1.1. Why Learn Java?

This course is an introduction to **Java** and the Spring Boot framework. Java is a powerful programming language and one of the most widely used across the globe. This course is designed for learners who are already familiar with at least one programming language. Thus, we'll move quickly through the syntax rules and procedural basics of the language (ie, how does one write a for loop in Java? a conditional statement?).

As you've likely heard before, once you grasp the fundamentals of programming in one language, learning another becomes much easier. Professional software developers often work in environments involving several programming languages so it's wise to start learning new ones early on in your career.

Beyond the basics of programming in Java, you'll learn some key *object oriented* concepts to enhance the capabilities of your programs. You'll also download a program called **IntelliJ IDE** to get familiar with the tools of the Java developer. In the final lessons of this course, you will use a Java framework called **Spring Boot**. This framework gives us the scaffolding necessary to create *MVC* programs, another concept we'll cover, effectively and efficiently.

1.2. Setup For Java

For the entirety of this course, we will be coding in Java. Besides installing Java on your machine, you must also add some support technologies to allow you to run and edit Java code.

1.2.1. Java Development Kit

Installing Java means downloading a package of software called the **Java Development Kit**, or **JDK**, for short. The JDK contains software the tools needed to develop and run Java code, namely the the Java compiler, **javac**, and the **Java Virtual Machine (JVM)**.

While the compiler is responsible for processing Java code into machine readable code, the JVM allows us to run that code on any computer. These tools together, downloaded as the JDK, give us the means to write, compile, and run Java on our machines.

A step-by-step walk-through of the process:

1. We write code in Java,
2. The code is passed through the compiler program,
3. The compiler translates Java into **bytecode**, a language readable by the JVM.
4. In the JVM, bytecode is translated to machine code,
5. Your computer then reads and executes the machine code.

The JVM gives Java more flexibility than other compiled programming languages because it will translate bytecode into the appropriate machine code, depending on the operating environment.

1.2.2. Install the JDK

Open a terminal window on your machine and enter the following command:

```
java -version
```

If the response returns a version 13 or higher, you can move on to the section below, Java in the Terminal.

If you do not have a version of Java at 13 or higher, you can download it here. The relevant install link is on the bottom of the page:

Java SE Development Kit 13

You must accept the Oracle Technology Network License Agreement for Oracle Java SE to download this software.

○ **Accept License Agreement** ◉ **Decline License Agreement**

Product / File Description	File Size	Download
Linux	155.95 MB	jdk-13_linux-x64_bin.deb
Linux	163.02 MB	jdk-13_linux-x64_bin.rpm
Linux	179.97 MB	jdk-13_linux-x64_bin.tar.gz
mac OS	173.33 MB	jdk-13_osx-x64_bin.dmg
mac OS	173.68 MB	jdk-13_osx-x64_bin.tar.gz
Windows	159.82 MB	jdk-13_windows-x64_bin.exe
Windows	178.97 MB	jdk-13_windows-x64_bin.zip

Install Java

To install, you must first select *Accept License Agreement*, then select any of the file type options for your operating system.

Tip

- Mac users, we reccommend the `.dmg` option
- Windows users, we reccommend the `.exe` option

Once you have completed the installation steps, return to your terminal and enter `java -version` again to ensure Java has been installed.

1.2.3. Java in the Terminal

Let's write a simple "Hello, World" program and watch the JDK in action.

In the future, we'll be doing most of our Java coding with the IntelliJ IDE. IntelliJ contains many features to help us write Java properly and easily, including its own compiler. For now though, we'll use a simpler text editor so we can demonstrate what we get with the JDK.

In the text editor of your choice, create and save a file called `HelloWorld.java` and include the code below:

```
public class HelloWorld {

   public static void main(String[] args) {

      System.out.println("Hello, World");
   }

}

```

We'll discuss the syntax of this program soon, but you can likely trust your gut that this program has an expected output of "Hello, World". To test this hypothesis, open a terminal window and navigate to the parent directory of your new file. Run:

```
java HelloWorld.java
```

You should see your greeting printed!

