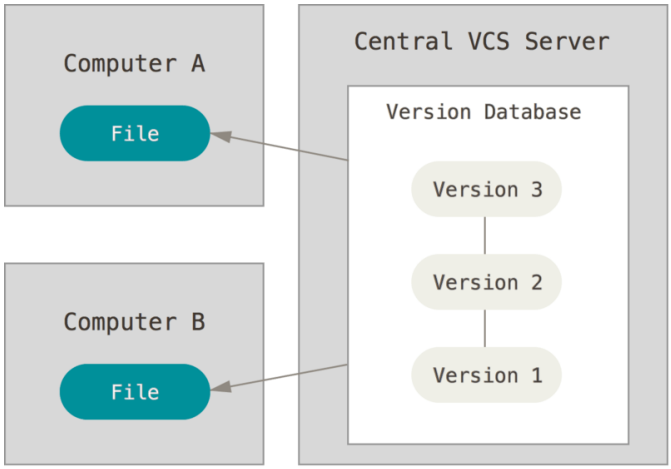
What is Version Control?

- **Version control systems help developers keep track of changes to their source code over time**
  - Commonly referred to as SCM (***Source Code Management***)
- **The idea is to keep track of every modification made to the code base**
- **You "commit" your changes and store them in a repository**
  - Each time you add "something that works" you generally do a commit
  - This might be a several times per hour if you are productive
- **SCM features allow you to roll back to a previous commit if needed**
  - You can also "roll forward" if you've already rolled back
- **Tools also let you compare code in different commits**
  - It can help you figure out what changes might have caused a bug
- **Version control systems can be:**
  - centralized
  - distributed

Centralized Version Control

- **Centralized Version Controls systems store a single "central" copy of the versions of your project on a server somewhere**
  - A popular centralized version control system is SVN (Subversion)

- **Developers "commit" their changes to this centralized version control system**
  - If the server isn't available (no internet access? the server is down?), the developer is not able to save changes to their work within the Version Control System



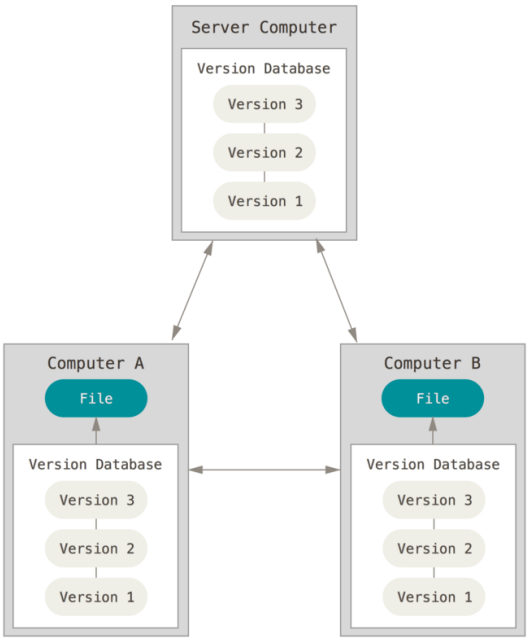
- **Typical Workflow for CVC**
  - Developer checks out code from version control or pulls down the latest changes to code already checked out
    - The file will be locked when someone checks it out, so no one else can check out until it is checked back in
  - Developer makes some changes and tests their work
  - Developer commits changes back to a central repository
    - Once changes are committed, others can pull them into their local copies

Distributed Version Control

- **Distributed Version Control systems do *not* rely on a central server to store all the versions of a project's files**
  - Each developer has a copy of the repository and all of the versions
  - A popular Distributed Version Control system is Git

- **However, cloud based services like GitHub, GitLab and BitBucket can provide a centralized (additional) copy of a repository**
  - To work on a project, each developer could create a repository on their local machine or "clone" an existing repository from the cloud service

- **The developer's local repository would contain the full history of the project**
  - A developer can make changes to the code and commit those changes to their local repository
  - When they are ready to share changes with other developers, they "push" their changes and all the attached history to the remote repository
  - Once pushed, other developers can now update their local copies with those changes.



- **Typical Workflow for DVC**
  - Developer checks out code from version control or pulls down the latest changes to code already checked out
  - Developer makes some changes and tests their work
  - Developer makes a local "commit" that represents the changes made
  - Developer repeats this cycle locally until they are ready to share their changes with other developers
  - Developer commits changes back to the central repository for others to "pull" into their local copies or provide a patch file to another developer if they choose not to use a service like GitHub

Section 2–2

Git Basics and the Local Repository

Git

- **Git is a free, open source distributed version control system**
  - It can handle both small and very large projects
- **It is easy to learn and has a tiny footprint**
  - Most importantly, it is very fast
- **Before you start using Git, install it on your developer machine from** `https://git-scm.com/downloads`
  - Note: Your machine should have been configured before the class started and Git will already be on it
- **To run Git commands, you can use:**
  - Git Bash shell
  - or the Windows command prompt window
- **NOTE: We will use Git Bash throughout session 1**
  - Afterwards, you can use Git Bash or the Command Prompt window -- whichever you prefer

Basic Command : `git`

- **The top-level Git command is `git`**
- **The `git` command is followed by an action you want Git to execute and/or a set of flags**
- **Run the following command to check the version of Git installed**
  - Your version may be different than what is shown here

```
$ git --version
git version 2.8.1
```

Git Setup

- **Git stores configuration information in `~/.gitconfig`**
  - This global configuration contains the settings for all of your repositories

- **Each repository also has a local configuration file named `.git/config` which affects only the repository in which it is located**

- **You can set key/value pairs in a config file using the command `git config`**
  - `user.name` and `user.email` are used when work is committed top keep a history of who made changes

Example

Setting the user's name (global)

```
$ git config --global user.name "Sallie Sheppard"
```

Setting the user's email (global)

```
$ git config --global user.email "ssheppard@myemailprovider.com"
```

  - NOTE: It is possible that these will already be configured on your machine

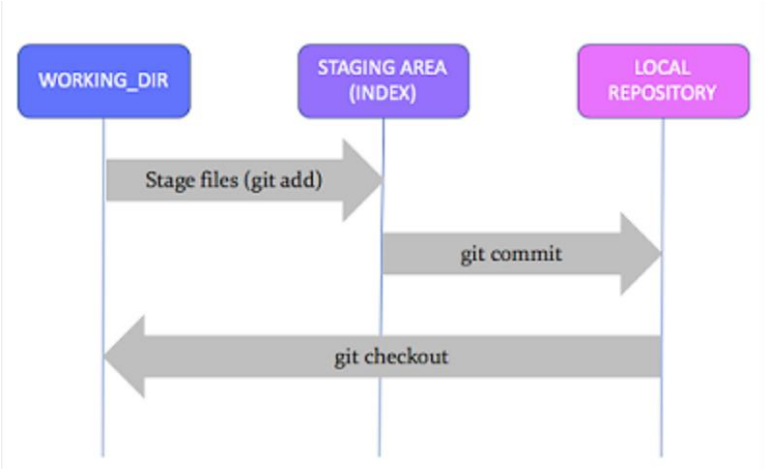
- **To get a list of all current config values, use the `--list` flag**

```
$ git config --list
```

Git Areas

- **There are three core areas where files and folders are maintained within Git:**
  - The *Working Tree* is the folder(s) where you are currently working on your source code
    - Files here are called *untracked* files
  - The *Staging Area* (also known as Index) is where you place things you plan to commit to the repository
    - It allows you to "batch" or "box" a set of changes into one commit
  - The *Local Repository* is a collection of your checkpoints or commits
    - It is the area where everything is saved

- **Learning Git is mostly about learning how and why to move changes from one Git area to another, or learning to query what changes have been committed**



Creating a Repository: `git init`

- **The `git init` command creates a Git repository from the directory you are in when the command is run**
  - Files in the directory can now be tracked by Git

Example

Create a local repo using the current folder

```
$ mkdir Repo1
$ cd Repo1
$ git init
```

- **The `git init` command can create the repository in a subfolder of the current folder if you add the new repo's name after `init`**

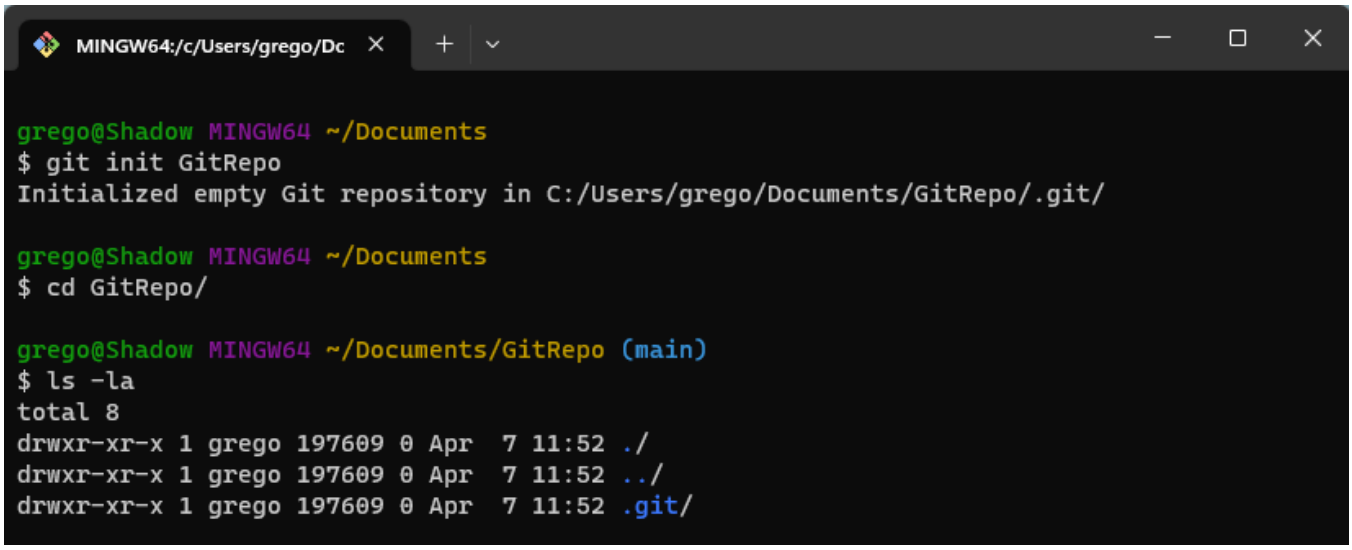
Create a local repo in a specified subfolder

```
$ git init GitRepo
Initialized empty Git repository in /Path/to/repo/GitRepo/.git/
$ cd GitRepo
```

- **When you initialize a repo, git creates the staging area and the local repository in a HIDDEN subfolder named .git**
  - The hidden folder is also referred to as the git internals directory

After Initialization

- You can see the `.git` directory using `ls -la`



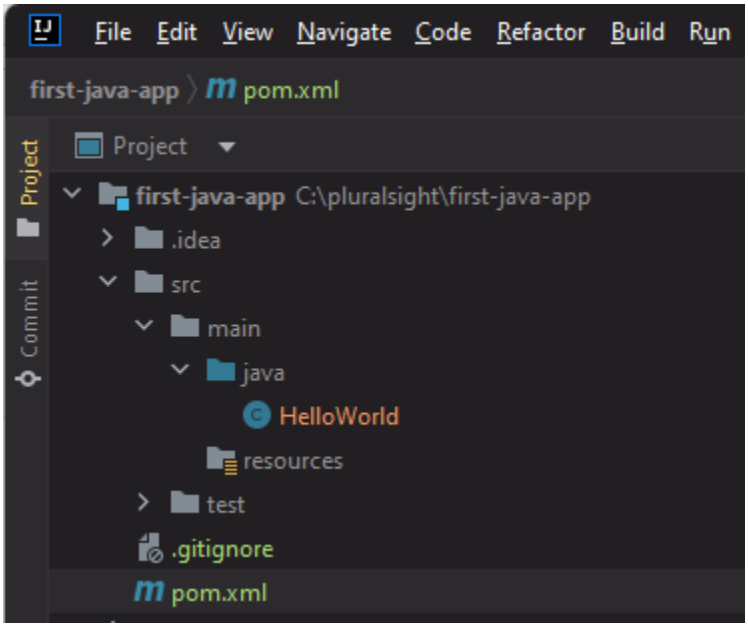
- We won't concern ourselves with the physical structure of `.git` -- suffice to say it holds the staging area and local repository

Working on a Project

- **You can work on a Java project in any text editor**
  - After creating the file structure from the Module 1 exercise we could open this project in IntelliJ

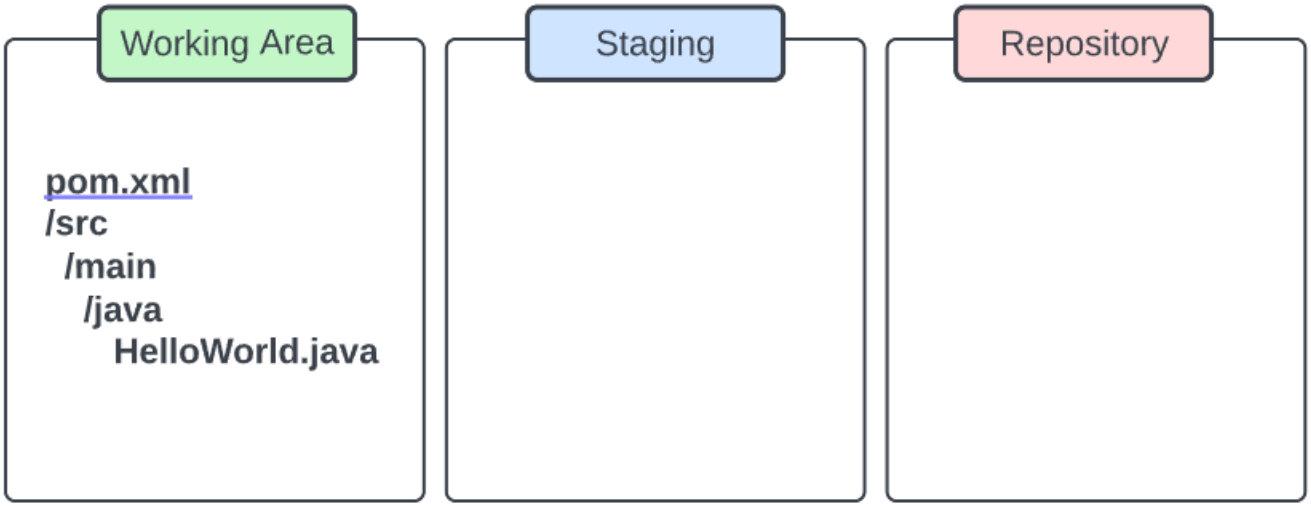
```
first-java-app/
├── src/
│   └── main/
│   │   └── java/
│   │       └── HelloWorld.java
│   └── test/
│       └── java/
│           └── .gitkeep
└── pom.xml
```

  - None of the files have any content in them yet



Working on a Project *cont'd*

- **The folders and files you create in the `GitRepo` folder are in the repository's Working Area**
  - The files are not actually tracked by git until they are *staged* and *committed*



- **Nothing special has happened yet… they are just files in a folder!**

```
MINGW64:/c/pluralsight/com
grego@Shadow MINGW64 /c/pluralsight/command-line/first-java-app (main)
$ ls -la
total 4
drwxr-xr-x 1 grego 197609 0 Sep  5 23:31 ./
drwxr-xr-x 1 grego 197609 0 Sep  5 22:52 ../
drwxr-xr-x 1 grego 197609 0 Sep  5 23:31 .git/
-rw-r--r-- 1 grego 197609 0 Sep  5 22:54 pom.xml
drwxr-xr-x 1 grego 197609 0 Sep  5 22:54 src/
```

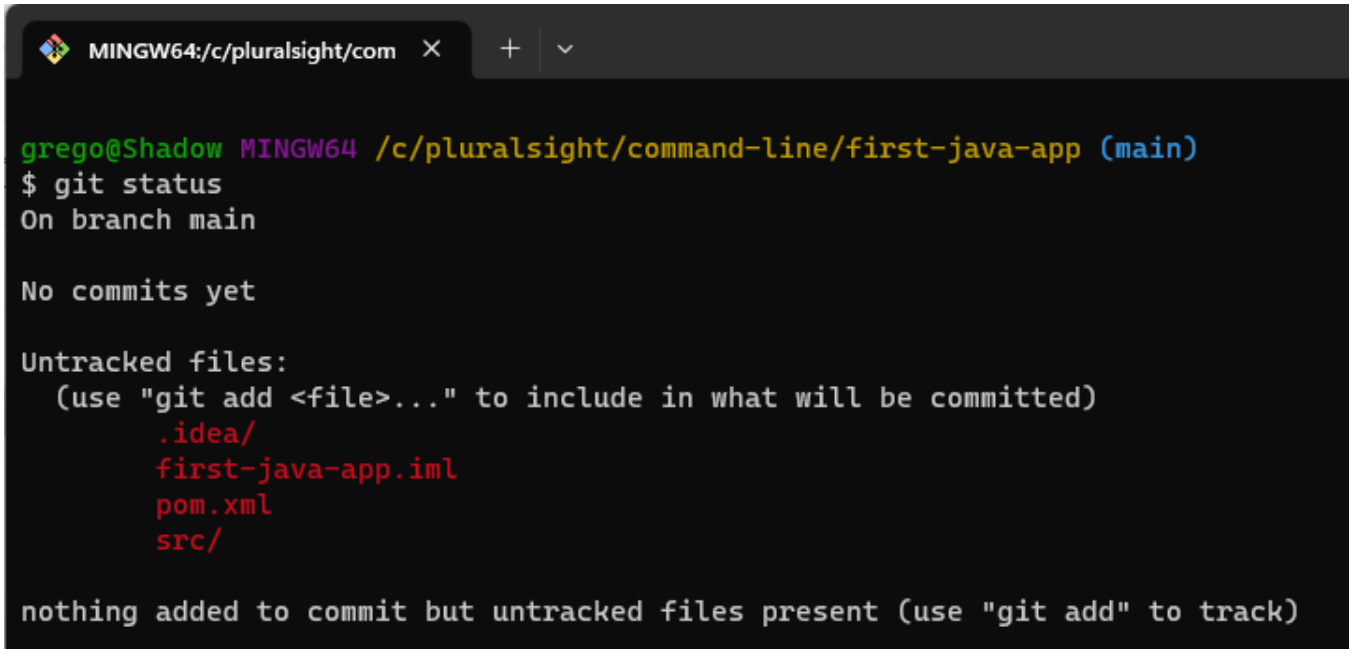
Checking the Status: `git status`

- **The `git status` command displays the state of the working directory and the staging area**
  - It shows you which changes have been staged and which haven't
  - It also shows you which files *aren't* being tracked by Git

Example

```
$ git status
... response varies based on status ...
```

Checking status before staging:



- **The red coloring catches our attention and helps us realize these files/folders aren't tracked**
  - The `.idea/` folder and the `.iml` file were created by IntelliJ when we opened the project

Adding Files to the Staging Area: `git add`

- **When you are ready to commit the current "version" of your code, use the `git add` command followed by a path to one or more files to tell Git to copy them to the staging area**
  - This let's Git know that the files should be ***tracked*** by version control and staged for the next commit

Example

You can stage a single file:

```
$ git add pom.xml
```

You can stage many files using wildcards:

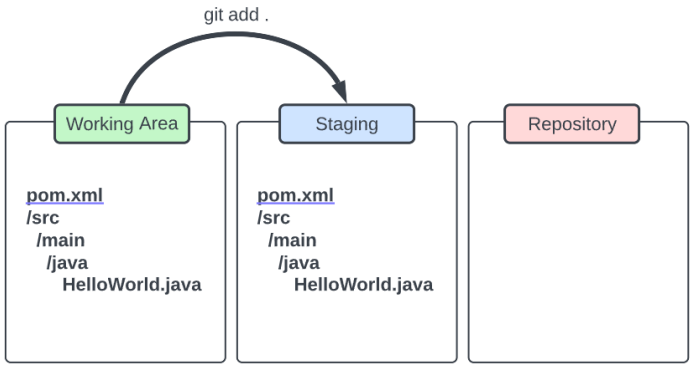
```
$ git add *.java
```

You can stage a whole folder:

```
$ git add src
```

You can stage everything in the Working Directory:

```
$ git add .
```

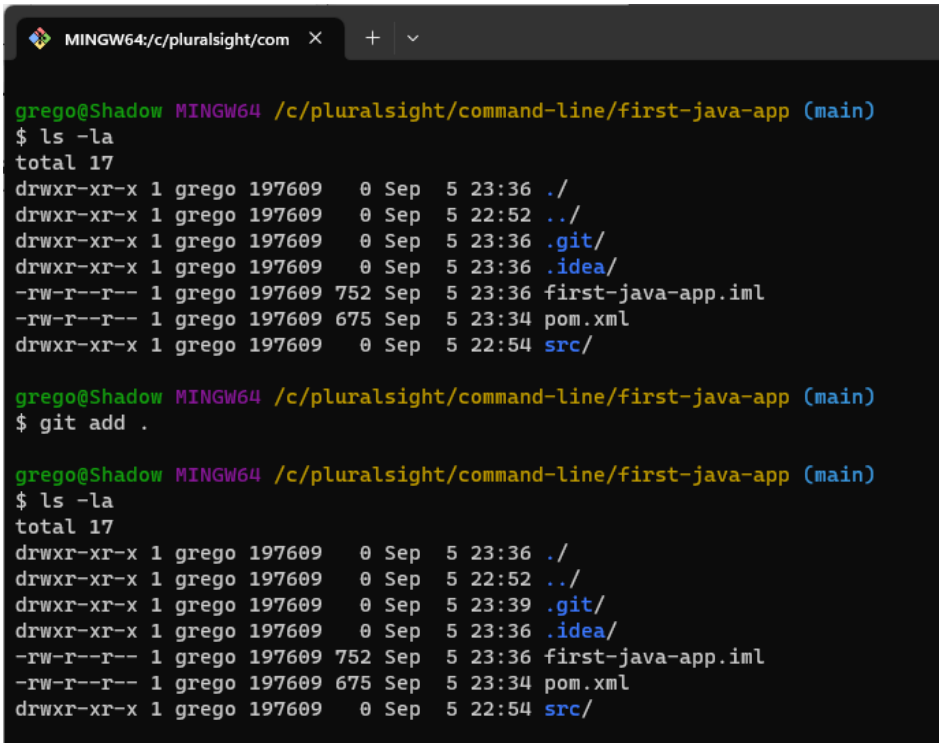


You can stage all changes in the entire repo from any subdirectory:

