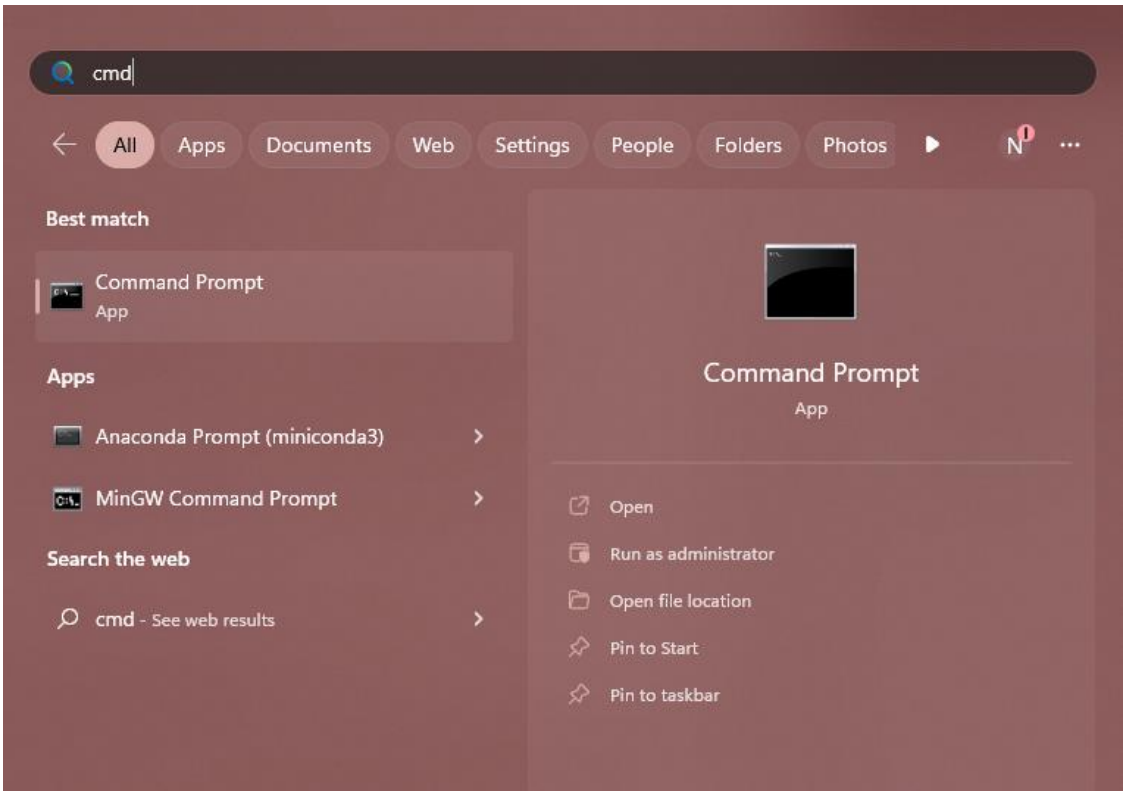
Write Once Run Anywhere

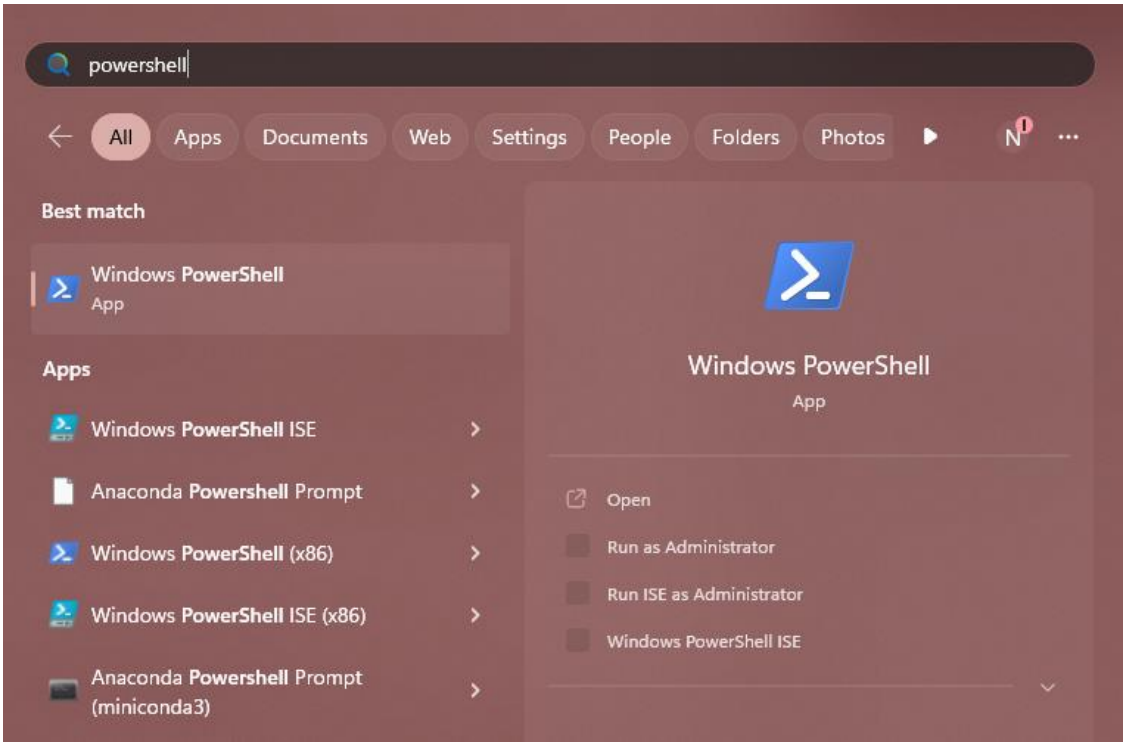
Lab setup

- You will need a text editor and a command line shell. Both are available in most operating systems.
- Eclipse can act as both of these by creating a plain project and using the built-in terminal emulator, but some of Eclipse's functions will still try to work (and fail in a disruptive way) and we would like to demonstrate development in a non-IDE setting so we will not be using Eclipse for this lab.
- If you don't want to use your system's default text editor, you can download and install Visual Studio Code for this lab. VS Code has a built-in terminal emulator too.

On Windows, you can use either `cmd` or `Powershell`. You can find either using the Search bar in the Start Menu.



cmd in the Start Menu search



powershell in the Star Menu search

On macOS, you will use the Terminal app, which will open a `zsh` shell.

On different Linux distros, the default terminal emulator might be called Terminal or Console and will probably open a `bash` shell.

```java
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        System.out.println("Enter your name");
        try (Scanner scan = new Scanner(System.in)) {
            String name;
            name = scan.nextLine();

            System.out.printf("Hello there, %s\n", name);
        }
    }
}
```

Figure 1 Hello World program

Command line basics

We will use the following simple Hello World program to demonstrate building a Java program using the command line. Save this code in a folder you will use in this lab.

1. Open a command-line shell.

   The two main commands we will be using are `java` and `javac`. You should already have these installed from when you installed the JDK. To double check, run the commands with the `--version` option.

   Note: Commands you should run in the terminal from this document will begin with "$ ". You should run these commands without the "$ " part. Example output of these commands will follow immediately after the command without "$ ".

   ```
   $ java --version
   java 17.0.7 2023-04-18 LTS
   Java(TM) SE Runtime Environment (build 17.0.7+8-LTS-224)
   Java HotSpot(TM) 64-Bit Server VM (build 17.0.7+8-LTS-224, mixed mode,
   sharing)
   $ javac --version
   javac 17.0.7
   ```

   The output of these commands should show the major version number of your installed JDK (which should the current LTS) if they are installed properly.