Recall from the walk-through above, Java needs to be be compiled before executing. Java version 11 introduced the capability to compile single-file Java programs without explicitly running a command to compile. If our `Hello, World` program were more complex and contained another file, we would need to first run `javac HelloWorld.java`, to compile, followed by `java HelloWorld.java`.

1.2.4. Install IntelliJ

IntelliJ is an **integrated development environment (IDE)**. An IDE is like a text editor on steroids. It not only allows you to write and edit code, but also contains many features that enhance the coding experience. IntelliJ offers code completion hints, debugging, and even it's own compiler. We'll be using it throughout this course, so it's time to get familiar with some of the basics.

Visit the IntelliJ download site. Select your operating system and the Community version. Follow the installation prompts to select your settings. When you reach the window asking for your UI theme, you can choose to *Skip Remaining and Set Defaults*. You will finish on an IntelliJ window listing the options to *Create New Project*, *Import Project*, *Open*, and *Check out from Version Control*.

IntelliJ welcome window

You've installed IntelliJ, and you're ready to start exploring its many features.

1.2.5. Your First Java Project

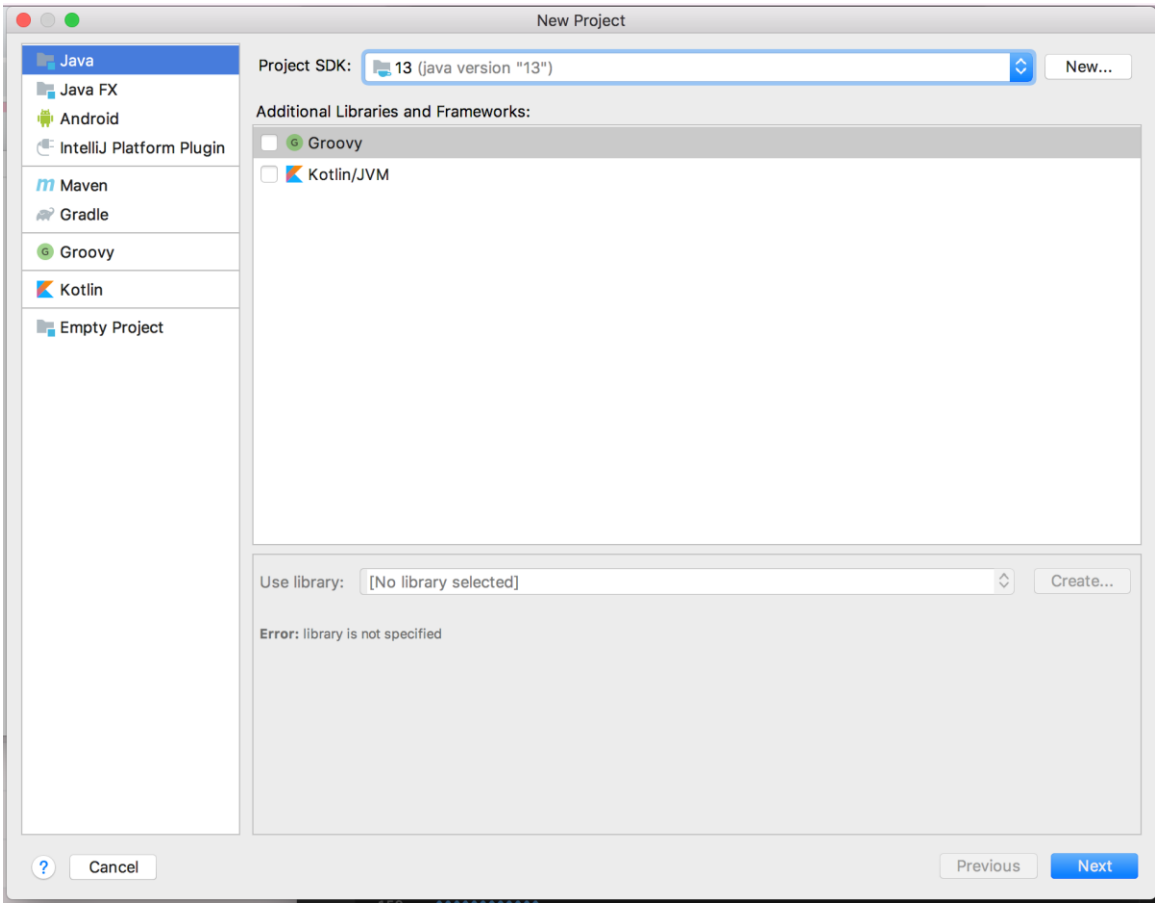
Following the "Hello, World" trend, let's create a new IntelliJ project.