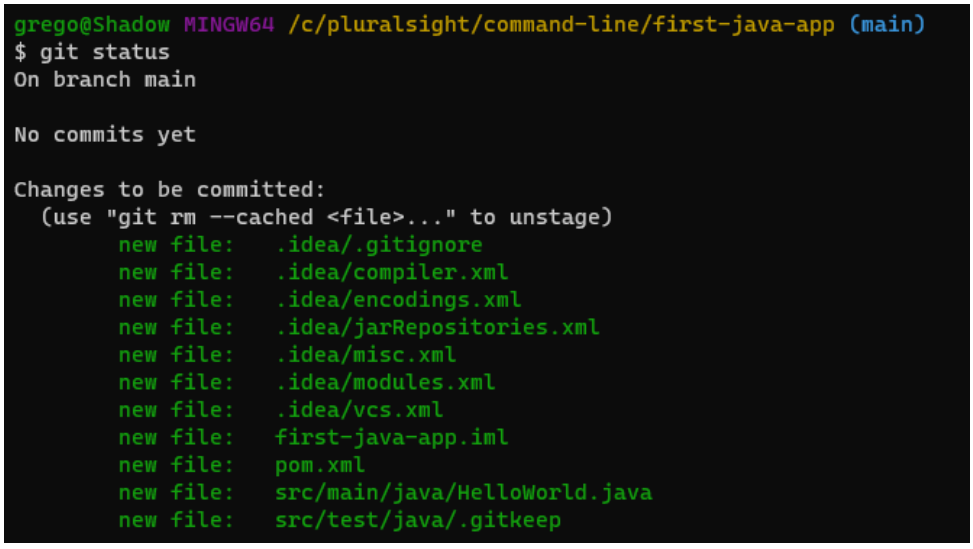
```
## current path c:/pluralsight/command-line/first-java-app/src/test/java
$ git add -A
```

After Staging

- From a simple glance, it doesn't look like `git add` did anything because you don't see any changes in the project's "visible" folder



- However, the `git status` command confirms that the Working Directory has been staged



Placeholder Directories

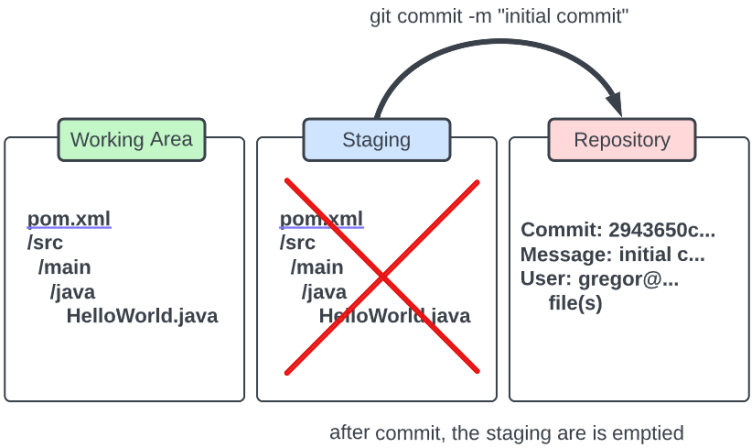
- **Git won't track empty directories**
  - This might be a problem if you want to keep an empty folder (ex: `src/test/java`) where you are going to add files later on

- **The solution to this problem is to place an empty file in the folder that you won't really use**
  - People usually name this file `.gitkeep` or `.keep`

- **By starting the file name with a dot (`.`), most people won't confuse it with a code file in the project**
  - In fact, files that start with a dot are hidden when you list the content of the folder using the `ls` command in most operating systems

Committing Changes to the Repo: `git commit`

- **The `git commit` command takes the staged files and commits them all to version control *as a single transaction***
  - It creates a point in the version control history that can be referenced later
  - The changes can be rolled back as a unit



- **You must supply a short message for the commit**
  - Later, when you look at the project history, this should remind you of what is "inside" the commit, and why the files were changed

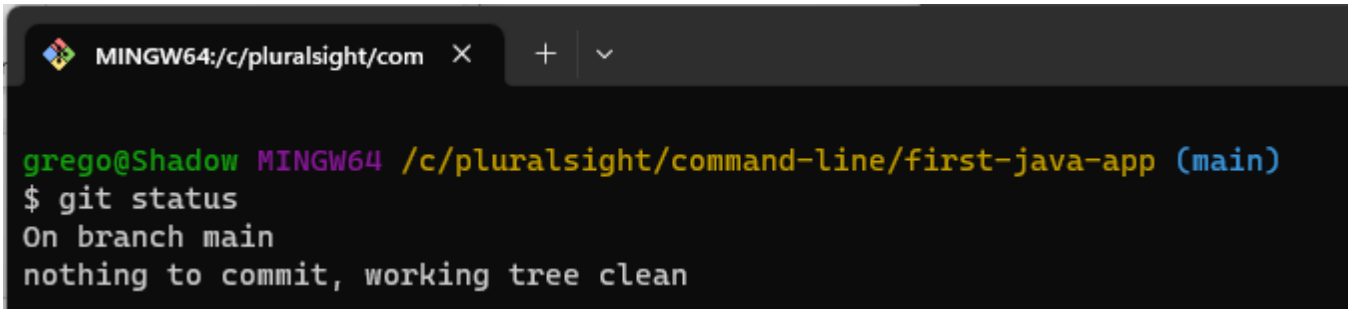
Example

```
$ git commit -m "initial commit"

[main (root-commit) 2943650] initial commit

 3 files changed, 3 insertions(+)
 create mode 100644 pom.xml
 create mode 100644 src/main/java/HelloWorld.java
 create mode 100644 src/test/java/.gitkeep
```

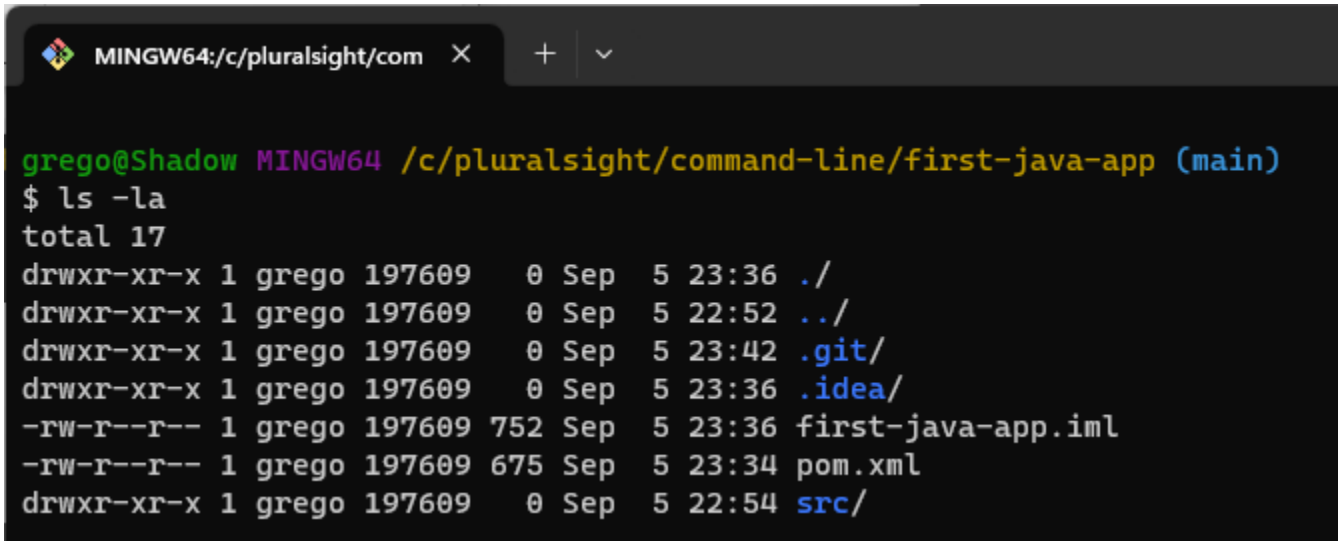
  - Each Git commit is assigned a unique hex number that can be used to find and query the commit later on

After Commit

- After the commit, `git status` shows that there is nothing left in the staging area to commit



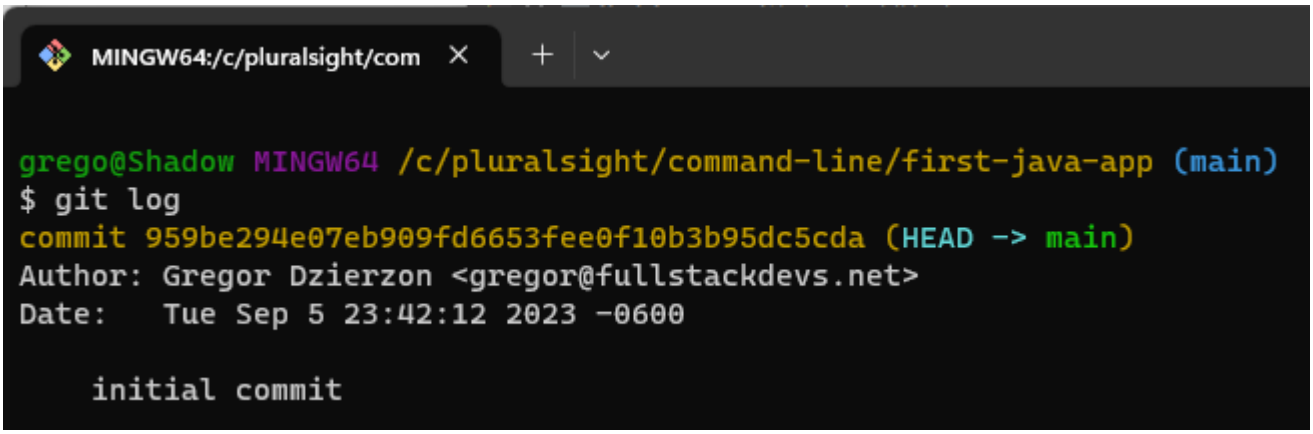
- You can't see the commits unless you look in the hidden `.git` folder



Checking the Commits: `git log`

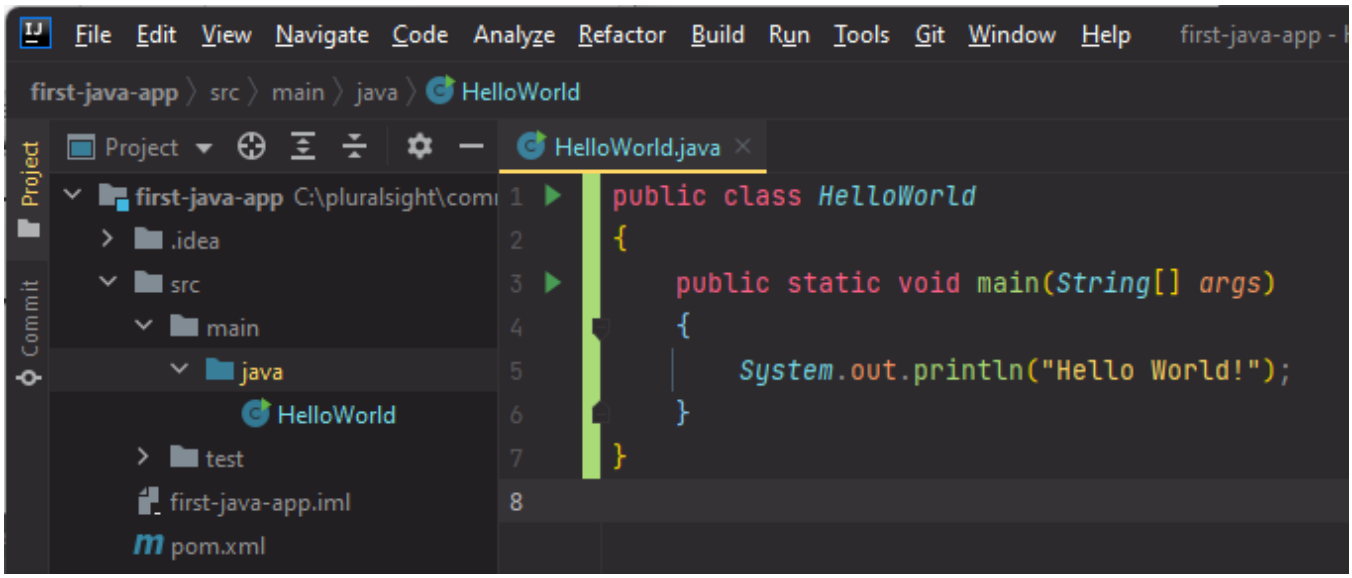
- The `git log` command displays information about the previous commits

```
$ git log
  ... shows a list of previous commits ...
```



Modifying Files in IntelliJ

- **After you make changes to your files in IntelliJ, you must *save*, *stage* and *commit* those changes**
  - We will discuss the actual code changes later



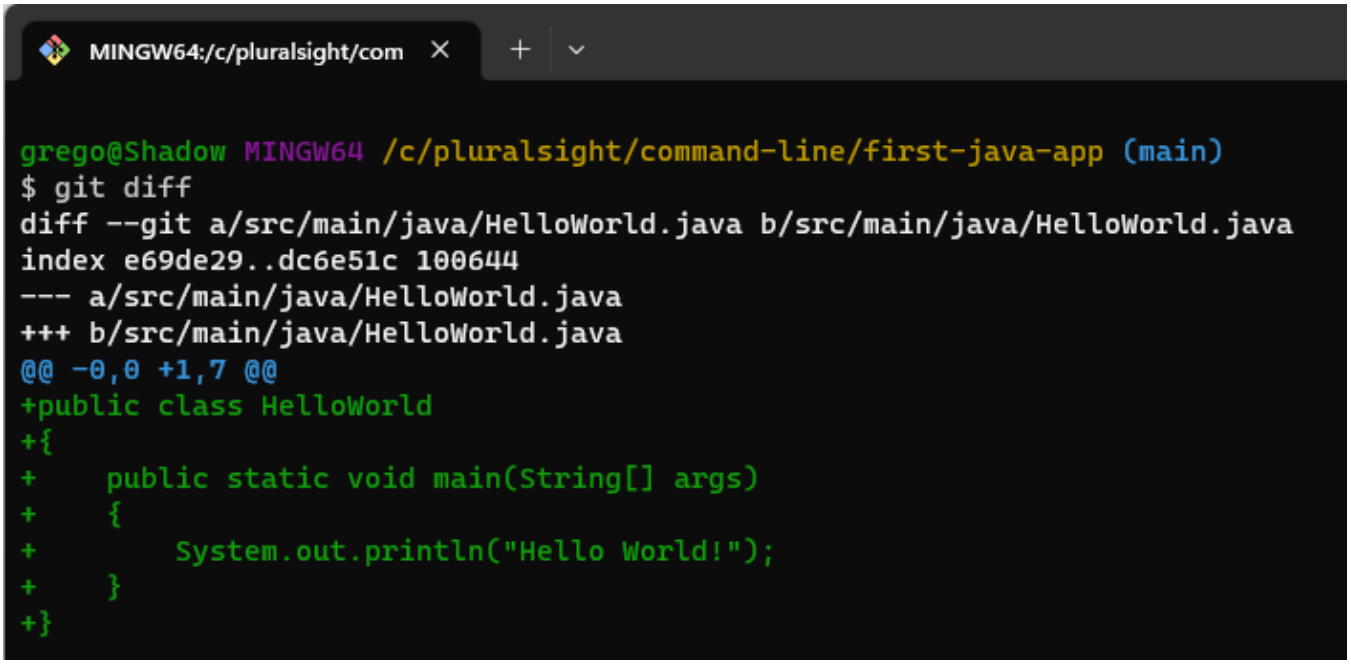
- **Before you add and commit, you may want to browse the changes you made**

Comparing Differences: `git diff`

- **The `git diff` command displays line-by-line differences between files in your working area compared to the last commit**
  - By default, it displays them to the console
  - You can configure a graphical tool to show the differences in an easier to read manner

- **[redacted] changes to the code in our current Working Directory so that it differs from the last commit**
  - We added code to `HelloWorld.java`

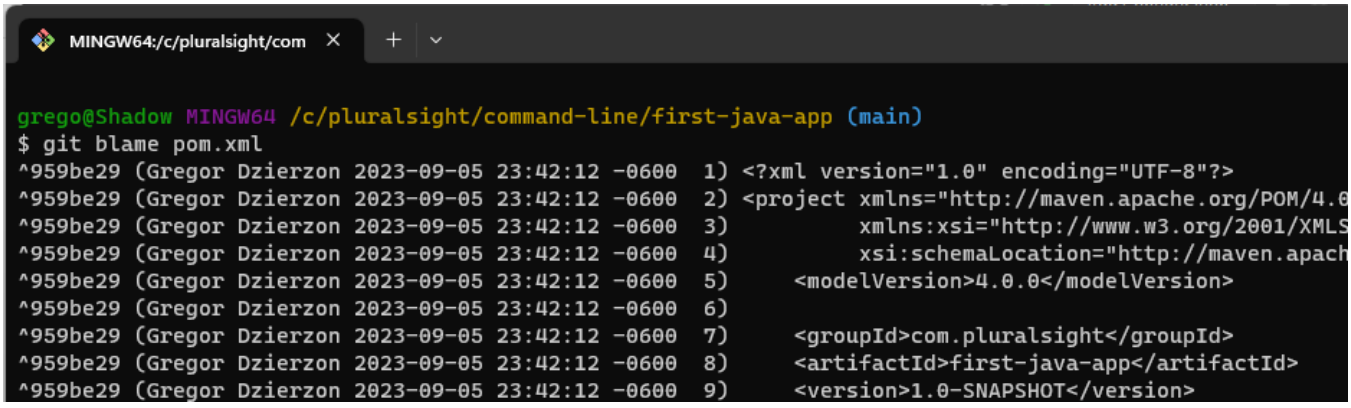
```
$ git diff

  ... shows the difference between files in the working directory
      and the last commit ...
```



Figuring Out Who Made Changes: `git blame`

- **The `git blame` command shows what revision and author are related to each line in a file tracked by version control**
  - While **`blame`** has a negative connotation, this is not intended as a tool to set blame to a developer for changes they made
    - It's just a funny little Linus Torvalds programming joke
  - It is intended to help quickly discover where a change originated, so if you have questions you know who to ask

```
$ git blame pom.xml

... responds with a line-by-line list of pom.xml
```



Ignoring Unimportant Files

- **The really important files in your project are the ones that were built with the hands (and tears) of programmers**
  - These are precious, and would be hard to replace
- **Sometimes, your project has files that you want Git to intentionally ignore**
  - In a JavaScript project, this often means Node.js modules
  - In a Java project, this might mean compiled class files
- **We want Git to ignore these files because we frequently reinstall or rebuild them automatically**
  - Node.js modules are simply ***reinstalled*** using NPM
  - Java class files are always ***rebuilt*** by compiling the project
  - Log files are *generated* each time you run your program
  - These files are cheap and not worth tracking
- **We also want git to ignore large binary files that are hard to compare and are not written directly by programmers**
  - By not tracking these very large files, the commit process and the process of pushing the repository to a cloud service like GitHub or BitBucket is much faster
  - It also makes cloning or pulling from a remote repository faster too!

.gitignore file

- **To ignore files, add a text file named `.gitignore` to the root of project**
  - Within `.gitignore`, identify the files to be ignored
- **`.gitignore` can list individual files, or include wildcards and regular expressions**

```
# Node modules
node_modules/
# Logs
logs/*.log
# Other files
*.pdf
```

- **When you put `.gitignore` at the root, it applies to the whole project**
  - You can also put `.gitignore` files in individual folders for finer grained control over the process
- **To learn more about `.gitignore`, see:**
  `https://git-scm.com/docs/gitignore`
- **To see examples of `.gitignore` files, see:**
  `https://github.com/github/gitignore`

References

- **A cool Git cheat sheet:**
  `https://education.github.com/git-cheat-sheet-education.pdf`

- **Official docs on Git Internals:**
  `https://git-scm.com/book/en/v2`

- **Quick explanation of Git internals:**
  `http://gitready.com/advanced/2009/03/23/whats-inside-your-git-directory.html`

Exercise

In this exercise, you will create a local git repository using common commands and we will point out some quirks like how Git works with empty directories. You will create the repo and commit changes similar to the demos in the preceding pages.

EXERCISE 1

First you need to initialize your `first-java-app` folder as a git repository.

Step 1: Navigate to the `first-java-app` folder. Run `pwd` to confirm you are there (`C:/pluralsight/command-line/first-java-app`).

Run `git init` to initialize your repository

```
$ pwd
$ git init
```

Step 2: Create an initial commit to "save" all of the work that we have done to this point.

NOTE: we will discuss the following commands in more detail in exercise 2

Run `git add .` to stage your work, then run `git commit -m "initial commit"` to commit it

```
$ git add .
$ git commit -m "initial commit"
```

EXERCISE 2

In this exercise you will make changes to the `HelloWorld.java` file and commit the changes..

Step 1: Using Notepad (or IntelliJ) open `C:/pluralsight/command-line/first-java-app/src/main/java/HelloWorld.java` file and add the following code:

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

Step 3: Check the "status" of your repository

Run `git status` and you should see output similar to the following:

```
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   src/main/java/HelloWorld.java

no changes added to commit (use "git add" and/or "git commit -a")
```

Notice that it shows `HelloWorld.java` as a modified file.

Step 6: Stage the files so that they can be committed

Run `git add .` to add the changed files to our staging area to be committed. `git add` is the command to stage files for a commit. `.` is everything in the current directory.

Step 7: Check the "status" of your repository

Run `git status` to see the status of your repository. You no longer have untracked files. You have a list of files to be committed.

```
On branch main

Initial commit

Changes to be committed:

  (use "git restore --staged <file>..." to unstage)

        modified:   src/main/java/HelloWorld.java
```

Step 8: Commit the changes to the java file

Run `git commit -m "Added HelloWorld code"`. This command will take all the files to be committed (staged files) and create a point in history related to these changes.