2. Navigate to the folder you have saved your Java source code in.

   Use the `pwd`, `cd` and `ls` or `dir` commands to reach the appropriate directory.

   If you opened your command line from a Start Menu or similar, it probably will have opened in your home directory. Run the `pwd` command to see where you currently are in the directory tree.

   ```
   $ pwd
   Path
   ----
   C:\Users\natha
   ```

   In my case, I am currently in my Windows home directory. I have saved the code file for this lab in the directory with the absolute path `"C:\Users\natha\OneDrive - Old Dominion University\cs251_javalabs\Lab22"`.

   If you know the full absolute path of your directory or the path relative to your current working directory, you can `cd` directly into it. Otherwise, you can use `ls` or `dir` to list the contents of your current working directory and use `cd` to change directories until you reach your file.

```
$ ls
<contents of my home directory>
OneDrive - Old Dominion University
<rest of the contents of my home directory>
$ cd "OneDrive - Old Dominion University"
$ pwd
Path
----
C:\Users\natha\OneDrive - Old Dominion University
```

If any of the names in your path have a blank space, you will need to surround the path in quotes or the shell will interpret them separately.

Building and running a Java program from the command line

Now that you are in your project directory, we can build the program.

`javac` is the command for the Java compiler. Running `javac` with the `--help` option will show all the possible options. We will focus on only compiling .java files into .class files.

Run `javac` and pass the "Main.java" file as the argument then run `ls`. `ls` should now be showing a new file in the current working directory called "Main.class" If there are any compilation errors, `javac` will fail to produce the .class file and instead list the compilation errors.

```
$ javac Main.java
$ ls
Main.class
Main.java
```

Now that you have compiled the program you can run it using the `java` command and passing the name of the main class as an argument. For our example, the main class is called Main.

```
$ java Main
Enter your name
Mark
Hello there, Mark
```

Java programs with packages

We've built a Java program from the command line with only the main class in the default package. How do we build one that has packages?