1. Create a new folder to hold your Java practice files. Since you will be creating lots of small projects as you move through this course, we suggest that you also add sub-folders with names corresponding to the related chapters and projects. Something like `java-practice/chapter-name/project-name`.

2. Select the *Create New Project* option from the welcome screen.

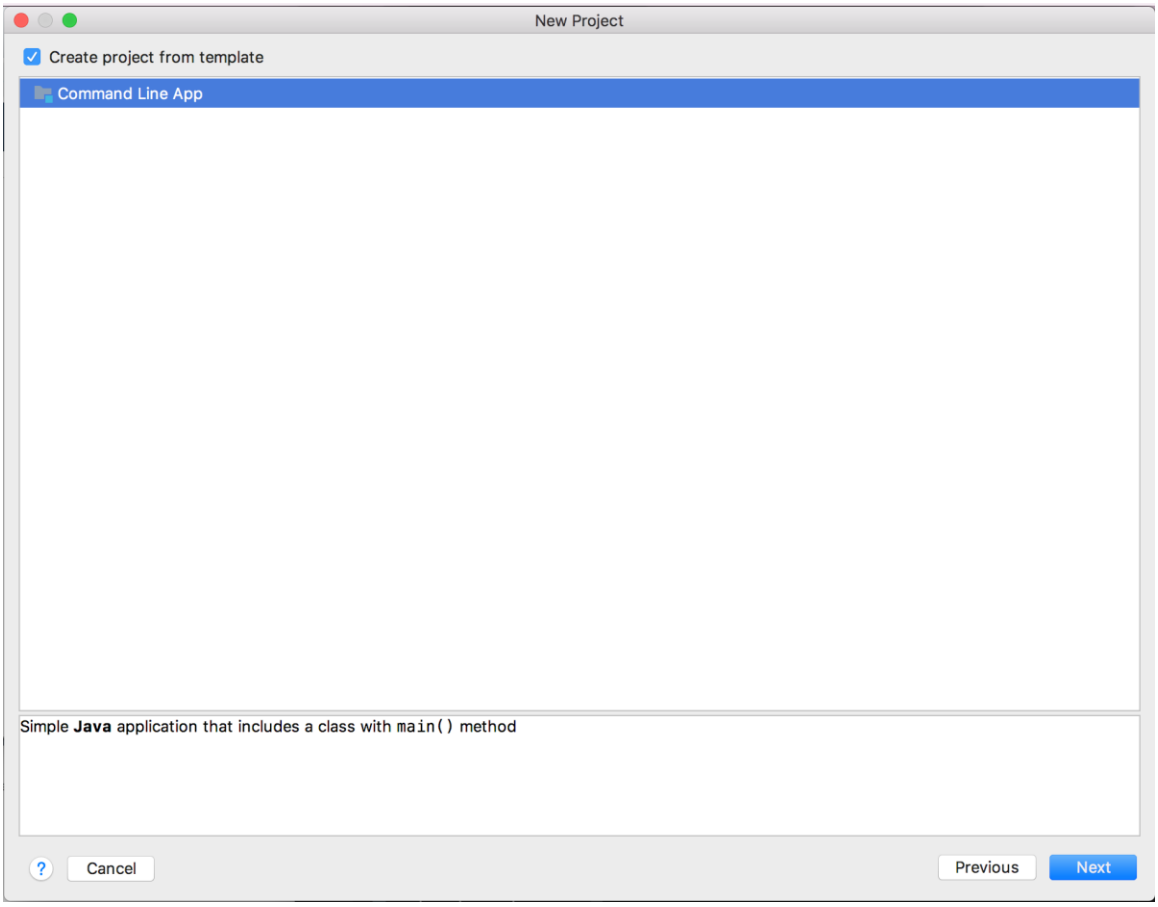
Create new project

3. Clicking *New Project* opens a window with a series of project settings to select. For this first setting, make sure your selected project SDK is the JDK you have installed. This allows IntelliJ to compile our Java code in-app. Click *Next* in the lower right corner of the window to continue selecting settings.