When you create a commit, it should encompass all the changes related to a certain task or logical set of changes

Step 9: Check the log

If you run `git log` , you will see your commit history

```
commit d4e2f53651cb3d9d5debf6102ce80d1ed8a1ff84 (HEAD -> main)
Author: Gregor Dzierzon <gregor@fullstackdevs.net>
Date:   Wed Sep 6 00:02:07 2023 -0600

    Added HelloWorld code

commit 959be294e07eb909fd6653fee0f10b3b95dc5cda
Author: Gregor Dzierzon <gregor@fullstackdevs.net>
Date:   Tue Sep 5 23:42:12 2023 -0600

    initial commit
```

Section 2–3

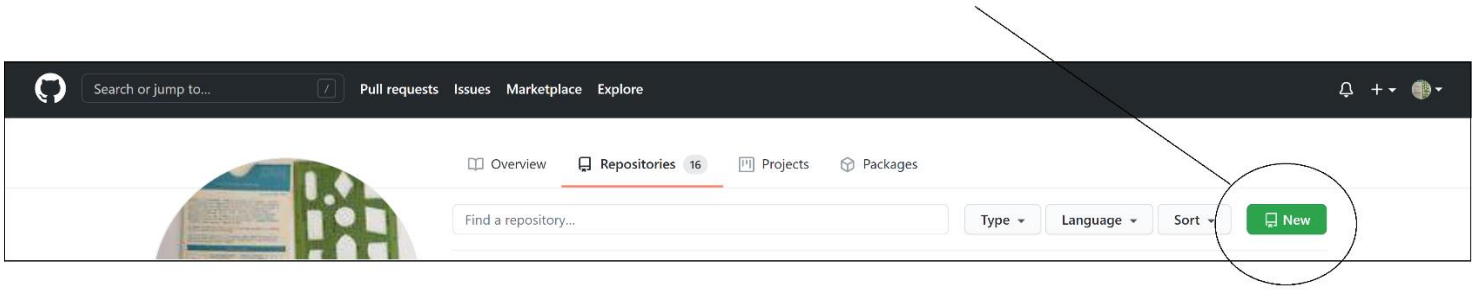
GitHub

GitHub - A Remote Repository Service

- **GitHub is a cloud-based service that can be used to store and manage remote git repositories**
  - Think of it as a backup for your local repo (!)
- **In addition to providing cloud storage for your repositories, it lets you to make your Git repositories available to other developers**
- **Anyone can sign up at GitHub and host public code repositories for free**
  - This makes GitHub very popular as a hosting site for open-source projects
  - Sign up for an account at: `https://github.com`
- **GitHub is a for-profit company and makes money by selling hosted private code repositories and other team-based development services**
- **Many organizations host their own internal GitHub cloud service so that they have complete control over its visibility**
- **However, we will use the public GitHub in this class**

GitHub User Interface

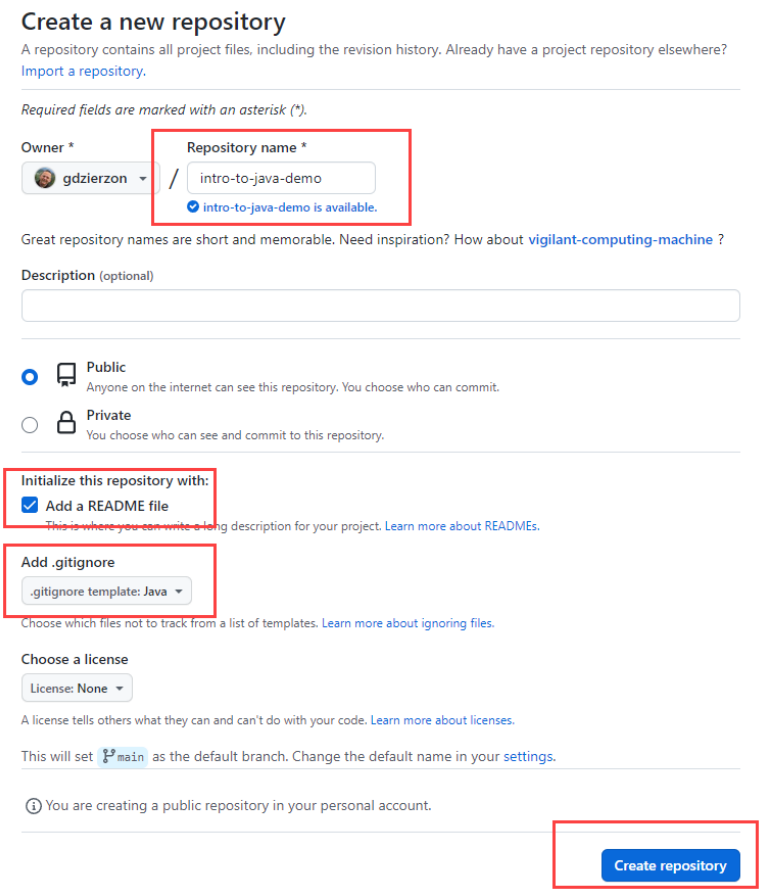
- **GitHub has a web-based graphical interface**
  - It allows you to create new repositories easily



- **It has opinions on how to grant access to your code as well as how people can contribute to your code**

- **It provides several collaboration features for your project, such as:**
  - basic repo management
  - wikis

Creating a Remote Repository



- **After clicking the New button, you can:**
  - Name the repo
  - Set the visibility of the repo (public / private)
  - Add a README file and/or .gitignore file
  - Specify licensing
  - Accept the default main branch name as 'main' or change it to something else

    * Until recently, it defaulted to 'master'

- **There may be small differences on your internal GitHub**

Exercise

In this exercise you will create a GitHub repository that you will use to store and submit your work throughout this coding academy. You will organize all of your work throughout this coding academy within this repo.

EXERCISE 2

Step 1: In your browser navigate to https://github.com and create a new repository with the following settings:

- Name: `java-development`
- Create a `README` file
- Add a `.gitignore` file with a `java` template

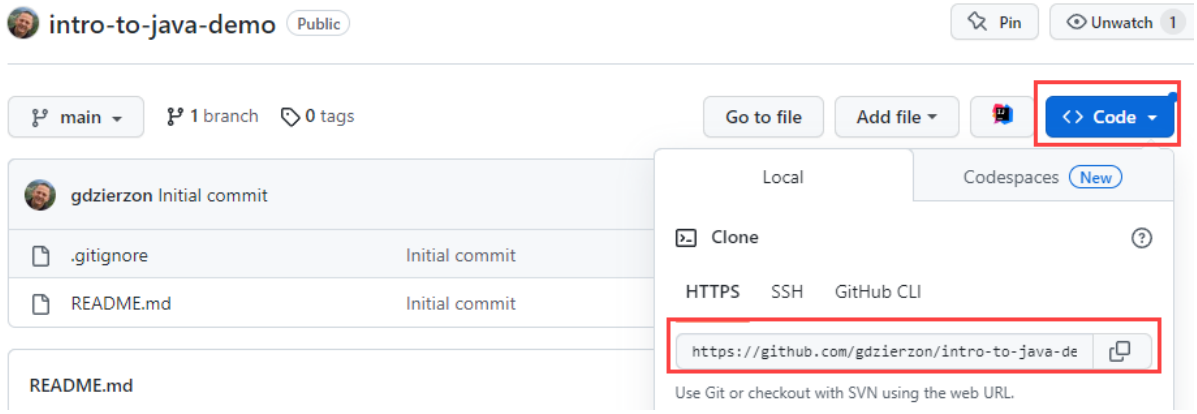
Step 2: Send a link to your GitHub repo to your instructor

Section 2–4

Common Git Commands Used With Remote Repositories

Cloning: `git clone`

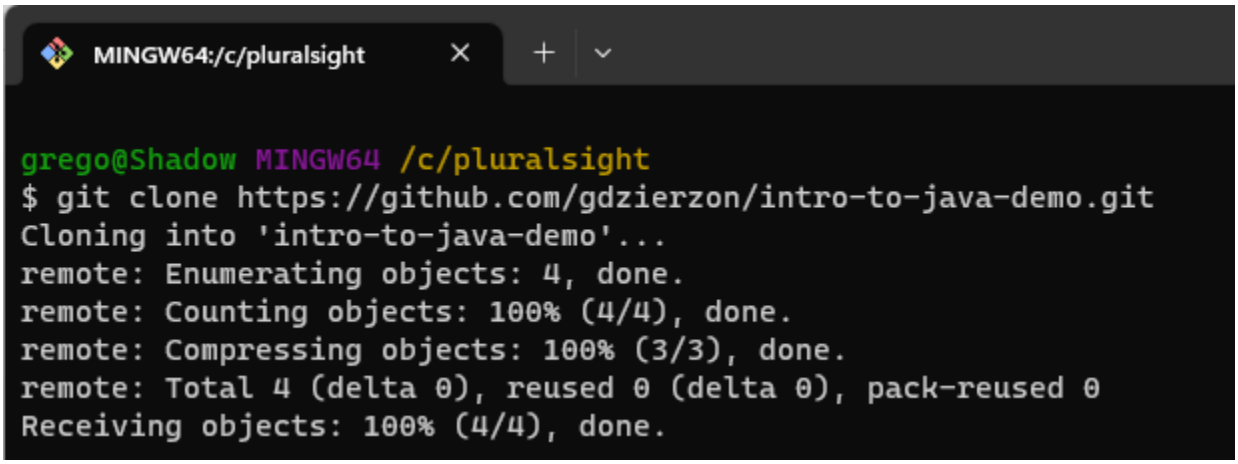
- **The `git clone` command is used to "clone", or make a local copy, of a remote repository**
  - When issued, it is followed by the URL of the remote repository
- **It will create a new repository as a subfolder of wherever you were when you issued the command**
- **Depending on the remote server, you may need to present some authentication credentials:**
  - Use HTTPS and specify a username and password
  - Use SSH keys
  - Specify a Personal Access Token
- **You can locate the URL of your git repo on the repo page**



Example

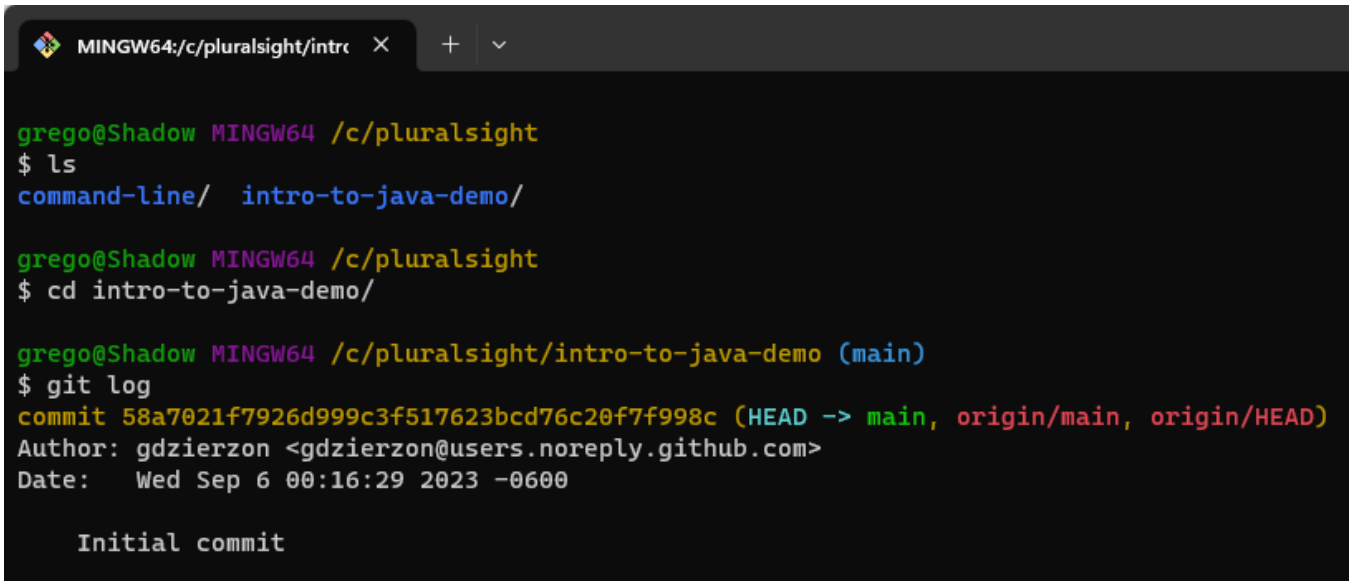
Clone a remote repo to your local system

```
$ git clone https://github.com/gdzierzon/intro-to-java-demo.git
```



Cloning Creates a New Local Repo

- It not only copies the source files down, it brings the entire repository history!



Pushing to the Remote Repository: `git push`

- **The `git push` command says "push the commits from the local branch to the remote branch"**
  - Once executed, commits since your last push will be available on GitHub (the remote repo)

Example

Initial push

```
$ git push -u origin main
```

  - Use the -u flag the FIRST time you push any new branch to create an ***upstream*** tracking connection
  - `origin main` is the name of the remote and the name of the new branch

- **Development is a constant cycle of:**
  - making local changes to files
  - *staging* the files you want to commit, and *committing* to the local repo
  - *pushing* changes up to the remote repo
  - *pulling* changes back from the remote repo

Pulling from the Remote Repository: `git pull`

- **So, why do you pull changes?**
  - Because other developers might be working in the same project and have made pushes that you want to refresh in your own local repo

- **The `git pull` command that says "pull the commits in the remote branch to the local branch"**
  - When executed, any commits made to the remote branch since your last pull become available in your local copy of the repository
  - It's a good practice to make sure you are in the right branch before you execute the pull command

Example

Set the branch you want to update, and pull from the remote repo

```
$ git pull origin main
```

Common Programming Steps

- **Create a remote repository on GitHub first**
- **Clone the repository to your computer**

```
$ git clone <your_github_repo_url>.git
```

- **Create a new Java project in the local repository - commit and push the initial code**

```
$ git add -A
$ git commit -m "initial code commit"
$ git push origin main
```

- **Make changes to your project - commit and push OFTEN**
  - Use `git pull` to "download" any changes made by other team members
  - You should commit and push your changes very frequently - possibly multiple times a day

```
## git pull is generally only required when working as part of a team
$ git pull origin main
## make changes to your code/project
$ git add -A
$ git commit -m "initial code commit"
$ git push origin main
```

Exercise

In this exercise you will clone your GitHub repository to the `C:/pluralsight` directory of your computer and modify the `.gitignore` file.

EXERCISE 3

Step 1: Navigate to your GitHub account and copy the url for your `java-development` repo.

Step 2: Open GitBash, navigate to the `C:/pluralsight` directory and clone it

Step 3: Open the newly cloned `java-development` directory and view all contents (including hidden files and folders)

You should see following files and folders
`.git/` (hidden folder)
`.gitignore` (hidden file)
`README.md`

Step 4: Open the `.gitignore` file in Notepad to view its contents. This file provides instructions for git to define which files and folders should be ignored as you are making changes in that repository (directory).

Add the following line to the end.

```
# IntelliJ IDEA #

**/.idea/
```

NOTE: Throughout this cohort, you will work on many java projects with IntelliJ. Eac time you open a project in IntelliJ it will open or create a folder called `.idea`. This line of code ensures that these `.idea` folders will not get pushed to GitHub.

Step 5: In the `java-development` directory, create a new folder named `workbook-1` and add a `.gitkeep` file (empty directories are ignored by git)

```
java-development/
└── workbook-1/
    └── .gitkeep
```

Step 6: Stage, commit and push your changes

```
git add .

git commit -m "completed git setup exercise"

git push origin main
```

NOTE: You should `add`, `push`, and `push` your changes **EACH DAY** (possibly multiple times a day). So get in the habit of using the previous 3 commands frequently.

Working with IntelliJ IDEA

Student Workbook

Version 2.0 Y

Table of Contents

Module 1

IntelliJ Basics

Section 1–1

IntelliJ

Understanding IntelliJ IDEA

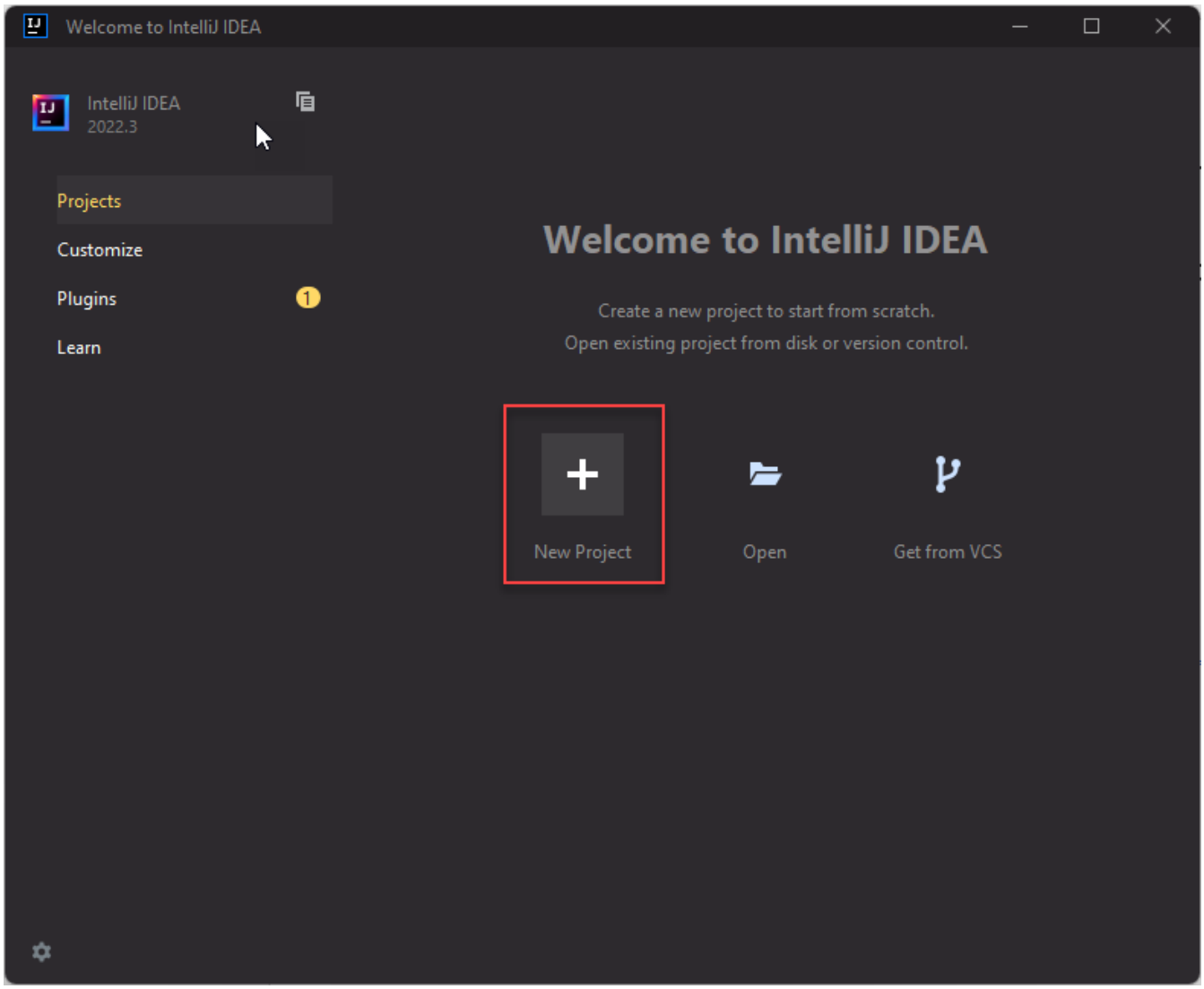
- **Integrated Development Environment - A program that contains comprehensive utilities to develop software**
- **The leading IDE for Java and Kotlin development**
  - Run, Test and Debug Java projects
  - Integrated with Git
- **Plugins available to customize and extend your environment**
- **A product of JetBrains**
  - Free Community Edition or paid Ultimate Edition

Java Projects

- **A Java Project is just a folder that contains all of the project files**
- **Java source code**
- **IntelliJ manages all types of Java projects and build tools**
  - There are a few different project build management tools
    * Maven – currently the most popular build manager for java
    * Gradle
    * Ant
- **Multiple ways to create projects**
  - Create the project directly from IntelliJ
    * IntelliJ will build the appropriate folder structure and starter files
  - Import a project from a VCS – such as Git
  - Create a project manually, then open it in IntelliJ
    * Projects can also be created with tools like Maven

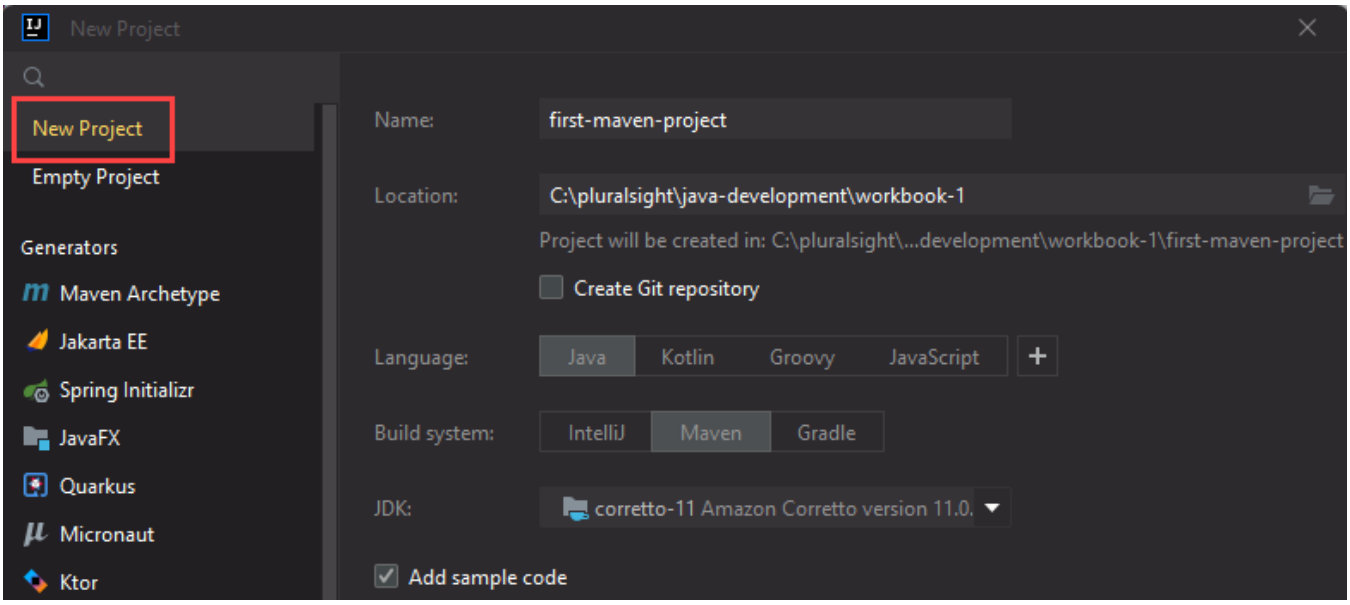
Creating a New Java Project

- Open IntelliJ and select Projects -> New Project



The New Project Dialog

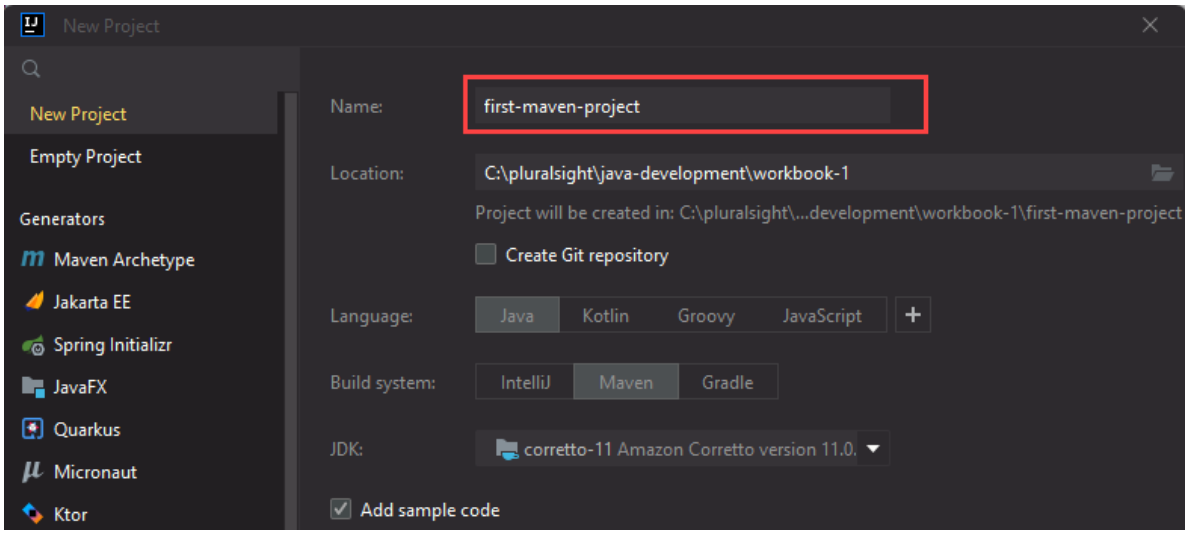
- **The New Project dialog lets you choose from various ways to create and initialize your project**