We will add two more files to this project, both within a package called `pkg`. Remember that packages are directories in a file system. We need to add these files in the subdirectory `pkg`.

You can create a directory from the command line using `mkdir`.

```
$ mkdir pkg
```

Add the classes in Figure 2 and Figure 3 to your project.

```java
package pkg;

public class Planet {

    private String name;
    public double gravity;

    public Planet(String name, double gravity) {
        this.name = name;
        this.gravity = gravity;
    }

    public String getName() {
        return name;
    }

    public double getGravity() {
        return gravity;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setGravity(double gravity) {
        this.gravity = gravity;
    }

    @Override
    public String toString() {
        return String.format("%s: g = %.2f", name, gravity);
    }
}
```

Figure 2 Planet class

Add code to Main.java that uses these two classes. Don't forget to import the classes.

You can build the program with *either* of the following commands.

```
$ javac Main.java pkg/Planet.java pkg/SolarSystem.java
$ javac Main.java
```

```java
package pkg;

public class SolarSystem {
    private Planet[] planets;

    public SolarSystem(int numPlanets) {
        planets = new Planet[numPlanets];
    }

    public Planet getPlanet(int idx) {
        return planets[idx];
    }

    public void setPlanet(Planet p, int idx) {
        planets[idx] = p;
    }
}
```

Figure 3 SolarSystem class

The second command works because the compiler can find the other two classes because they are used within the main class.

If you look in the `pkg` subdirectory, you will find the class files for the other two classes.

You can use the `-d` option to choose where to place the generated class files. Build the program using the `-d` option to place the class files in a bin directory. `cd` into the `bin` directory and try to run the program.

```
$ javac Main.java -d bin
```

It will most likely run fine until it reaches the point when it uses the `Planet` or `SolarSystem` class when it will encounter a runtime error because it cannot find either of the classes. Build the two classes in the package and place their class files in the bin directory.

```
$ javac pkg/Planet.java pkg/SolarSystem.java -d bin
```

The bin directory will now contain the class files for `Planet` and `SolarSystem`. They will be in a subdirectory of bin called pkg and running the program in bin will now work.

Creating a JAR file from the command line

A JAR file is a file format that allows you to bundle multiple files together in a single archive file. JAR files typically contain class files and any auxiliary resources required for the application. Executable JAR files include an entry point for the application which is usually the main class. Create the JAR file from your program with the following command. First `cd` into the bin directory then run the following command.

```
$ jar cvfe WORA.jar Main Main.class pkg/Planet.class pkg/SolarSystem.class
```

Creating a JAR file with this command will automatically generate a manifest file which is packaged with and describes the content of the JAR file. It is also possible to specify your own manifest file if you need more control over the creation of the JAR file. We will not cover this in detail here, however.

You can run your executable JAR file using the `java` command with the `jar` option.

```
$ java -jar WORA.jar
```

Run Anywhere

Now that you've built your program and created a JAR file, we can test the "run anywhere" claim. Your program should run anywhere that has a compatible Java Runtime Environment (JRE).

We will be transferring the JAR file you created to the ODUCS remote Linux machines. Alternatively, if you have been working through this lab on the remote machines, you can transfer it to your local machine.

Use SFTP to transfer your JAR file to the Linux machines.

```
$ sftp yourCSLogin@linux.cs.odu.edu
```

You'll find yourself in a prompt like this:

```
sftp >
```

Here you can use `cd` to move into the directory you want to transfer your JAR file. You can also use `mkdir` if you want to create a new directory to transfer the file into. If you don't already have a cs251 directory, go ahead and make one and `cd` into it.

```
$ sftp > mkdir cs251
```

```
$ sftp > cd cs251
```

You can transfer a file from your local machine to the remote machine using `put`.

```
$ sftp > put WORA.jar
```

If this doesn't work, you can double-check the filename and your current working directory on your local machine by prepending the `ls` command with 'l'.

```
$ sftp > lls
```

If you find you are not in the directory with your JAR file use `lcd` to navigate to the directory.

```
$ sftp > lcd <directory-name>
```

Once you've uploaded your JAR file, you can end the SFTP session using `exit`, `quit`, or (pers onal favorite) `bye`.

```
$ sftp > bye
```

More information on SFTP and a GUI alternative can be found here: https://www.cs.odu.edu/~zeil/cs252/latest/Public/ftp/index.html#secure-file-transfer-sftp-and-scp

Now ssh back into the remote machine, navigate to your JAR file and try running it.

```
$ ssh yourCSLogin@linux.cs.odu.edu
$ cd cs251
$ java -jar WORA.jar
```

There should be no issues and it should run as expected.