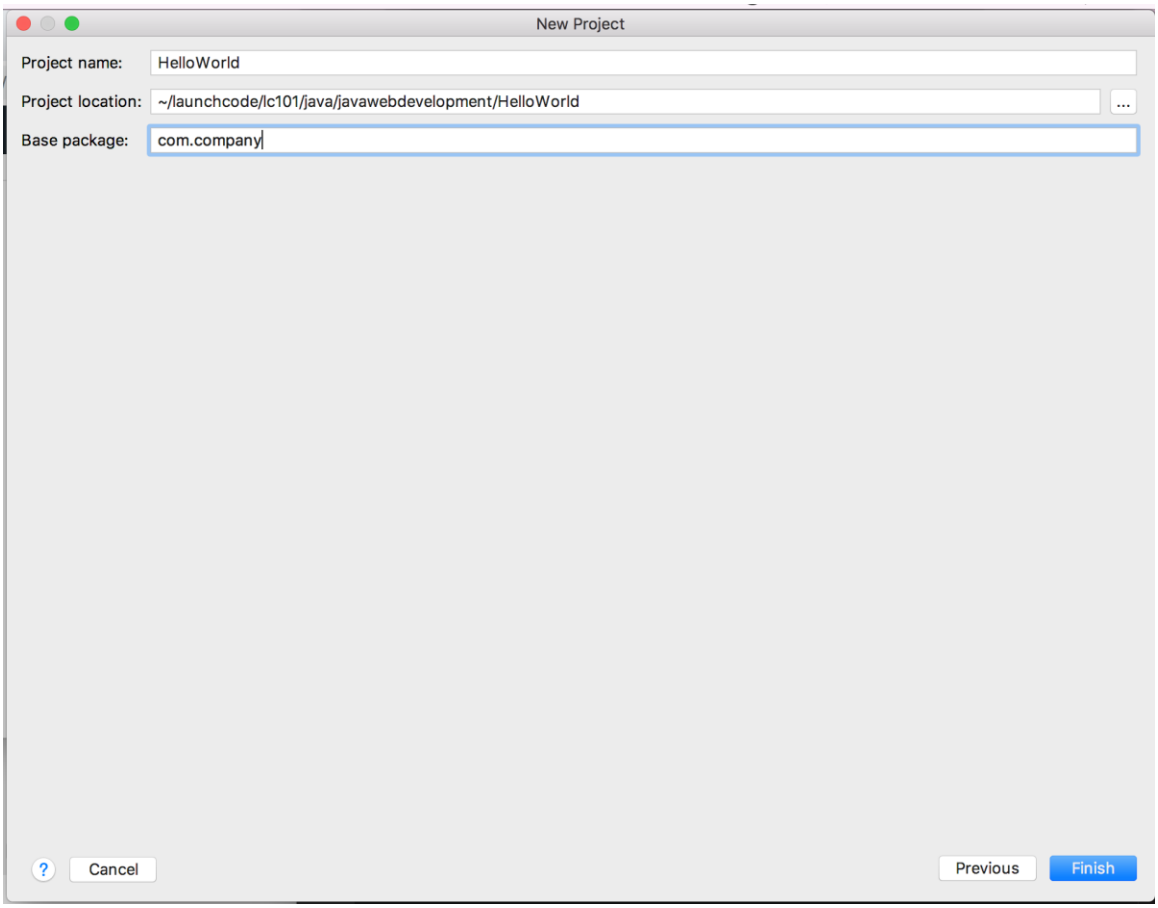
Select project SDK

4. In the second window, select *Create project from template*. This gives us some of the Java project scaffolding to save us some time with project infrastructure.