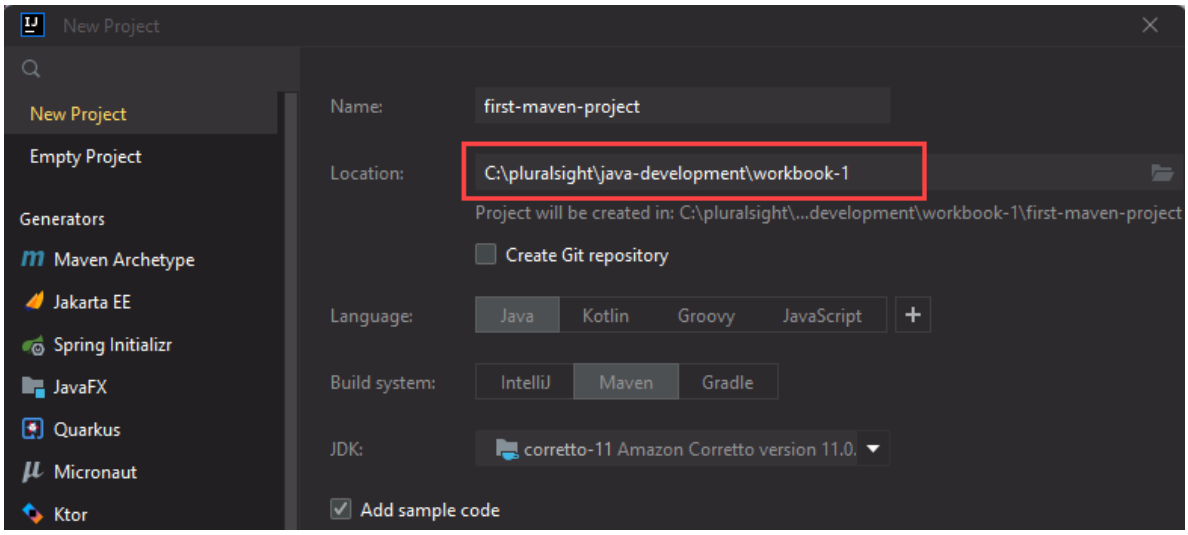
  - New Project creates a basic project with some minimal starter code
  - Empty Project creates the project folder only; all other configuration must be added manually later
  - Generators are specialized project templates
    - These allow you to create projects with significant pre-generated boilerplate (or starter) code

Project Name and Location

- Enter a Project Name



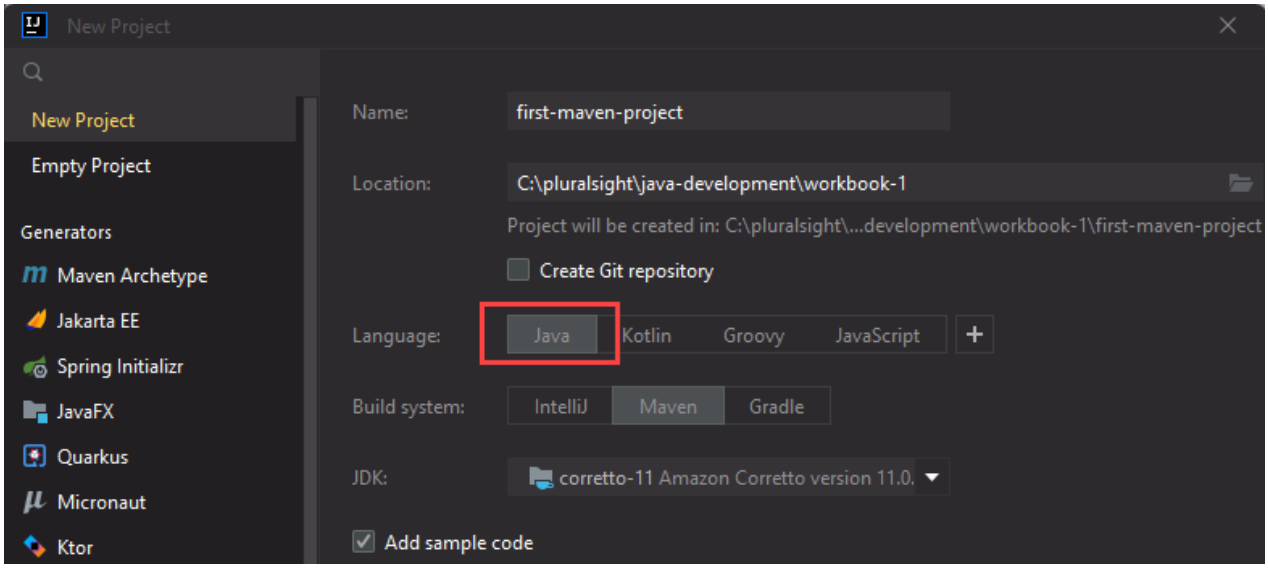
- Select the directory/folder where the project will be saved



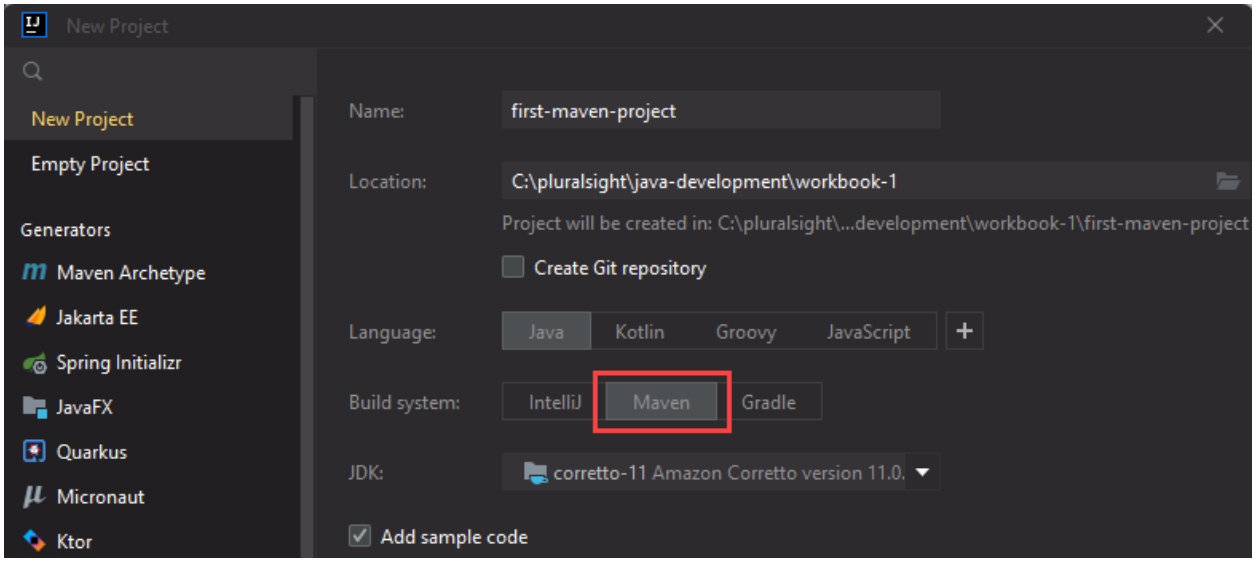
- You can choose whether or not to create a local Git repository to hold the project code

Project Language and Build System

- Select the project language

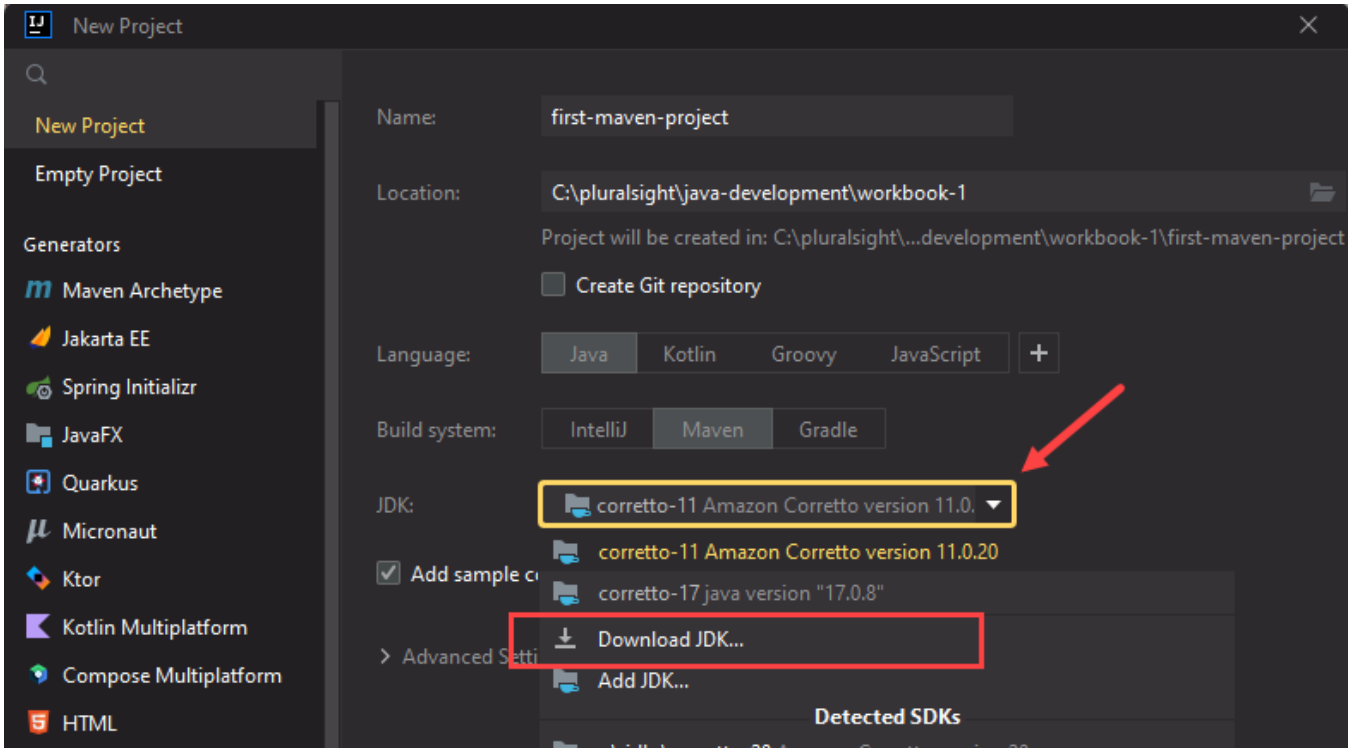


- Select the project build system



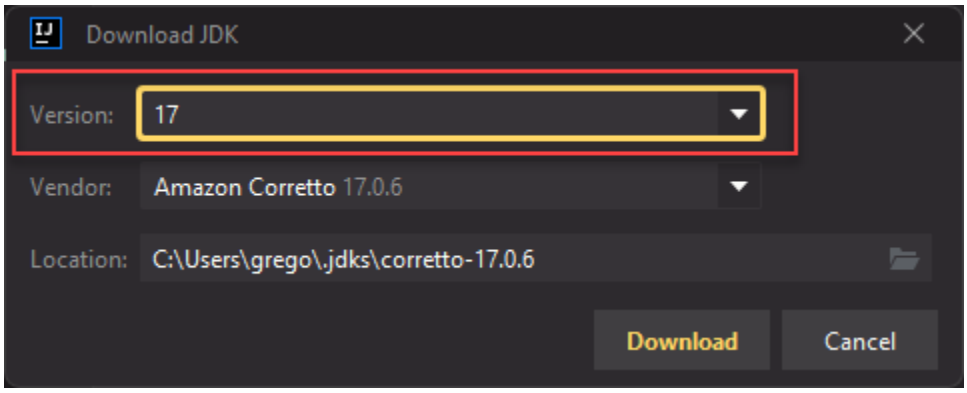
Select the Java Version

- **We will be using Java 17 during this course**
  - If you do not have the Java 17 JDK, IntelliJ will give you options to download and install it directly

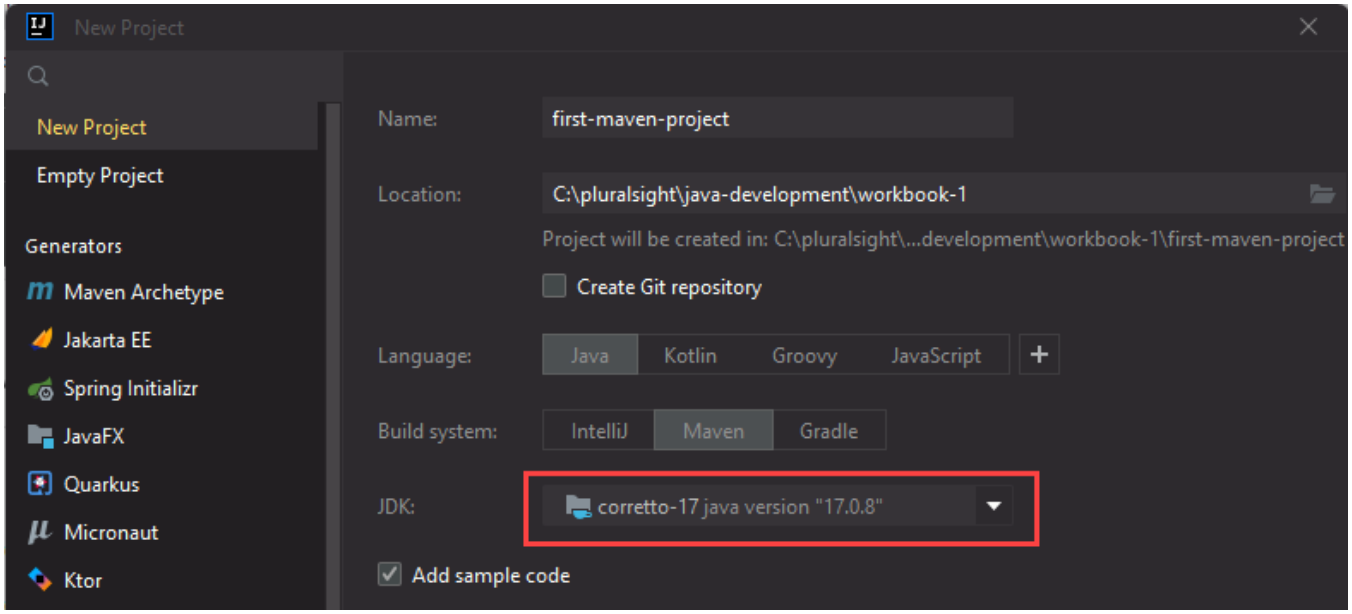


Installing JDK 17 (If Necessary)

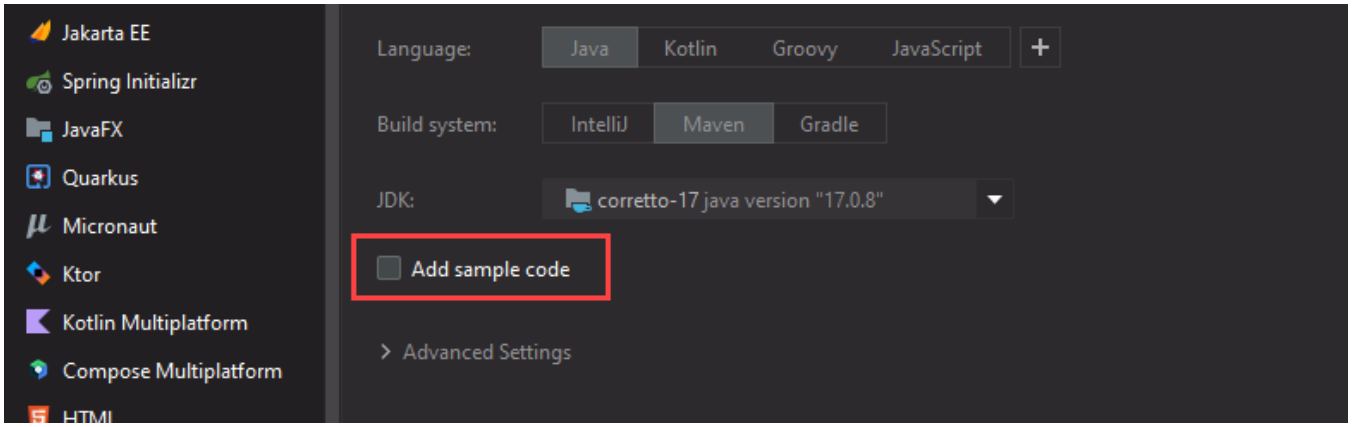
- Select and download the Java 17 JDK



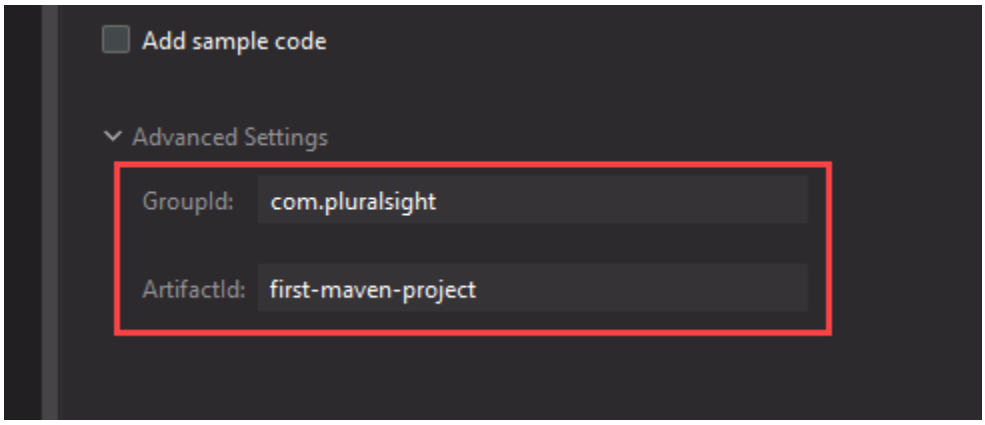
- Ensure that JDK 17 is selected



- Uncheck **Add sample code**



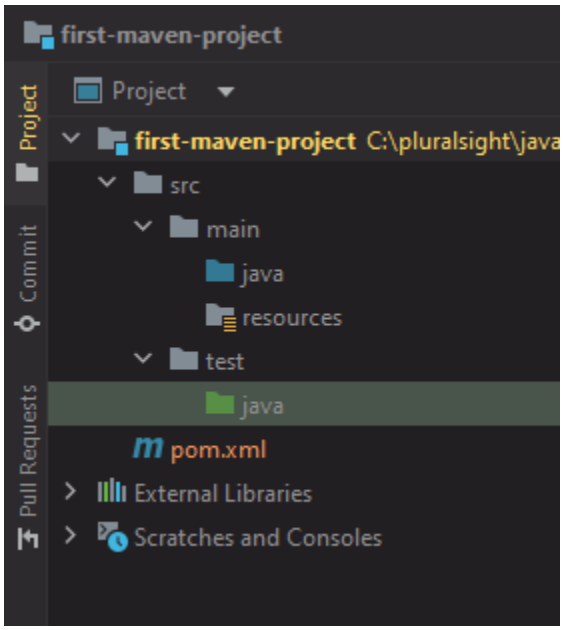
- Expand the Advanced Settings tab and update the **GroupId** and **ArtifactId**



- Click Create

Explore the Project

- After creating the project IntelliJ will open the project folder



- This folder structure is the standard structure for Maven Projects

```
First-maven-project/
    ├── pom.xml
    └── src/
        └── main/
            └── java/
        └── test/
            └── java/
```

The `pom.xml` file

- **The `pom.xml` file is a Maven file that is used to define**
  - project configurations (name, version, jdk build version, etc)
  - a list of external project dependencies

pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.pluralsight</groupId>
    <artifactId>first-maven-project</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

</project>
```

- **We will learn more about this file later in the cohort**

Maven project folder structure

- **All application code must be added to the `src/main/java` directory**

- **Unit tests are added to the `src/test/java` directory**
  - You will learn more about unit tests later in the cohort

Adding a package

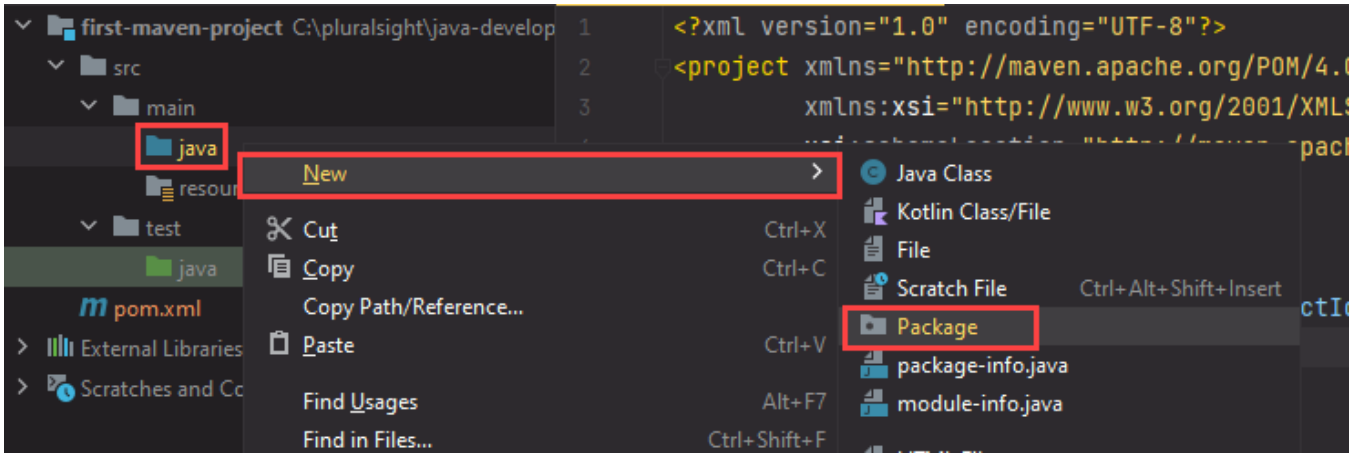
- **Packages appear as folders in a Java project**
  - They allow us to organize our code
  - Packages names are all lower case and follow the following convention

    `com.companyname.projectname`

- **Each dot in the package name implies a subdirectory in the java source tree**

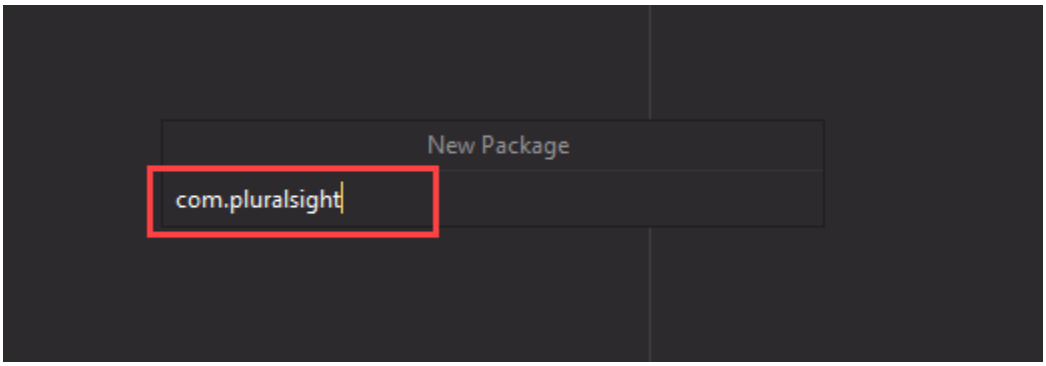
- **Packages are added relative to the `src/main/java` folder**
  - All Java projects *should* have at least one package
    - i.e. we should not add a Java file directly into the `java` folder

- **Create a package by right-clicking on the `main/java` folder and select New -> Package**

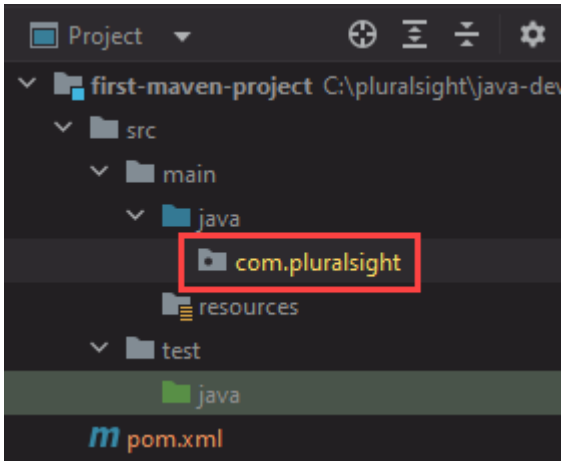


Set the package name

- Add a package name in the **New Package** window and hit **Enter**



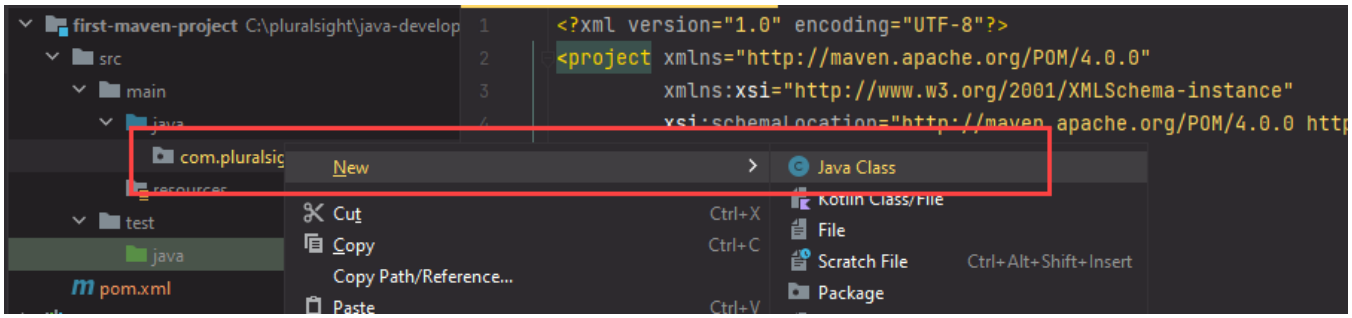
- A new **Package** will have been created for you



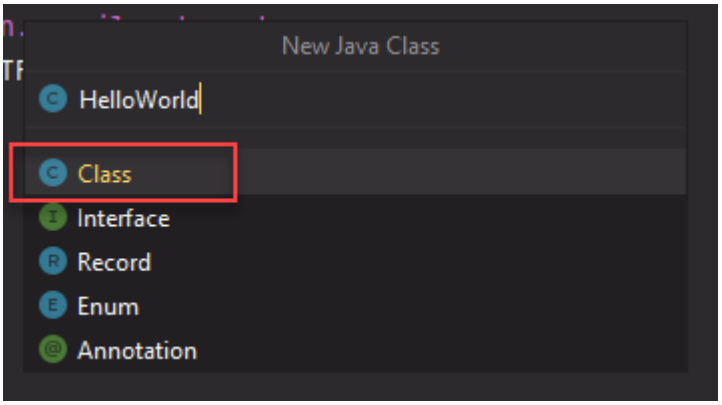
Creating a class

- Now that you have created the package, you can create a class in the package.

- **Right-click** on the package you just create in the **Project Explorer** on the left hand window. Select **New -> Java Class**

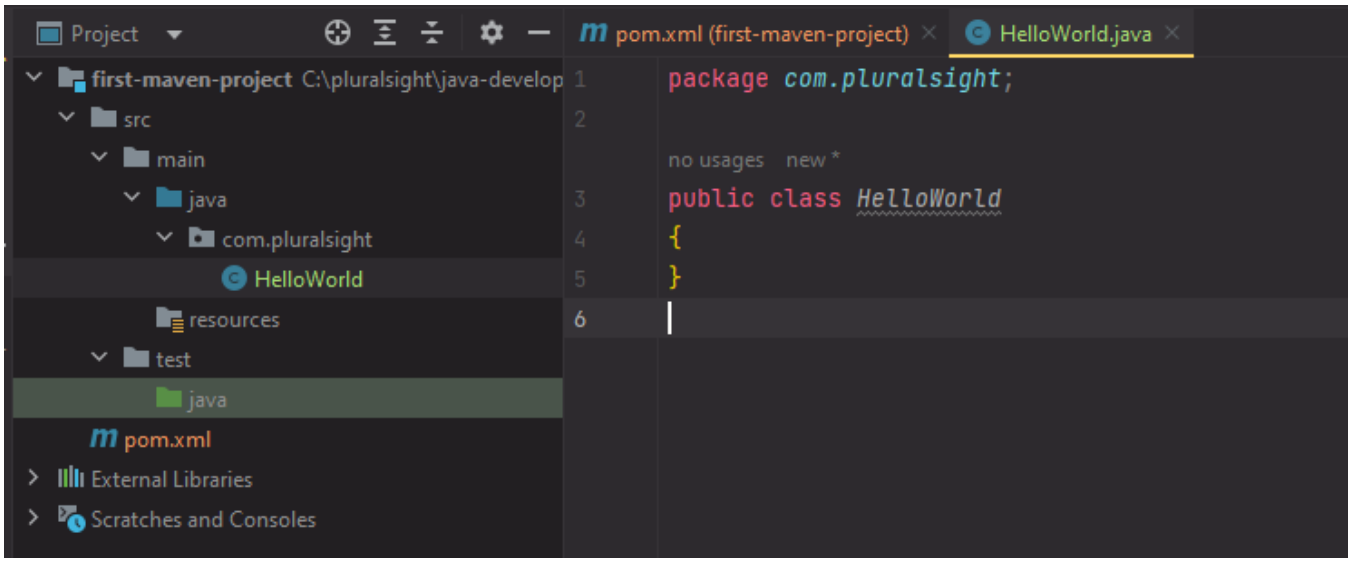


- In the **New Java Class** window enter the name of your class and hit **Enter**
  - Class names always start with an uppercase letter



The Java source file

- Your new class will be created and open in IntelliJ



Finishing the Application

- A Java application must have an **Entry Point** into the application in order to run it
- The **Entry Point** is a function named main, defined like this

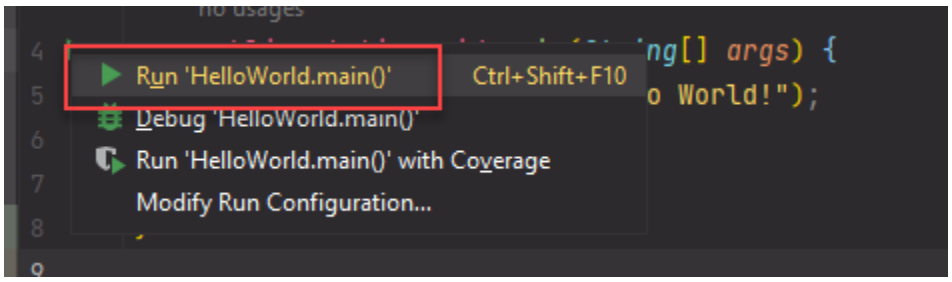
```
public static void main(String[] args) {
   // your code goes here
}
```

- Finish the HelloWorld project

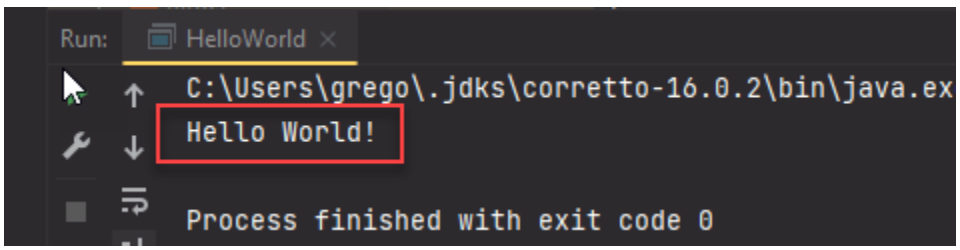
Running your Application

- Once you have added the `static void main` function, a green arrow appears next to that function

- At any time, you can click on the green arrow, to run your application

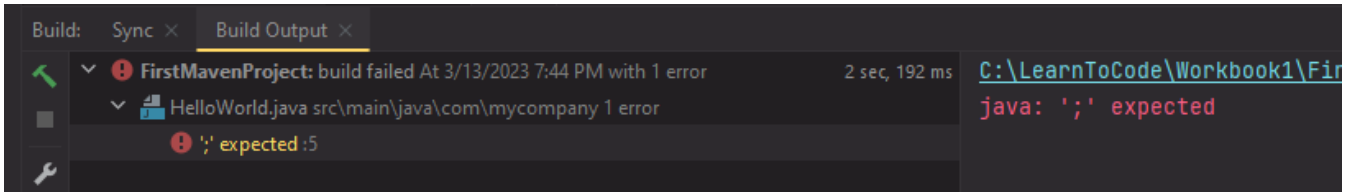


- This should immediately print the results at the bottom of the window



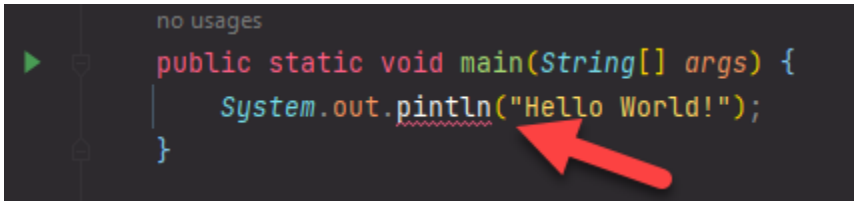
Detecting and Fixing Errors

- **If your application does not run, it is most likely because of an error in your code**
  - IntelliJ can help find errors quickly
- **When you attempt to run the application you may get a compile error message**

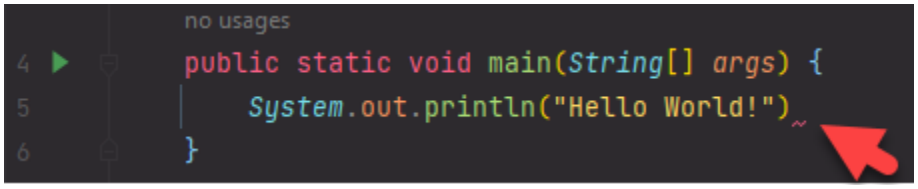


Common Errors

- **Misspelled functions or variables**
  - Missing "r" in `println()`



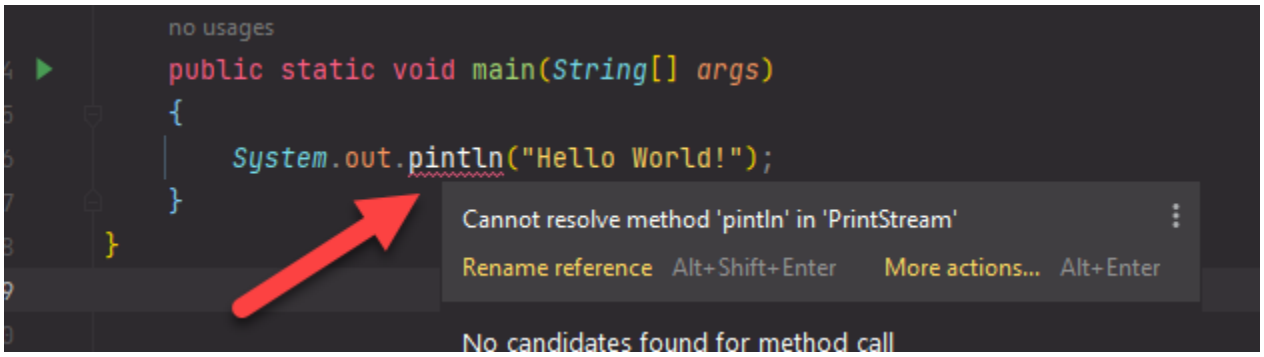
- **Missing semi-colon at the end of a line**



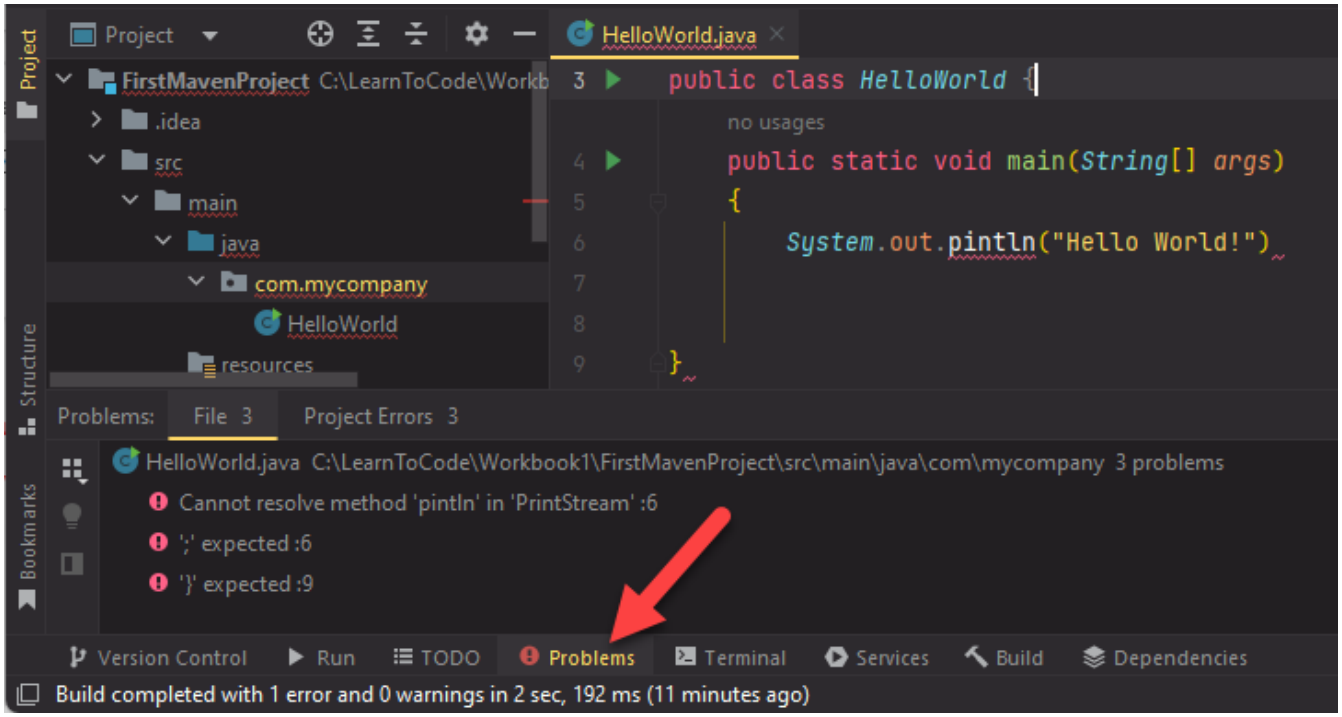
- **Missing close curly brace**
  - Here the compiler believes that the function has a close curly but the class is missing one

Lean on Your Tools

- DO read the error messages; they really want to help!
- You can also hover over the **red squiggly** line to get information about the error



- Additionally there is a **Problems** tab at the bottom of IntelliJ that will list all potential problems with your code



Exercises

Complete the following exercise by adding the new project into the `C:/pluralsight/java-development/workbook-1` folder.

EXERCISE 1

Using IntelliJ, create a new Java application that will list at least 10 items that should be on your shopping list.

1. Create a new package named `com.pluralsight`
2. In the `com.pluralsight` package create a new java class named `ShoppingList`. Remember it must be in a .java file of the same name.
3. Within the `ShoppingList` class, create a `main` method.
4. In the `main()` method use the `System.out.println()` to display a shopping list with at least 10 items.
5. Run the program. If there are any errors, fix them and run it again.
6. Push your changes to GitHub (always stage, commit and push your changes)
   i. `git add -A`
   ii. `git commit -m "completed ShoppingList app"`
   iii. `git push origin main`

Java Fundamentals

Student Workbook 1 - Introduction to Java

Version 2.0

Table of Contents

Module 1

Introduction to Programming

Section 1–1

Thinking About Programming

Programming

- **Programming is essentially the process of telling a computer what to do -- step by step -- in a language the computer understands**

- **When you program, you must give precise and detailed instructions**
  - "Close only counts in horseshoes and hand grenades"

- **When you program, you must use a language the computer understands**
  - There are many, maNY, MANY languages out there
  - This week, we will begin our study of Java
  - Along the way, you'll learn about some other languages like bash, SQL, XML, JSON and others!

- **Java is a good language to learn**
  - Java is used by more than 6 million developers and runs on more than 5.5 billion devices

Section 1–2

Learning to be Precise

Learning to Be Precise

- **To demonstrate using precise instructions, we'll spend a little time on a paper/pen coding exercise**

- **In this first phase, we will make a monkey move to a banana and eat it using a set of specific commands**
  - Note: the ideas for this exercise came from `codemonkey.com`[1]

- **Right now, your monkey understands 3 commands**

  `Step` *number*
  where *number* is the number of squares to move

  Example: `Step 5`

  `Turn` *direction*
  where *direction* is `left` or `right`

  Example: `Turn left`

  `Eat banana`
  This only works if the monkey is standing on an adjacent square to the banana *and* facing it.

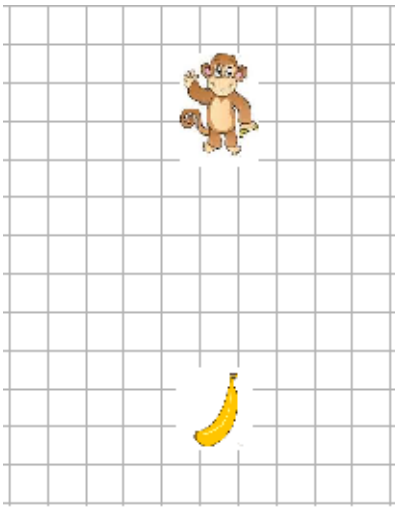
  Example: `Eat banana`

[1] And before there was codemonkey.com, there was a really old programming language called "LOGO", in which you could move "turtles" to draw pictures. It still works, and we'll try it later.

Examples: "Program the Monkey"

Example

Move the monkey to the banana and have him eat it.

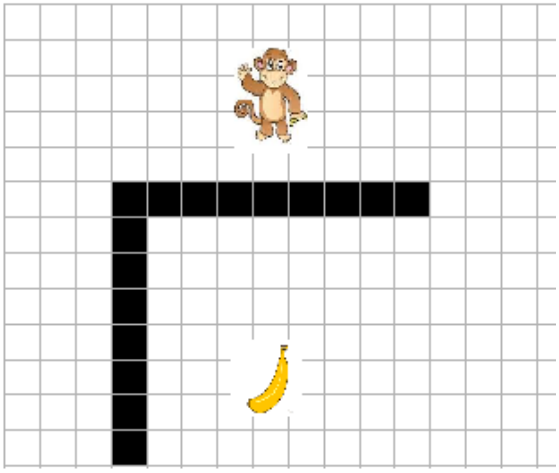


SOLUTION

```
Step 6
Eat banana
```

Example

Move the monkey to the banana and have him eat it. Avoid the barrier. Pay attention to the direction the monkey is facing.



SOLUTION

```
Turn left
Step 5
Turn right
Step 7
Turn right
Step 4
Eat banana
```

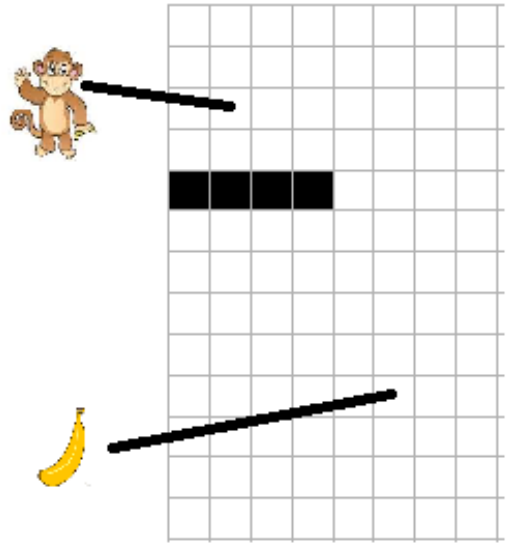
- **Note: Just like programming in real life, there is often more than one way to be successful**

Exercises: "Program the Monkey"

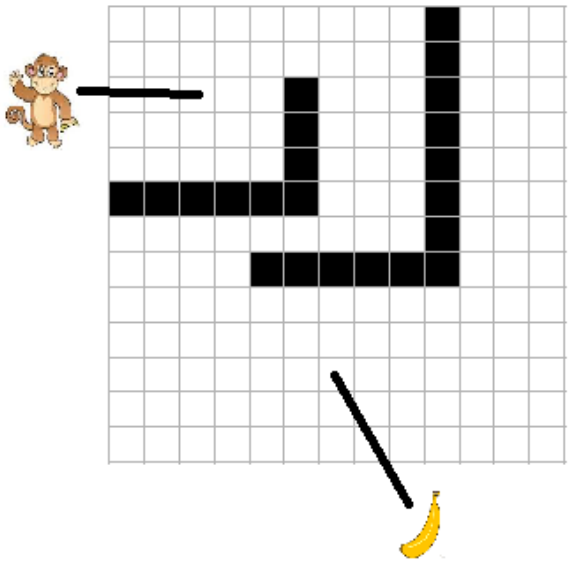
In the next few exercises, we will let you figure out the steps needed to "program your monkey" to move to and eat the banana. We are using lines to show you where the monkey and banana are so that the positions are very precise.

When you finish, talk it over in your group to see how others programmed their monkey.

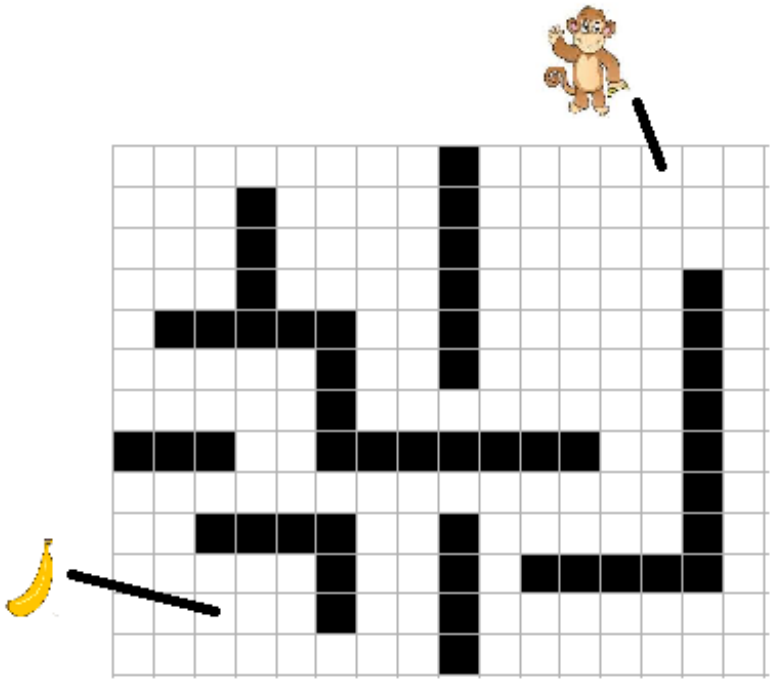
EXERCISE 1



EXERCISE 2



EXERCISE 3



Expanded Command Vocabulary

- **When you are learning a programming language, you start with just a few commands and learn to use them**
  - When you conquer them, you learn more commands and how to use them too!
  - Before long, you know a LOT about the programming language

- **Your monkey understands the commands we discussed before, plus some commands that can be used to pick up an item or drop an item**

:

`Step` *number*
where *number* is the number of squares to move

`Turn` *direction*
where *direction* is `left` or `right`

`Pickup` *item*
where *item* is `banana` or `basket`

When you pick up an item, you must be facing the item in an adjacent square.

Example: `Pickup basket`

`Drop` *item*
where *item* is `banana` or `basket`

When you drop an item, it stays in the square directly in front of the square you are standing in. NOTE: Only one item can reside on a square, however, if the banana is IN the basket that counts as one item.

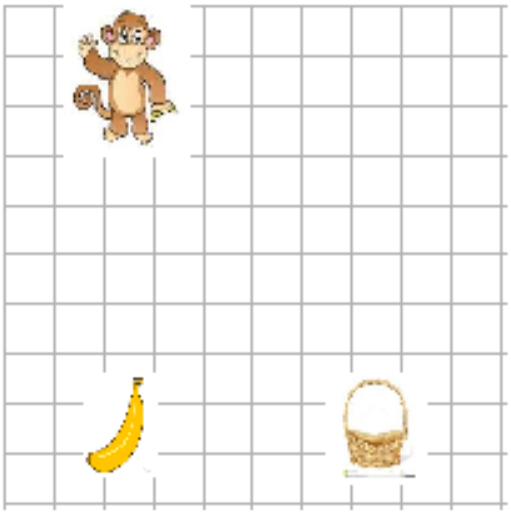
Example: `Drop basket`

`Eat banana`
This only works if the monkey is standing on square adjacent to the banana directly facing it.

Examples: "Smarter Monkey"

Example

Make the monkey drop the banana into the basket.



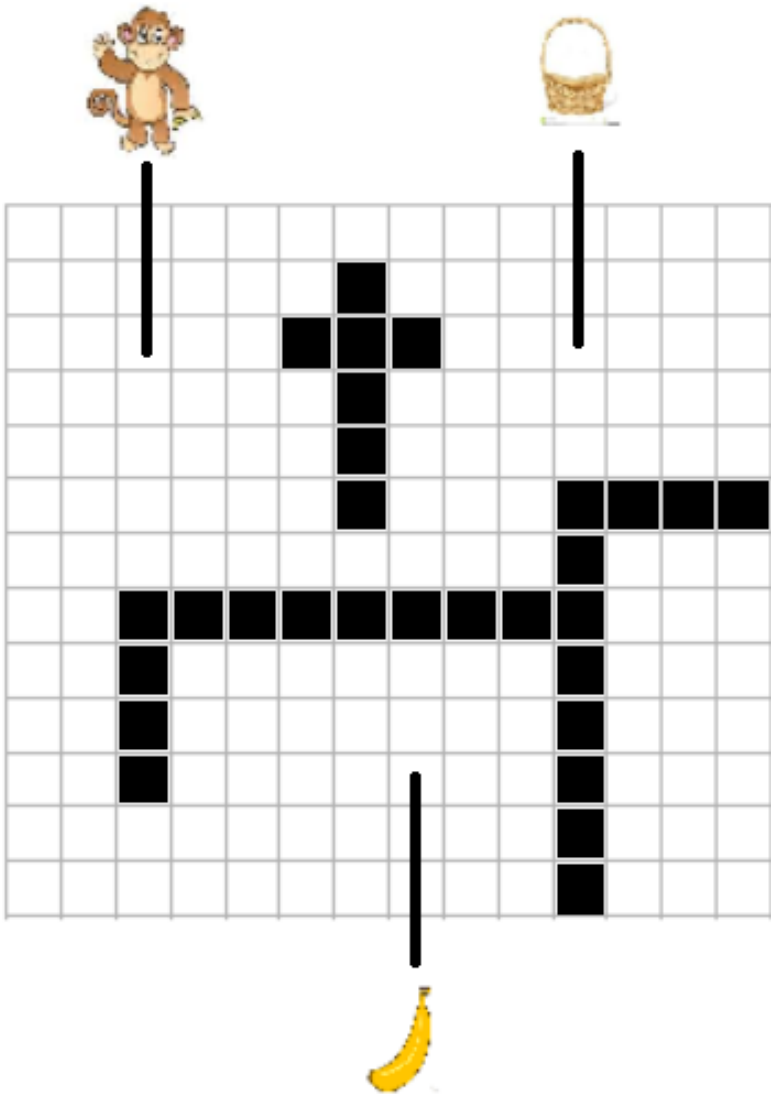
SOLUTION

```
Step 5
Pickup banana
Turn left
Step 5
Turn right
Drop banana
```

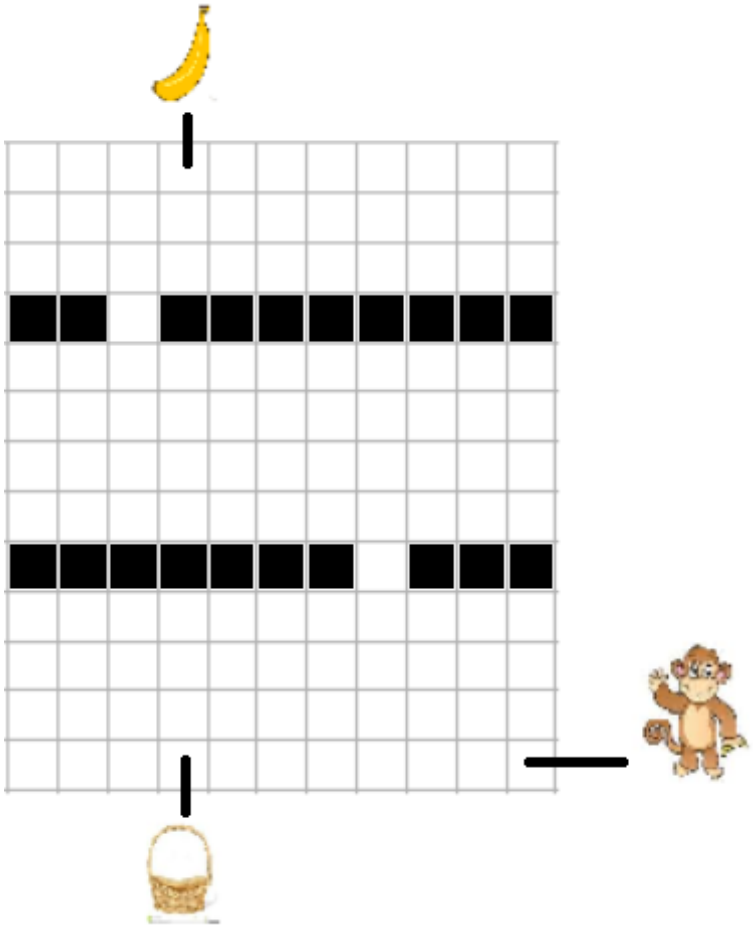
Exercises: "Program the Monkey" - Part 2

In these exercises, you need to "program your monkey" to put the banana in the basket. Like before, when you finish, talk it over in your group to see how others programmed their monkey.

EXERCISE 1



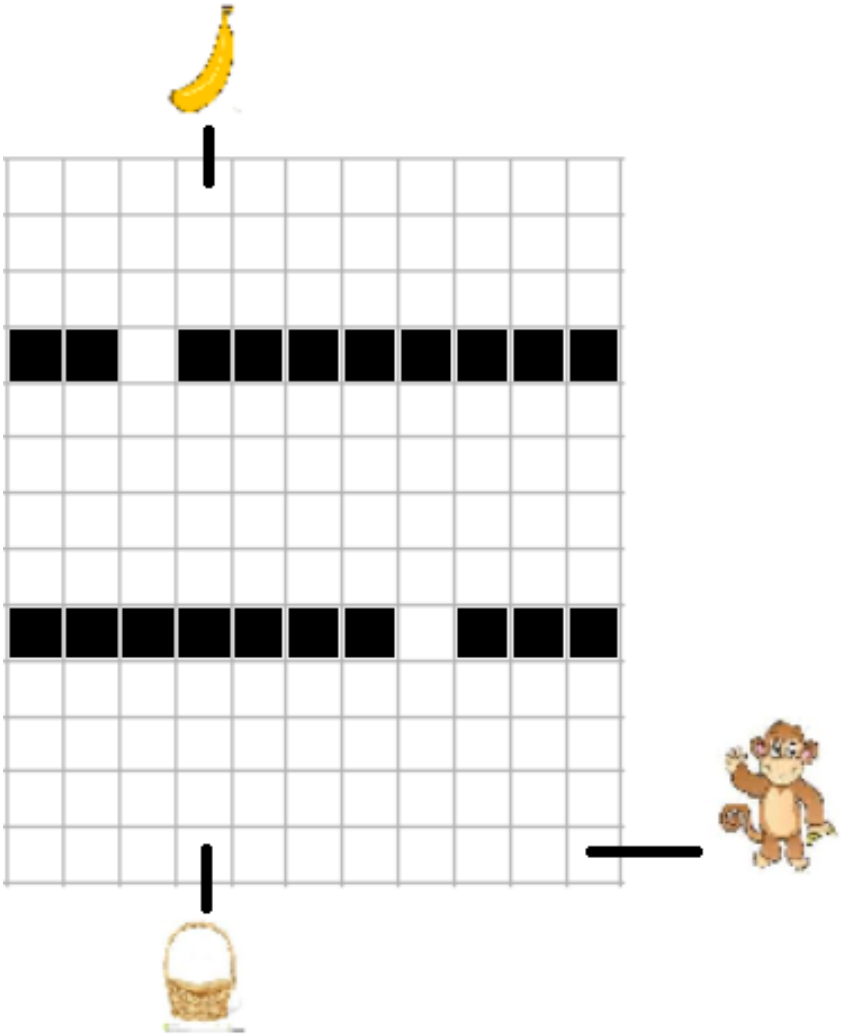
EXERCISE 2



EXERCISE 3

Would your answer change if you knew that the goal was to take the smallest number of steps possible?

The goal here is: *the banana ends up in the basket*. It doesn't say where the basket has to be. Do you pick up the banana and carry it to the basket? Do you take the basket over near the banana?



Section 1–3

Dealing with Ambiguity

Dealing with Ambiguity

- Sometimes, it's harder to generate a list of the tasks you must perform when the process isn't quite as well defined or as visual as working with monkeys!

- In these situations, we often write down the steps that we need to perform in English in order to "get our head around" the problem

- Computers follow your exact instructions -- but learning to be precise is tricky!

- Watch the "Exact Instructions Challenge" on YouTube before proceeding any further!

  https://www.youtube.com/watch?v=Ct-lOOUqmyY

Trying to Be Exact

- **How do you make scrambled eggs for two people?**

ATTEMPT 1

```
Get four eggs from refrigerator
Crack eggs into skillet
Scramble eggs
```

  - Do you understand the process from our first attempt
  - What's wrong with the description above?

- **The write up has some pre-conditions to scrambling eggs, but it doesn't really tell us how to scramble eggs**

ATTEMPT 2

```
Get four eggs from refrigerator
Crack eggs into skillet

Repeat
[
   Stir eggs as they cook
   Wait 30 seconds
] until they are the desired consistency

Turn stove off
```

  - Better, but where did the eggs come from?

- **How exact do you have to be?**
  - A lot of that depends on how intelligent the reader is

ATTEMPT 3

```
Open the refrigerator
Get four eggs
Close refrigerator

Crack eggs into skillet

Repeat
[
   Stir eggs as they cook
   Wait 30 seconds
] until they are the desired consistency

Turn stove off
```

- **Of course, if we are programming a computer - it has no preconceived insights and you have to be very exact**

- **You also have to describe possible error conditions**
  - For example, what if we are out of eggs?

ATTEMPT 4

```
Open the refrigerator
If we are out of eggs
[
   Make different breakfast plans
   Close refrigerator
   Exit script
]
Get four eggs
Close refrigerator

Crack eggs into skillet

Repeat
[
   Stir eggs as they cook
   Wait 30 seconds
] until they are the desired consistency

Turn stove off
```

- **Not bad -- but are you really being detailed?**
  - Where did the skillet come from?
  - Will your eggs possibly stick to the pan as you cook?

ATTEMPT 5

```
Open the refrigerator
If we are out of eggs
[
   Make different breakfast plans
   Close refrigerator
   Exit script
]
Get four eggs
Close refrigerator

Get skillet from rack above stove
Spray PAM all around skillet for 3 seconds
Put skillet on stove

Crack eggs into skillet

Repeat
[
   Stir eggs as they cook
   Wait 30 seconds
] until they are the desired consistency

Turn stove off
```

- **Can you see your sarcastic friend following the instructions for spraying PAM?**
  - My sarcastic friend would spray the handle and bottom of the skillet!
  - Oops! Replace with:

```
Spray PAM all around the inside of the skillet for 3 seconds
```

Complex Problems

- **When you are working with complex problems, you really have to think about all aspects of the problem**
  - Where did the PAM come from?
  - Do they know how to crack eggs?
  - What if some of the shell ended up in the pan?
  - What did they do with the shells once they put the egg in the skillet?
  - Who turned on the stove?
  - Stir the eggs with what?
  - What is "desired consistency"?
    - In my house, one of us likes eggs runny and the rest want them cooked into dried hard unappetizing flecks!"
  - Is plating the eggs part of the scope of the problem?
  - Is cleaning up your mess part of the scope of the problem?

- **Other things that might impact how you write include:**
  - The vocabulary of the user (new to the process, experienced)
  - The skills of the user (never cooked, casual cook, chef)
  - The complexity of the process
    - Sometimes, you just won't understand enough about the process to get all of the details right at the beginning
    - Software development is an iterative process!

Exercises

EXERCISE 1

Write out the process of how to "brush your teeth". This may take 4-10 minutes if you do a good job.

Then, meet with your group to go over the write-ups. Tweak the write-ups as needed.

Your team will "present" a "brush your teeth" process to the class as a whole. Your team can select one of the team member's to advance or can blend them together using steps from each team member.

Module 2

Introducing Java

Section 2–1

Overview of Java

Java

- **Java is an object-oriented programming language that has been around since the mid 1990s**
  - It was originally developed by Sun Microsystems, and has now been acquired by Oracle
- **One of its strengths is that it is *platform independent***
  - Once translated into Java bytecode, it can run on many different types of computers
  - Often people refer to this as "write once, run anywhere!"
- **Platform independence is what makes Java a popular choice desktops, mobile devices, servers, IoT, and more**

Versions of Java

- **Java has continued to evolve -- Java 17 is the latest "Long Term Support" version**
  - However, many production apps are written using older versions

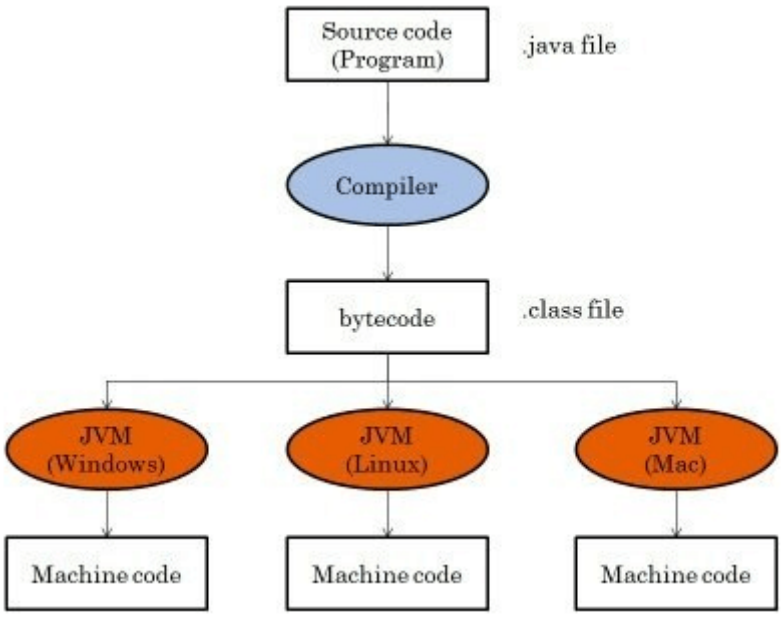
Version	Release date	End of Free Public Updates[1][5][6][7]	Extended Support Until
JDK Beta	1995	?	?
JDK 1.0	January 1996	?	?
JDK 1.1	February 1997	?	?
J2SE 1.2	December 1998	?	?
J2SE 1.3	May 2000	?	?
J2SE 1.4	February 2002	October 2008	February 2013
J2SE 5.0	September 2004	November 2009	April 2015
Java SE 6	December 2006	April 2013	December 2018 December 2026 for Azul[8]
Java SE 7	July 2011	April 2015	July 2022
Java SE 8 (LTS)	March 2014	**January 2019 for Oracle (commercial)** December 2030 for Oracle (non-commercial) December 2030 for Azul December 2030 for IBM Semeru At least May 2026 for Eclipse Adoptium At least May 2026 for Amazon Corretto	December 2030[9]
Java SE 9	September 2017	March 2018 for OpenJDK	N/A
Java SE 10	March 2018	September 2018 for OpenJDK	N/A
Java SE 11 (LTS)	September 2018	September 2026 for Azul September 2026 for IBM Semeru At least October 2024 for Eclipse Adoptium At least September 2027 for Amazon Corretto At least October 2024 for Microsoft[10][11]	September 2026 September 2026 for Azul[8]
Java SE 12	March 2019	September 2019 for OpenJDK	N/A
Java SE 13	September 2019	March 2020 for OpenJDK	N/A
Java SE 14	March 2020	September 2020 for OpenJDK	N/A
Java SE 15	September 2020	March 2021 for OpenJDK March 2023 for Azul[8]	N/A
Java SE 16	March 2021	September 2021 for OpenJDK	N/A
Java SE 17 (LTS)	September 2021	September 2029 for Azul At least September 2027 for Microsoft At least TBA for Eclipse Adoptium	September 2029 or later September 2029 for Azul
Java SE 18	March 2022	September 2022 for OpenJDK	N/A
Java SE 19	September 2022	March 2023 for OpenJDK	N/A
Java SE 20	March 2023	September 2023 for OpenJDK	N/A
Java SE 21 (LTS)	September 2023	TBA	September 2031[9]

Legend: Old version | Older version, still maintained | **Latest version** | Future release

from Wikipedia - https://en.wikipedia.org/wiki/Java_version_history

Java Bytecode and the Java Virtual Machine (JVM)

- **When a Java program is compiled, it is not translated into the machine code native to the developer's computer**
  - This would mean it could only run on similar computers
- **Instead, Java programs are translated into something called Java *bytecode***
  - Bytecode is stored in a `.class` file and is eventually executed on a computer by using a JVM
- **The Java Virtual Machine (JVM) is the program responsible for loading and executing a Java application**
  - It executes the Java bytecode instructions
  - There are several JVMs available depending on the type of computer you use

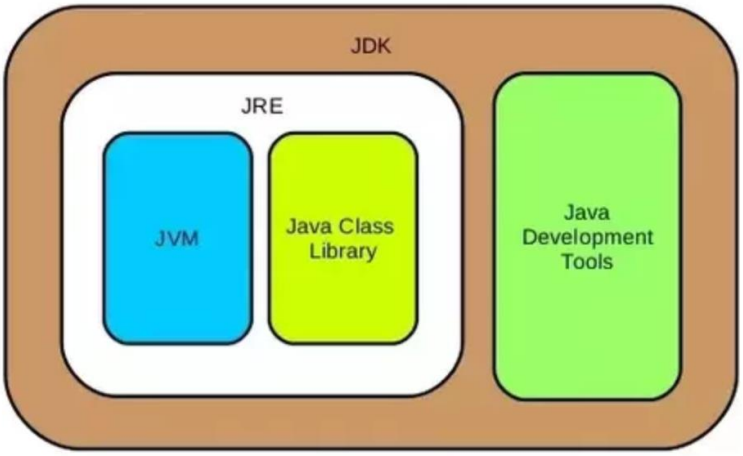


Java Runtime Environment (JRE)

- **You must install the Java Runtime Environment (JRE) in order to *run* Java applications on your computer**
- **The JRE is a software layer that sits on top of a computer's native operating system**
- **It provides the class libraries and other resources (including the JVM) needed to run Java applications**
- **However, the JRE does not contain the resources a developer needs for creating Java applications**

Java Developer Kit (JDK)

- **The Java Developer Kit (JDK) is set of tools for developing Java applications**
  - It includes a CLI, the Java compiler, debuggers, etc.
- **If you want to create Java applications, you must install a JDK**
- **Developers choose JDKs by Java version and by package or edition—**
  - Java Standard Edition (Java SE) <-- We want this one!
  - Java Enterprise Edition (Java EE)
  - Java Mobile Edition (Java ME)
- **The JDK also includes a compatible JRE because so that the Java application can be tested and run on the developer's machine**



Compiling Java

- **To compile a Java application into bytecode, you use `javac`**

- **`javac` is a Java compiler**
  - It confirms the grammar you coded is correct
  - It then translates the code into bytecode and places it in a `.class` file

- **If you have syntax errors, the compiler will not generate the bytecode**

- **This is the difference between a compiled language like Java and an interpreted language like JavaScript**
  - You can't even try to run the code if there are syntax errors

Section 2–2

Configuring your Machine

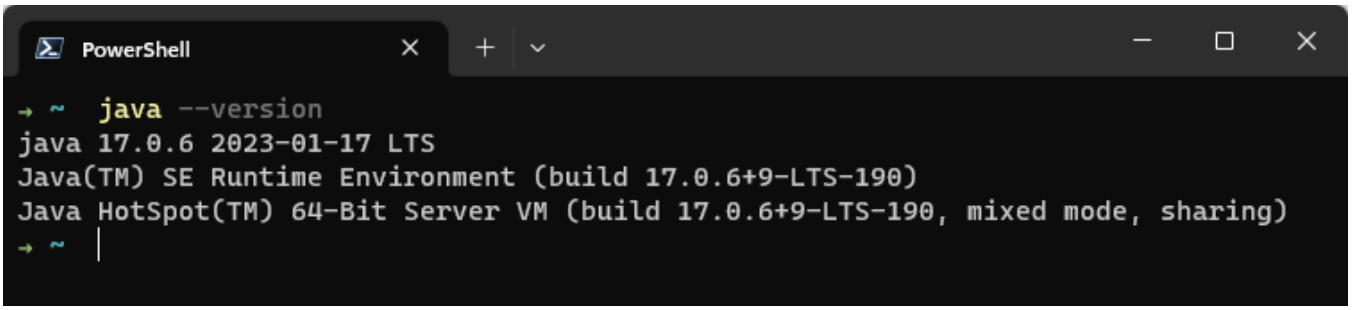
Configuring your Development Machine

- **If you don't have a JDK, you can download the version you need here**
  - https://www.oracle.com/java/technologies/downloads/
- **Once you have installed a JDK on your development machine, you will need to do some (hopefully) simple configurations**
- **Step 1: Configure the JAVA_HOME environment variable**
  - On a Windows machine, this will be something like: `C:\Program Files\Java\jdk-17`
- **Step 2: Add the `bin` folder containing Java developer tools to your PATH environment variable**
  - On a Windows machine, this will be something like: `C:\Program Files\Java\jdk-17\bin`
- **Step 3 (possible): If your path variable includes an Oracle Java reference, either move it to the bottom of the PATH list or remove it**

Verify Java Configuration

- Open Windows Terminal and execute the following command

```
java --version
```



Module 3

Basic Java Syntax

Section 3–1

Basic Java Syntax

Java Syntax

- **Java is one of the C-family languages**
  - Includes C, C++, C#, Java, Kotlin, etc.

- **This means, amongst other things:**
  - it is case sensitive, everywhere and always (!)
  - it uses semicolons (;) as command terminators
  - there are 2 types of comments
    - single line comments
      ```
      //   this line is a comment
      ```
    - and multi-line comments
      ```
      /*
      this line is a comment
      this line is also a comment
      */
      ```
  - the `if`, `switch`, `for`, and `while` statements will look very familiar

- **Source code for Java applications is contained in one or more `.java` files**
  - Unlike most other languages, Java really cares about the name of the file
    - The file must start with an uppercase letter and match the name of the class it contains

Javadoc

- **The Javadoc tool, included in the JDK, generates API documentation from comments found in the source code**

- **Comments should have a specific style**

  Syntax

  NOTE: a javadoc comment begins with 2 "stars"

```
/**
 * This is a Javadoc comment
 */
```

- **Javadoc looks for these special comments in front of class, method, and variable declarations**

- **These comments are commonly made up of two sections:**
  - Description of what you are commenting
  - Tags marked with an `@` symbol that describe specific metadata

Example: Code Written with Javadoc in Mind

Example

```
/**
 * HelloWorldApp is an example of most people's first
 * Java program
 *
 * @author Dana Wyatt
 *
 */
public class HelloWorldApp {

    public static void main(String[] args) {
        /**
         * The greeting to be displayed in the Console.
         */
        String message = "Howdy Java!";
        display(message);
    }

    /**
     * Dislays any message the console window
     *
     * @param message the message displayed
     */
    public static void display(String message) {
        System.out.println(message);
    }
}
```

- **You can run Javadoc from the terminal window**

Example

```
javadoc HelloWorldApp.java
```

- **We won't focus on Javadoc in this course, but it produced the online docs that you'll use all the time**

Data Types

- **There are two types of data types in Java:**
  - *Primitive* data types include: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`
  - *Non-primitive* data types include arrays, classes, and interfaces

- **Primitive types are pre-defined by Java and represent simple values:**
  - `boolean` - a true/false value in a *single bit*
  - `char` - a single Unicode character (16-bits)
  - numbers such `byte`, `short`, `int`, `long`, `float`, and `double` as that have different memory structures and sizes
    - Note: floats and doubles store their values in an imprecise way - they are not usually used for storing currency values that must be precise

- **Non-primitive types are arrays, class types, etc**
  - They are built by combining primitive types in various ways
  - Java libraries provide many predefined types
  - The `String` data type is actually implemented in a Java class
    - That's why its name starts with a capital S!
  - Programmers also create many classes as they go about developing their application using object-oriented design principles

Variables

- **Java is a statically-typed language and every variable MUST be assigned a data type**

- **Variables in Java are similar to those in other languages**
  - They hold information and allow a programmer to interact with it by the variable name

Example

```
// hold a whole number
int num1;
long num2;

// hold a number with decimal points
float num3;
double num4;

// hold a Boolean
boolean isHappy;

// hold a string
String word;
```

- **You can declare many variables on the same line if they are of the same type**
  - You can choose to initialize some and not others

Example

```
int x = 5, y, z = 50;
```

Read-only Variables

- **In Java, a read-only variable is declared with the keyword `final`**

- **It must be initialized when you declare the variable**
  - Any attempt to change it later will be an error

Example

```
final String name = "Dana";

System.out.println("Your name is " + name);

name = name + " Wyatt";  // syntax error!!

System.out.println("Your full name is " + name);
```

Data Types : Numbers

- **Java allows programmers to choose from many primitive data types when working with numbers**
  - The difference between them boils down to how many bits are allocated in the computer's memory and how does the computer format those bits
- **For example, a `byte` is an 8-bit number**
  - 7 of the bits are used for the number and 1 bit is used to indicate whether the number is positive or negative
  - This means the range of a `byte` variable is -128 to 127
- **Other whole number types include:**
  - a `short` that is 16 bits and its range is -32,768 to 32,767
  - an `int` that is 32 bits and its range is -2,147,483,648 to 2,147,483,647
  - a `long` that is 64 bits and its range is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
- **Java's two floating point primative types are:**
  - a `float` that is 32 bits and holds about 6-7 significant digits
    - Its range is approximately ±3.40282347E+38F
  - a `double` that is 64 bit and holds about 15 significant digits
    - Its range is approximately ±1.79769313486231570E+308

Data Types : Characters

- **Java also contains a `char` type**
  - a `char` is 16 bits and holds a Unicode character - it represents a printable character such as the letter "X"
  - It *can* also be used as an unsigned 16-bit number whose range is 0 to 64,767

Types of Variables

- **In Java, there are three types of variables, depending on *where* and *how* you declare them:**
  - local variables
  - static variables
  - instance variables

- ***Local variables* are declared inside a method**
  - They are created when the method is called and are garbage collected when the method ends
  - Notice that parameters are local variables, too

Example

When `add()` is called, it created the two parameters `num1` and `num2` as well as the variable `sum`. When the `add()` method finishes running, all three variables are destroyed

```
public int add(int num1, int num2) {
   int sum;
   sum = num1 + num2;
   return sum;
}
```

Static Variables

- ***Static* variables are allocated once and retain their values the for life of the program**
  - A static variable cannot be local
  - It must declared in a class
  - Must be declared using the `static` modifier

Example

The `counter` variable is created when the application starts running. It stays around for the life of the application. It can be seen by all of the methods in the `MainApp` class because that is where it is declared. However, because it is private it cannot be accessed outside of the class.

```
public class MainApp {
   private static int counter = 0;

   public static void main(String[] args) {
      counter++;
      ...
   }

   public static void anotherMethod() {
      counter--;
      ...
   }
}
```

Instance Variables

- ***Instance* variables belong to an object**
  - They are created when the object is instantiated and are garbage collected when the object is destroyed

Example

The `name` and `age` variables in the Person class are instance variables. Each time a person object is created, those variables are created for that object and stay allocated until that object goes away.

```
class Person {
   private String name;
   private int age;

   public Person(string name, int age) {
      this.name = name;
      this.age = age;
   }
   ...
}

public class MainApp {
   public static void main(String[] args) {
      Person me = new Person("Dana Wyatt", 63);
      Person you = new Person("John Q Student", 28);
      ...
   }
}
```

- **We will see lots of examples of these as the course continues and we become better Java programmers**

Default Values

- **Java provides default values for *class-level* variables based on their data types**
  - Numbers default to 0
  - Booleans default to `false`
  - Objects default to `null`
- **Knowing this explains why code acts as it does when you forget to initialize variables**
- **However, the compiler does NOT assign *local* variables a default**
  - If you try to access a local variable that is *uninitialized*, the compiler will generate a syntax error
- **A best practice says *always* initialize your variables**

Exercise

EXERCISE 1

Using Notepad declare variables to hold the following data. Consider what information is being stored with each variable, and how you will use the data.

NOTE: this exercise is a mental/paper exercise and **SHOULD NOT** be completed in IntelliJ.

Declare each variable with the correct data types:

- a vehicle identification number in the range 1000000 - 9999999
- a vehicle make /model (i.e. Ford Explorer)
- a vehicle color
- whether the vehicle has a towing package
- an odometer reading
- a price
- a quality rating (A, B, or C)
- a phone number
- a social security number
- a zip code

(you will have a chance to share your answers through zoom)

Section 3–2

Operators and Expressions

String Concatenation

- **Java uses the + operator to concatenate strings**

Example

```
public class BuildAStringApp {

  public static void main(String args[]){
     String word1 = "Hello";
     String word2 = "World";
     String greeting;

     greeting = word1 + " " + word2 + "!";
     System.out.println(greeting);
  }
}
```

OUTPUT

```
Hello World!
```

- **When you Google, you will also find that there is a `concat()` method**

Mathematical Operators and Expressions

- **Java supports the standard arithmetic operators you find in most languages (`+` `-` `*` `/`) and well as the remainder (or *modulo*) operator (`%`)**

- **Because Java is strongly typed, the results of arithmetic operations are sometimes surprising**
  - Note: one integer divided by another is an integer!

Example

```
public class BasicIntegerMathApp {

   public static void main(String[] args){
      int a = 10;
      int b = 3;
      int result;

      result = a + b;
      System.out.println(result);  // displays 13

      result = a - b;
      System.out.println(result);  // displays 7

      result = a * b;
      System.out.println(result);  // displays 30

      result = a / b;
      System.out.println(result);  // displays 3

      result = a % b;
      System.out.println(result);  // displays 1
  }
}
```

Operators and Expressions *cont'd*

- **Floating point math always returns a floating point result**

Example

```
public class BasicFloatingPointMathApp {

   public static void main(String[] args){
      float a = 10;
      float b = 3;
      float result;

      result = a + b;
      System.out.println(result);  // displays 13.0

      result = a - b;
      System.out.println(result);  // displays 7.0

      result = a * b;
      System.out.println(result);  // displays 30.0

      result = a / b;
      System.out.println(result);  // displays 3.3333333

      result = a % b;
      System.out.println(result);  // displays 1.0

      result = b - (a % b);
      System.out.println(result);  // displays 2.0
  }
}
```

- **Parentheses can be placed around expressions to control the order in which they are evaluated**
  - Expressions can be a combination of integer and floating point values, but if the result is a float point value, it must be assigned to a floating point variable

Pre/Post- Increment and Decrement

- **Like other C-based languages, Java embraces the pre/post-increment and decrement operator**

Example

```
int x = 5;
int y;

x++;   // adds 1 to x   (now it is 6)
++x;   // adds 1 to x   (now it is 7)

y = ++x;    // adds 1 to x and then assigns x to y
            // x is now 8 and y is 8

x = 5;

y = x++;    // assigns x to y and then adds 1 to x
            // x is now 6 and y is 5
```

Literals Are Typed

- **In Java, literal values (hard coded values) have a data type**

Example

```
101   <- an int
101L  <- a long (can use a little l, but it looks like a 1)
7.25  <- a double
7.25f <- a float
"Hi"  <- a String
'a'   <- a char
true  <- a boolean
```

- **Starting in Java SE 7, you can place underscore characters (_) between digits in a numerical literal**
  - This sometimes makes the number more readable

Example

```
long ssn = 111_22_3333L;

long creditCard = 5200_7500_6500_0001L;

double loanBalance = 1_225_570.00;
```

- **You can't place underscores:**
  - At the beginning or end of a number
  - Adjacent to a decimal point in a floating point literal
  - Prior to an `F` or `L` suffix

- **When using literals, you must decide whether you want to use 32-bit floating point values…**

Example

```
public class PriceApp {

   public static void main(String args[]){
      int count = 11;
      float unitPrice = 7.12f; //this is an error without the f
      float taxRate = 0.825f;
      float totalCost;

      totalCost = (count * unitPrice) * (1 + taxRate);
      System.out.println(totalCost);
   }
}
```

- **… or 64 bit ones**

Example

```
public class PriceApp {

   public static void main(String args[]){
      int count = 11;
      double unitPrice = 7.12;
      double taxRate = 0.825;
      double totalCost;

      totalCost = (count * unitPrice) * (1 + taxRate);
      System.out.println(totalCost);
   }
}
```

- **Static typing will take some getting used to!**

Widening Issues

- **When you take a variable or expression and assign it to a variable of a wider data type, there are no errors and it is considered a "safe" process**

- **What does wider mean?**
  - It means no loss of precision

```
byte -> short -> char -> int -> long -> float -> double
```

- **For example, assigning a `float` variable to a `double`!**
  - … or an `int` variable to a `long`!

Example

```
int myInt = 9;
long myLong;

float myFloat = 3.8f;
double myDouble;

// an int fits in a long
myLong = myInt;

// a float fits in a double
myDouble = myFloat;

// you can even put an int or long into a float or double
myFloat = myLong;
myDouble = myLong;
```

- **Widening isn't about a specific *value* for a variable and whether it will fit in the new variable**
  - It is about whether any value in the source *data type* will fit in the data type of the new variable

Narrowing Issues

- **When you take a variable or expression and assign it to a narrower variable, the compiler will generate an error**
  - If you want the assignment to happen, you have to explicitly tell the compiler
- **What does narrower mean?**
  - It means there is a "chance" that a value in the source variable might now fit in the narrower variable

`double -> float -> long -> int -> char -> short -> byte`

- **For example, assigning a `double` variable to a `float`!**
  - … or a `float` variable to a `int`!

Example

```
int myInt;
long myLong = 9;

float myFloat;
double myDouble = 123.4567890123;

// an int won't necessarily fit in a long
myInt = myLong;

// a double won't necessarily fit in a float
myFloat = myDouble;

// a float probably won't necessarily fit in an int
myLong = myFloat;
```

SYNTAX ERRORS

```
Main.java:12: error: incompatible types: possible lossy conversion from long to int
myInt = myLong;
        ^
Main.java:15: error: incompatible types: possible lossy conversion from double to float
myFloat = myDouble;
          ^
Main.java:18: error: incompatible types: possible lossy conversion from float to long
myLong = myFloat;
         ^
3 errors
```

Type Casting

- **Sometimes you know that a narrowing assignment is okay and can force the compiler to make the assignment using type casting**

- **To cast an expression to a narrow type, put the target type in parenthesis in front of the expression**
  - It will suppress the compiler error

Example

```
long myLong = 9;
int myInt;

// you know the range of values in the long variable
// will always fit in the int
myInt = (int) myLong;
```

- **This is often needed because you are calling a function whose return type you can't control**
  - For example, you are calling `Math.pow()` that returns a double but you need less precision
    - See more details on the next page

Example

```
float num = 2.2;

// if you don't cast the result, you have to store
// the value in a double variable
float result = (float) Math.pow(num, 2);
```

Java's `Math` Class

- **Java has a `Math` class with many helpful predefined methods**
  - See: `https://www.w3schools.com/java/java_ref_math.asp`
- **The methods are static methods**
  - This means you invoke them using the name of the class

Example

```
double degrees = 90.0;

double answer = Math.sin(degrees);
```

- **Most of the methods have a one or more double parameters and return a double**
  - But read the documentation because this isn't entirely true

Example

```
double value = 1234.567;

int wholeNumber = Math.round(value);
```

Example: Working with the `Math` Class

Example

```
public class MathApp {

   public static void main(String[] args) {

      int natKidCount = 2, brittKidCount = 4;

      int mostKids = Math.max(natKidCount, brittKidCount);

      System.out.println(
        "The biggest family has " + mostKids + " kids");

   }
}
```

Assignment Operators

- **Java supports a set of assignment operators that are combined with arithmetic operators to make certain operations easier to express**

- **They include: `+=` `-=` `*=` `/=` `%=`**

Example

```
// Original technique

int answer = 0;
answer = answer + 10; // 10
answer = answer - 5;  // 5
answer = answer * 10; // 50
answer = answer / 2;  // 25
answer = answer % 10; // 5  (int remainder of 25 / 10)
```

Example

```
// Shortcut technique

int answer = 0;
answer += 10; // 10
answer -= 5;  // 5
answer *= 10; // 50
answer /= 2;  // 25
answer %= 10; // 5  (int remainder of 25 / 10)
```

Exercises

In this exercise you will create a simple project that uses java to perform various mathematical calculations. Complete your work in the `pluralsight/workbook-1` folder.

EXERCISE 2

Using IntelliJ create a new Java project named MathApplication. Add a new package named `com.pluralsight` with a file named `MathApp.java` structured as follows:

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

      // Question 1:
      // declare variables here
      // then code solution
      // then use System.out.println() to display results
      // ex: System.out.println("The answer is " + answer);

      // REPEAT FOR NEXT EXERCISE

   }
}
```

Step 1

Write code to find answers to the following questions.

NOTES: Coding is not just about solving equations, your code should be legible and meaningful

Use comments and line spacing to visually separate each question.

Take your time to think of and use meaningful variable names (don't just use num1)

Print **meaningful** messages to the screen so that the reader has all of the context information for what you are.

QUESTIONS:

1. Create two variables to represent the salary for Bob and Gary, name them `bobSalary` and `garySalary`. Create a new variable named `highestSalary`. Determine whose salary is greater using `Math.max()` and store the answer in `highestSalary`. Set the initial salary variables to any value you want. Print the answer (i.e "The highest salary is …")
2. Find and display the smallest of two variables named `carPrice` and `truckPrice`. Set the variables to any value you want.
3. Find and display the area of a circle whose radius is 7.25
4. Find and display the square root a variable after it is set to 5.0
5. Find and display the distance between the points (5, 10) and (85, 50)
6. Find and display the absolute (positive) value of a variable after it is set to -3.8
7. Find and display a random number between 0 and 1

Step 2

Push your changes to GitHub (always stage, commit and push your changes)

```
1.git add -A

2.git commit -m "completed Mod3 Exercise 2 - MathApp"

3.git push origin main
```

Section 3–3

Writing to the Screen and Reading from the Keyboard

Writing Text to the Screen

- **You can use both the `System.out.print()` and `System.out.println()` methods to display a line of text**
  - The difference is that the `println()` method places a CR/LF in the output stream after it writes

Example

```
class Program
{
   public static void main(String args[])
   {
      System.out.print("Hello ");
      System.out.print("World!");
   }
}
```

OUTPUT

```
Hello World!
```

Example

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

OUTPUT

```
Hello World!
```

- **The two examples above aren't *exactly* the same…**
  - QUESTION: What would happen if you added a `println()` statement to each?

Formatting Output

- **There are several ways to format numbers in Java**
  - https://java2blog.com/format-double-to-2-decimal-places-java/

- **One of the easiest is to use the `String` class' `format()` method**
  - It accepts a format string (more on this on the next page)

Example

```
float subtotal = 22.87;
float tax = subtotal * 0.0825f;
float totalDue = subtotal + tax;

System.out.println(
    "Total due is: " + String.format("%.2f",  totalDue));
```

- **Another way is to use the `printf()` method instead of `print()` or `println()`**
  - It accepts a format string also

Example

```
float subtotal = 22.87;
float tax = subtotal * 0.0825f;
float totalDue = subtotal + tax;

System.out.printf("Total due is: %.2f",  totalDue);
```

Java Format Specifiers

- **Format specifiers begin with a percent character (`%`)**
- **They end with a type character that specifies type of data (`int`, `float`, etc) being formatted**
- **In between, you can specify width and precision**

Syntax

```
% [flags] [width] [.precision] type-character
```

- **Type characters include:**
  - `d` decimal integer (base 10 number)
  - `x` or `X` hexadecimal integer (base 16)
  - `f` floating point number
  - `s` string
  - `c` character

Example

```
int id = 10135;
String name = "Brandon Plyers";
float pay = 5239.77f;

System.out.printf("%s (id: %d) $%.2f", name, id, pay);
```

Reading Input with Scanner

- **You can use the `Scanner` class to read input**
  - A whole line at a time, or
  - One formatted value at a time

- **The `Scanner` class is defined in `java.util.Scanner` and you will have to import it at the top of the class file**
  - You can import just the `Scanner` class or all classes in the same package

Example

```
java.util.Scanner;     // import the Scanner class
import java.util.*;    // import all classes in the package
```

- **If you pass `System.in` as a constructor argument, the Scanner will read from the keyboard**

Example

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

Read a Whole Line of Text

- **Use the `nextLine()` method to read a line of input**
  - Scanner reads the input up to the next line separator (LF or CRLF) and returns it as a String
  - The line separator character is discarded
  - If the scanner is closed, it throws an **`IllegalStateException`**

Example

```
import java.util.*;

class Program
{
   public static void main(String[] args)
   {
      Scanner myScanner = new Scanner(System.in);

      System.out.print("Enter your name: ");
      String name = myScanner.nextLine();

      System.out.println("Howdy " + name);
   }
}
```

Read Individual Values

- **To read individual data values (numbers, words, booleans, etc.) you can use other `Scanner` methods; for example:**
  - `nextInt()`
  - `nextFloat()`
  - `nextDouble()`
  - `nextBoolean()`
- **By default, the scanner breaks the input into *tokens*, separated by *whitespace* (space, tab, line separators, etc.)**
- **Scanner reads the next token and returns its value, *if possible*, as the desired type**
  - Scanner may throw an `InputMismatchException` if the input is not compatible with the requested type
  - *Initial* whitespace is discarded, but *following* whitespace is left in the stream (more on this in a minute)

Example

```
System.out.println("How old are you? ");
int age = scanner.nextInt();

System.out.println("What is your salary? ");
double salary = scanner.nextDouble();
```

Example: A Calculator

Example

```
import java.util.*;

class CalculatorApp
{
   public static void main(String[] args)
   {
      Scanner myScanner = new Scanner(System.in);

      // get two numbers, add them together, and display the sum

      System.out.print("Enter first number: ");
      int num1 = myScanner.nextInt();

      System.out.print("Enter second number: ");
      int num2 = myScanner.nextInt();

      int sum = num1 + num2;

      System.out.println("The sum is " + sum);
   }
}
```

Mixed Input Types

Example

// File: UserInputApp.java

```
import java.util.*;

public class UserInputApp {

   public static void main(String[] args) {

         // create a scanner
         Scanner scanner = new Scanner(System.in);

         // read name
         System.out.print("What is your name? ");
         String name = scanner.nextLine();

         // read age
         System.out.print("How old are you? ");
         int age = scanner.nextInt();

         // display a greeting
         String greeting;
         if (age <= 21) {
            greeting = "Hi ";
         }
         else {
            greeting = "Hello ";
         }
         System.out.println(greeting + " " + name + "!");
    }
}
```

TRACE OF TERMINAL WINDOW SESSION

```
What is your name? Dana
How old are you? 63
Hello Dana!
```

- **It seems to work fine, but we must be careful...**

Buffered Input

- **Most operating systems require you to hit the ENTER key to send input on through to the program**
  - This actually inserts a line separator (Usually CRLF) in the input buffer, which must be handled by the scanner

Example

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

System.out.print("How old are you? ");
int age = scanner.nextInt();
```

TRANSCRIPT OF RUNNING CODE:

```
How old are you? 63↵     ← where the return arrow
                            is the ENTER key (CRLF)
```

- **In this example, `nextInt()` reads 63 correctly, and leaves the following CRLF in the input buffer**

- **Does this cause a problem?**
  - It depends...

Line Separators and Numeric Input

- **If another numeric input method is called (`nextInt()`, `nextFloat()`, etc), it discards the leftover CRLF as whitespace before reading the next token**
  - It's looking for numbers -- so an extra CRLF doesn't affect numeric input

Example

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

System.out.print("Enter number 1: ");
int num1 = scanner.nextInt();

System.out.print("Enter number 2: ");
int num1 = scanner.nextInt();
```

TRANSCRIPT OF RUNNING CODE:

```
Enter number 1: 13↲
Enter number 2: 145↲     ← This ignores the CRLF left in the
                           buffer after reading 13 and
                           successfully reads 145

        // HOWEVER, now there is a different CRLF
        // left in the buffer after 145
```

Line Separators and nextLine()

- **If the next read is performed with `nextLine()`, it stops at the first CRLF it finds!**

Example

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

System.out.print("Enter number 1: ");
int num1 = scanner.nextInt();

System.out.print("Enter number 2: ");
int num2 = scanner.nextInt();

System.out.println("What do you want to do? ");
System.out.print("   Enter 'add' or 'subtract': ");
String action = scanner.nextLine();

System.out.println("Preparing to do math... ");
```

TRANSCRIPT OF RUNNING CODE:

```
Enter number 1: 13↲
Enter number 2: 145↲
What do you want to do?
   Enter 'add' or 'subtract': Preparing to do math...

   // The system didn't wait for the user to enter
   // data because the CRLF in the buffer after 145
   // marked the end of the input when nextLine()
   // went to read its data
```

Dealing With Line Separators

- **So how do you fix this?**
  - Carefully and on a case-by-case basis
- **If you read a mix of numbers and text, whose job is it to get rid of the extra CRLF???**
  - The code reading the number (that left CRLF in the buffer)?
  - The code reading text (that doesn't want a CRLF before its read)?
- **But --**
  - If the code reading an int gets rid of the CRLF, it would be wasted energy if the next piece of code also reads an int
  - If the code reading text tries to get rid of the CRLF and its not there, then that could be a problem
- **There's no great answer! But someone has to decide. So, here is my answer for the code above…**
  - If will forcibly consume the CRLF that is causing the problem!

Consuming a Line Separator

Example

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

System.out.print("Enter number 1: ");
int num1 = scanner.nextInt();

System.out.print("Enter number 2: ");
int num2 = scanner.nextInt();
scanner.nextLine();   // "eat" the leftover CRLF

System.out.println("What do you want to do? ");
System.out.print("   Enter 'add' or 'subtract': ");
String action = scanner.nextLine();

System.out.println("Preparing to do math... ");
```

TRANSCRIPT OF RUNNING CODE:

```
Enter number 1: 13↵
Enter number 2: 145↵
What do you want to do?
   Enter 'add' or 'subtract': add↵
Preparing to do math...
```

Exercises

In these exercise you will create basic console applications which prompt the user for information and then perform calculations based on the user input. Remember to do your work in the `workbook-1` directory.

Remember to be conscious of the format and readability of your code.

* Use meaningful variable names
* Use whitespace and lines to organize your code and thoughts so that others can easily understand your code
* Don't try to do multiple things on a single line of code - when in doubt opt for code readability

EXERCISE 3

Create a Java application named BasicCalculator that reads in two floating point numbers and then asks the user whether they want to add, subtract, multiply or divide the two numbers.

Step 1

Perform the requested operation and display the results.

```
Enter the first number: 5
Enter the second number: 12

Possible calculations:
  (A)dd
  (S)ubtract
  (M)ultiply
  (D)ivide
Please select an option: M

5 * 12 = 60
```

Step 2

Push your changes to GitHub

```
git add -A
git commit -m "Completed Mod3 - Exercise 3 - Basic Calculator"
git push origin main
```

EXERCISE 4

Step 1

Create a Java application named PayrollCalculator that prompts the user to enter:

- their name
- their hours worked (a floating point number)
- their pay rate (a floating point number)Calculate their gross pay.

Display the employee's name and their gross pay.

Step 2

Push your changes to GitHub

```
git add -A
git commit -m "Completed Mod3 - Exercise 4 - Payroll Calculator"
git push origin main
```

(Optional)

You learned about `if` statements in your pre-work, so although we have not discussed them in class, you should be able to figure out how to pay 1.5x overtime after 40 hours. You can use W3Schools guide as a reference https://www.w3schools.com/java/java_conditions.asp

Push your changes to GitHub

```
git add -A
git commit -m "Updated Payroll Calculator with overtime pay"
git push origin main
```

Section 3–4

Static Methods

Static Methods

- **You can add additional static methods to your application class besides the required `main()` method**

- **Static methods do not require us to instantiate the class**
  - Here as we are first learning Java, they allow us to break our application logic up into more manageable "chunks"
  - Eventually, we will transition to fully object-oriented coding

- **Since the class may not be instantiated, a static method is not allowed to access instance variables (!)**
  - It can only access local variables or static variables

Declaring a static Method

- **If a method doesn't return anything, it uses the word `void` as the return type**

Example

MainApp.java

```
public class MainApp {

   public static void main(String[] args) {
      foo();
      moo();
   }

   public static void foo() {
      System.out.println("Foo");
   }

   public static void moo() {
      System.out.println("Moo");
   }
}
```

OUTPUT

```
Foo
Moo
```

Passing Data to Methods

- **You can pass data (called arguments) to a method in Java**
  - They are received in parameters in the order they are passed
  - Arguments and parameters don't have to have the same name, but they should be of the same type

Example

MainApp.java

```
public class MainApp {

   public static void main(String[] args) {
      int a = 4, b = 9, c = 10, d = 3;

      addAndDisplay(a, b);
      addAndDisplay(b, c);
      addAndDisplay(a, d);
   }

   public static void addAndDisplay(int num1, int num2) {
      int sum = num1 + num2;
      System.out.printf("%d + %d = %d", num1, num2, sum);
   }
}
```

OUTPUT

```
4 + 9 = 13
9 + 10 = 19
4 + 3 = 7
```

Returning Data from Methods

- **Java requires you to specify the type of data that will be returned from a method**
  - Instead of `void`, you would specify the data type of the value being returned

Example

MainApp.java

```
public class MainApp {

   public static void main(String[] args) {
      int a = 4, b = 9, c = 10, d = 3;
      int sum;

      sum = addAndDisplay(a, b);
      display(a, b, sum);

      sum = addAndDisplay(b, c);
      display(b, c, sum);

      sum = addAndDisplay(a, d);
      display(a, d, sum);
   }

   public static int add(int num1, int num2) {
      int sum = num1 + num2;
      return sum;
   }

   public static void display(int num1, int num2, int sum) {
      System.out.printf("%d + %d = %d", num1, num2, sum);
   }
}
```

OUTPUT

```
4 + 9 = 13
9 + 10 = 19
4 + 3 = 7
```

Using a `Scanner` in Multiple Methods

- **If you want to read data from the keyboard in multiple methods, you have three choices**
- **You could declare a new `Scanner` object in each method**
  - It is easy to copy and paste the Scanner declaration over and over
  - This leads to duplicate code (yuck!)
- **You could declare one `Scanner` object at the class level**
  - However, because it is accessed by static methods, the Scanner object must be assigned to static instance
  - A static class member is shared by all instances of the class
- **You could pass the `Scanner` object from `main` to any method that needs to use it**
  - This requires an extra parameter but is fairly easy

Example: Re-declaring the Scanner

Example

```
import java.util.*;

public class InputApp {

   public static void main(String[] args) {
     Scanner scanner = new Scanner(System.in);
     System.out.print(
       "What do you want to do (1-add, 2= subtract) ? ");

     int command = scanner.nextInt();
     if (command == 1) {
         doAdd();
     }
     else if (command == 2) {
         doSubtract();
     }
     else {
         System.out.printf(
            "%d -- Invalid command!", command);
     }
  }

   public static void doAdd() {
     Scanner scanner = new Scanner(System.in);

     System.out.print("Enter 1st number: ");
     double num1 = scanner.nextDouble();

     System.out.print("Enter 2nd number: ");
     double num2 = scanner.nextDouble();

     double sum = num1 + num2;
     System.out.printf("%f + %f = %f", num1, num2, sum);
   }

   public static void doSubtract() {
     Scanner scanner = new Scanner(System.in);

     System.out.print("Enter 1st number: ");
     double num1 = scanner.nextDouble();
```

Example: Re-declaring the Scanner *cont'd*

```
      System.out.print("Enter 2nd number: ");
      double num2 = scanner.nextDouble();

      double difference = num1 - num2;
      System.out.printf(
         "%f + %f = %f", num1, num2, difference);
   }
}
```

TRACE OF TERMINAL WINDOW SESSION

```
What do you want to do (1-add, 2= subtract) ? 1

Enter 1st number: 25.25
Enter 2nd number: 5.25
25.25 + 5.25 = 30.500000
```

Example: Declaring the Scanner at the Class-Level

Example

```
import java.util.*;

public class InputApp {

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

   public static void main(String[] args) {
    System.out.print(
       "What do you want to do (1-add, 2= subtract) ? ");

     int command = scanner.nextInt();
     if (command == 1) {
         doAdd();
     }
     else if (command == 2) {
         doSubtract();
     }
     else {
         System.out.printf(
            "%d -- Invalid command!", command);
     }
  }

   public static void doAdd() {
     System.out.print("Enter 1st number: ");
     double num1 = scanner.nextDouble();

     System.out.print("Enter 2nd number: ");
     double num2 = scanner.nextDouble();

     double sum = num1 + num2;
     System.out.printf("%f + %f = %f", num1, num2, sum);
   }

   public static void doSubtract() {
     System.out.print("Enter 1st number: ");
     double num1 = scanner.nextDouble();
```

Example: Declaring the Scanner at the Class-Level *cont'd*

```
      System.out.print("Enter 2nd number: ");
      double num2 = scanner.nextDouble();

      double difference = num1 - num2;
      System.out.printf(
         "%f - %f = %f", num1, num2, difference);
   }
}
```

TRACE OF TERMINAL WINDOW SESSION

```
What do you want to do (1-add, 2= subtract) ? 2

Enter 1st number: 25.25
Enter 2nd number: 5.25
25.25 - 5.25 = 20.000000
```

Example: Passing the Scanner

Example

```
import java.util.*;

public class InputApp {

   public static void main(String[] args) {
     Scanner scanner = new Scanner(System.in);
     System.out.print(
       "What do you want to do (1-add, 2= subtract) ? ");

     int command = scanner.nextInt();
     if (command == 1) {
         doAdd(scanner);
     }
     else if (command == 2) {
         doSubtract(scanner);
     }
     else {
         System.out.printf(
            "%d -- Invalid command!", command);
     }
  }

   public static void doAdd(Scanner scanner) {

     System.out.print("Enter 1st number: ");
     double num1 = scanner.nextDouble();

     System.out.print("Enter 2nd number: ");
     double num2 = scanner.nextDouble();

     double sum = num1 + num2;
     System.out.printf("%f + %f = %f", num1, num2, sum);
   }

   public static void doSubtract(Scanner scanner) {

    System.out.print("Enter 1st number: ");
     double num1 = scanner.nextDouble();
```

Example: Passing the Scanner *cont'd*

```
      System.out.print("Enter 2nd number: ");
      double num2 = scanner.nextDouble();

      double difference = num1 - num2;
      System.out.printf(
         "%f - %f = %f", num1, num2, difference);
   }
}
```

TRACE OF TERMINAL WINDOW SESSION

```
What do you want to do (1-add, 2= subtract) ? 1

Enter 1st number: 25.25
Enter 2nd number: 5.25
25.25 + 5.25 = 30.500000
```

Exercises

In this exercise you will refactor the code from your previous payroll calculator exercise. Refactoring is the process of updating existing code and breaking it into smaller methods to make it more modular and easier to maintain.

EXERCISE 5 (Optional)

Before you start making changes, make a plan to consider how you will break up your code. Use your notebook to diagram your plan. Talk to the people in your breakout room to discuss ideas about how you might do it.

Step 1

Code your changes

When you finish, we will get together as a class and examine some of the different approaches people used.

Step 2

Push your changes to GitHub

```
git add -A
git commit -m "Completed Mod3 - Exercise 3 - Basic Calculator"
git push origin main
```

Module 4

Conditionals

Section 4–1

Conditionals

Conditionals

- **Conditional logic lets us do different things depending on the value of an expression**

- **Conditionals include:**
  - `if` and `if/else` statements
  - `switch` statements
  - conditional operator (sometimes called the ternary operator)

`if` Statement

- **The if statement executes some code if a condition expression evaluates as `true`**
  - The condition is surrounded by parentheses
  - Braces are only *required* around the code if there is more than one statement, but best practice is to always have them

Syntax

```
if (condition) {
   // statement(s)
}
```

- **The condition is often expressed by comparing values:**

== **!=** **<** **<=** **>** **>=**

  - The equality and inequality operators can be used with both primitive types and objects
  - The other comparison operators only work with primitive types that can be represented in numbers

Example

```
String name = "Ezra";
int age = 17;

double price = 25.00;

if (age < 18) {
   price = price * .9;
}

System.out.println("Price of admission: " + price);
```

if / else

- **The `else` statement can be used to write alternative code that executes when the condition is false**
  - If the else condition has more than one statement, you must surround them with curly braces
  - `else` must always follow an `if`

Example

```
String name = "Ezra";
int age = 17;

double price;

if (age < 18) {
   price = 22.50;
}
else {
   price = 25.00;
}

System.out.println("Price of admission: " + price);
```

`if / else` Statements *cont'd*

- **You can string together `if / else if / else` statements for more complex logic**

Example

```
String name = "Ezra";
int age = 17;

double price;

if (age < 18) {
   price = 18.00;
}
else if (age < 65) {
   price = 25.00;
}
else {
   price = 18.00;
}

System.out.println("Price of admission: " + price);
```

- **You can build complex conditions by using the logical operators: `&&` `||` `!`**

Example

```
String name = "Ezra";
int age = 17;

double price;

if (age < 18 || age >= 65) {
   price = 18.00;
}
else {
   price = 25.00;
}
```

Comparing Strings

- **The == operator tests whether both operands are the same (identical) object**
  - This doesn't work reliably for Strings, since the string you read as input is a different object from the one that holds a literal value

- **The String class' `equals()` method tests whether the strings contain the same characters in the same order**

Example

```
String homeState = "Texas";
String contactPhone;

if (homeState.equals("Texas") || homeState.equals("Kansas")) {
   contactPhone = "800-555-5555";
}
else {
   contactPhone = "855-555-5555";
}
```

Example: Better Calculator?

Example

```
import java.util.*;

public class InputApp {

   public static void main(String[] args) {

     Scanner scanner = new Scanner(System.in);
     System.out.print(
       "What do you want to do (add, subtract) ? ");

     String command = scanner.nextLine();
     if (command.equals("add")) {
         doAdd(scanner);
     }
     else if (command.equals("subtract")) {
         doSubtract(scanner);
     }
     else {
         System.out.printf(
            "%s -- Invalid command!", command);
     }
  }

   public static void doAdd(Scanner scanner) {
     System.out.print("Enter 1st number: ");
     double num1 = scanner.nextDouble();

     System.out.print("Enter 2nd number: ");
     double num2 = scanner.nextDouble();

     double sum = num1 + num2;
     System.out.printf("%f + %f = %f", num1, num2, sum);
   }

   public static void doSubtract(Scanner scanner) {
     System.out.print("Enter 1st number: ");
     double num1 = scanner.nextDouble();

     System.out.print("Enter 2nd number: ");
     double num2 = scanner.nextDouble();

     double difference = num1 - num2;
     System.out.printf(
        "%f + %f = %f", num1, num2, difference);
   }
}
```

Conditional Operator

- **The conditional operator (? :) in Java is like a shorthand version of `if/else`**
  - A similar operator is found in all C-family languages

Syntax

```
variable = (condition) ? expressionTrue : expressionFalse;
```

Example

```
String name = "Ezra";
int age = 17;

double price = (age < 18) ? 22.50 : 25.00;

System.out.println("Price of admission: " + price);
```

Exercises

In these exercises you will create a Sandwich Shop application. You will allow users to order a simple sandwich and you will calculate the price. The customer may receive a discount based on their age.

Remember to create your projects in the `pluralsight/java-development/workbook-1` folder.

EXERCISE 1

Create a Java application named SandwichShop. This will be a point of sales application to calculate the cost of a sandwich.

Step 1

Prompt the user for the size of the sandwich (1 or 2):

- 1: Regular: base price $5.45
- 2: Large: base price $8.95

Prompt the user for their age:

- Student (17 years old or younger) – receive a 10% discount
- Seniors (65 years old or older) – receive a 20% discount

Display the cost of the sandwich to the screen

Step 2

Push your changes to GitHub

```
git add -A
git commit -m "Completed Mod4 - Exercise 1 - Sandwich Shop"
git push origin main
```

EXERCISE 2

Modify the SandwichShop from the last exercise to allow customers to order a "loaded" sandwich (double everything).

Step 1

Prompt the user for the size of the sandwich (1 or 2):

- 1: Regular: base price $5.45
- 2: Large: base price $8.95

Prompt the user if they would like the sandwich "loaded" (yes/no). If so there is an additional cost for a loaded sandwich

- Regular: $1.00
- Large: $1.75

Prompt the user for their age:

- Student (17 years old or younger) – receive a 10% discount
- Seniors (65 years old or older) – receive a 20% discount

Display the cost of the sandwich to the screen

Step 2

Push your changes to GitHub

```
git add -A
git commit -m "Added Loaded option for the sandwich shop"
git push origin main
```

Section 4–2

The `switch` Statement

`switch` Statement

- **The `switch` statement evaluates an expression and executes the code in a matching `case` block**
  - If there is no match, the `default` block executes
  - Code in the case executes until a break statement is encountered or the switch statement ends

Syntax

```
switch(expression) {
  case value-1:
    // code block
    break;
  case value-2:
    // code block
    break;
  default:
    // code block
}
```

`switch` Example

- **Switch statements can get pretty long if the expression has many unique values**
  - It's still easier to read than the equivalent if/else construct

Example

```
int dayNumber = 3;
String description = "";

switch (dayNumber) {
  case 0:
    description = "Sunday";
    break;
  case 1:
    description = "Monday";
    break;
  case 2:
    description = "Tuesday";
    break;
  case 3:
    description = "Wednesday";
    break;
  case 4:
    description = "Thursday";
    break;
  case 5:
    description = "Friday";
    break;
  case 6:
    description = "Saturday";
    break;    // optional break -- okay without it
}
```

The break statement

- **If a `case` block doesn't end with a `break`, then code falls into the next block**
  - This is intentional; it can make your switch statement more compact when several values lead to the same behavior

Example

```
int dayNumber = 3;
String description = "";

switch (dayNumber) {
  case 0:
  case 6:
    description = "Weekend";
    break;
  case 1:
  case 2:
  case 3:
  case 4:
  case 5:
    description = "Weekday";
    break;
}
```

Exercises

In this exercise you will create a CLI application for a rental car company. You will prompt the user to answer questions about their selected options and then calculate the cost of the rental.

You are not required to use only `switch` statements in this exercise. You can choose to use `if/else` or `switch` statements as you deem appropriate.

EXERCISE 1

Step 1

IMPORTANT: DO NOT just start coding - take time to plan the flow of your application in your notebook.

Read the requirements below and consider how you would calculate the cost if you did not have a computer. Practice several scenarios (with different options) in your notebook before you create your project in IntelliJ.

Take a picture or screenshot of your notebook and save the image to your project directory (after you create the project).

Step 2

Create a Java application named RentalCarCalculator that estimates the cost of reserving a rental car. Prompt the user for the following information:

- pickup date (store as a string)
- number of days for the rental
- whether they want an electronic toll tag at $3.95/day (yes/no)
- whether they want a GPS at $2.95/day (yes/no)
- whether they want roadside assistance at $3.95/day (yes/no)
- their current age

Calculate and display:

- basic car rental
- options cost
- underage driver surcharge
- total cost

The basic car rental is $29.99 per day. There is a 30% surcharge on the basic car rental for drivers under 25. All taxes have already been incorporated into the fees shown.

Step 3

Push your changes to GitHub

```
git add -A
git commit -m "Completed Mod4 - Exercise 1 - Rental Car Calculator"
git push origin main
```

Financial Calculators

Workbook 1's Workshop

What Are Workshops?

At the end of each week of instruction, you will code a workshop that resembles a 1 to $1^{1}/_{2}$ day mini-capstone. Its goal is to reinforce everything you learned about Java during the week in a larger project.

You will create a new GitHub repository for each of your workshop projects. You should clone the repo to the `C:/pluralsight/java-development` directory.

These projects will all be console (CLI) projects. In other words, you will not be creating a fancy UI (user interface) such as a website. The primary concern of each workshop is for you to complete the requirements in Java. Even though the UI is not the primary concern, you should still consider the formatting of your screen, and try to make the application clean and intuitive for the user.

Remember to `commit` and `push` your code frequently (don't just wait until the project is complete).

Project Description

Create a new GitHub project named `financial-calculators`, and clone it to your `java-development` directory.

Read all of the project requirements before you begin to code. Use your notebook to plan your project. Use the links provided in the Hints section to help research how each calculator should work.

The Project

You will build an application for a financial organization that wants to provide a set of financial calculators for their clients. The screen will prompt the user to select which calculator they would like to use.

They are interested in having you implement as many of the following calculators you can in the time allotted. Expectations: Getting two done would be good, getting three done would be great.

- **Calculator 1: A mortgage calculator** - it is used to calculate out how much a monthly payment for a loan would be (minus any insurance or taxes), as well as how much interest you would pay over the life of the loan.

    a. It would accept the principal, interest rate, and loan length from the user
    b. It would display the expected monthly payment and total interest paid

    Example: A $53,000 loan at 7.625% interest for 15 years would have a $495.09/mo payment with a total interest of $36,115.99

    This calculator would use a compounded interest formula.

- **Calculator 2:** A calculator that determines the **future value** of a one-time deposit assuming compound interest - it is used to help you decide how much a CD will be worth when it matures

    a. It would accept the deposit, interest rate, and number of years from the user
    b. It would display the future value and the total interest earned

    Example: If you deposit $1,000 in a CD that earns 1.75% interest and matures in 5 years, your CD's ending balance will be $1,092.62 and you would have earned $92.62 in interest

    Note: The numbers above assume *daily* compounding

- **Calculator 3:** A calculator that determines the **present value** of an ordinary annuity. **(Note: this is difficult)**
    a. It would accept the monthly payout, expected interest rate, and years to pay out from the user
    b. It would display the present value of that annuity

    Example: To fund an annuity that pays $3,000 *monthly* for 20 years and earns an expected 2.5% interest, you would need to invest $566,141.46 today.

    NOTE: If your results on any of these calculators are off by a few pennies (not dollars!), don't worry. The difference is likely attributable to rounding and we aren't that concerned about it in this academy.

Hints

Although you are not creating a website, these links will give you an idea about what the each calculator does:

https://www.bankrate.com/calculators/managing-debt/annual-percentage-rate-calculator.aspx

https://www.nerdwallet.com/article/banking/cd-calculator

https://financialmentor.com/calculator/present-value-of-annuity-calculator

A quick Google will reveal the formulas needed to make each calculator work or you can read up on some of the formulas at:

Mortgage Payment: https://www.quora.com/How-is-the-division-of-principal-vs-interest-calculated-on-mortgage-payments

Future Value: https://www.gobankingrates.com/banking/cd-rates/how-calculate-cd-accounts-value

Note: the n in the formula stands for how often the compounding occurs; use 365x per year

Present Value of Annuity: http://www.1728.org/annuity-presval-formulas.htm 3

What Makes a Good Workshop Project?

- **You should:**
  - Have a clean and intuitive user interface (give the user clear instructions)
  - Implement at least the first two calculators

- **You should adhere to best practices such as:**
  - Create a Java Project that follows the Maven folder structure
  - Create a Java package for your classes
  - Use good variable naming conventions (camelCasing, meaningful variable names)
  - Have clean formatted code that is easy to understand
  - use Java comments effectively

- **Make sure that:**
  - Your code is free of errors and that it compiles

- **Include a README for the project**
  - A README.md file describes your project and should include screen shots of
    * your home screen
    * EACH of the calculator screen you build that shows user input prompts and correct outputs
    * one calculator page that shows erroneous inputs and an error message.
  - ALSO make sure to include one interesting piece of code and a description of WHY it is interesting.