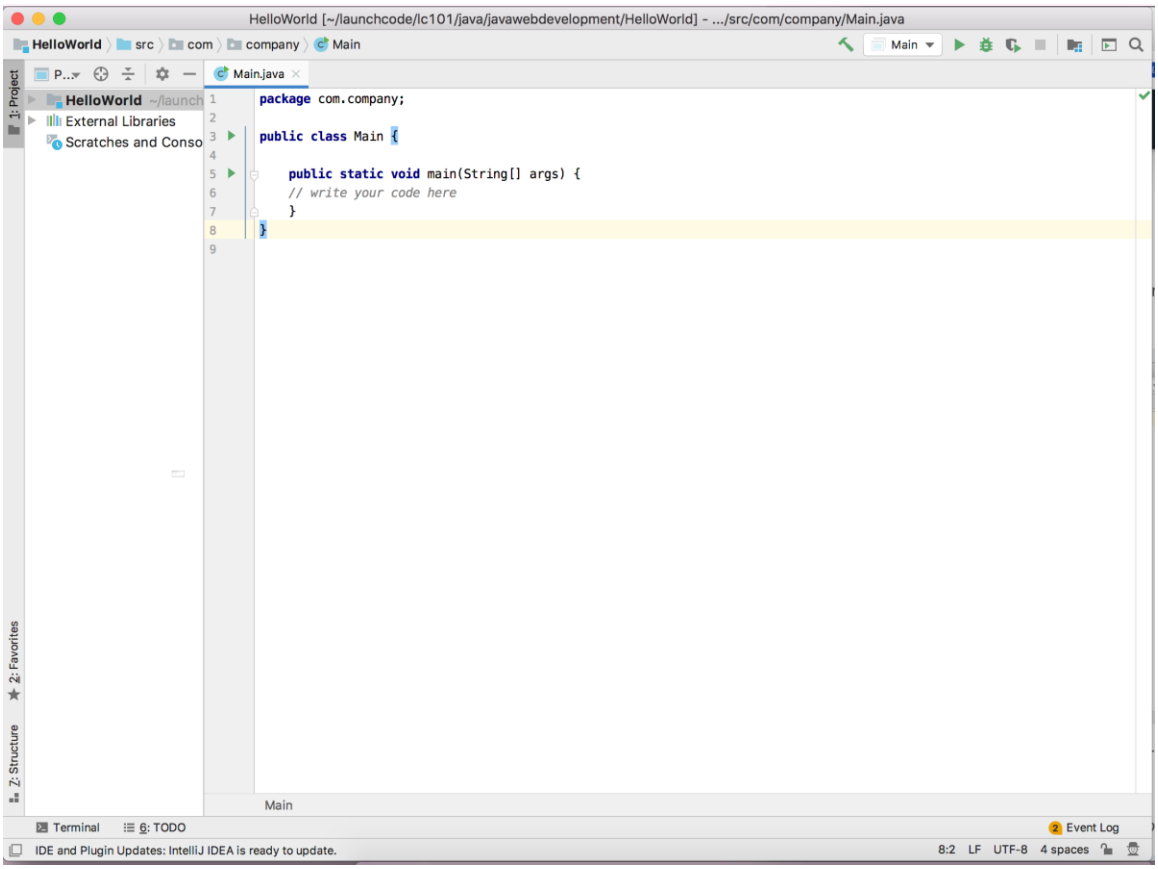
Select project template

5. On the next window, enter `HelloWorld` for the name of the project. Click on the "3-dot" button to select a location to save the project. Here you can choose the Java projects folder you created in step one. Leave the base package as `com.company`.



Create the `HelloWorld` project in your Java projects folder.

6. Click *Finish* to create the project. Below is the view of your new project:



Initial IntelliJ project view

The section on the left is the project's file tree.

Clicking the triangle next to the project name, `HelloWorld`, displays the `src` file, followed by the base package we created, and finally our `Main.java` file.

`Main.java` is also opened on the right in this initial project view.

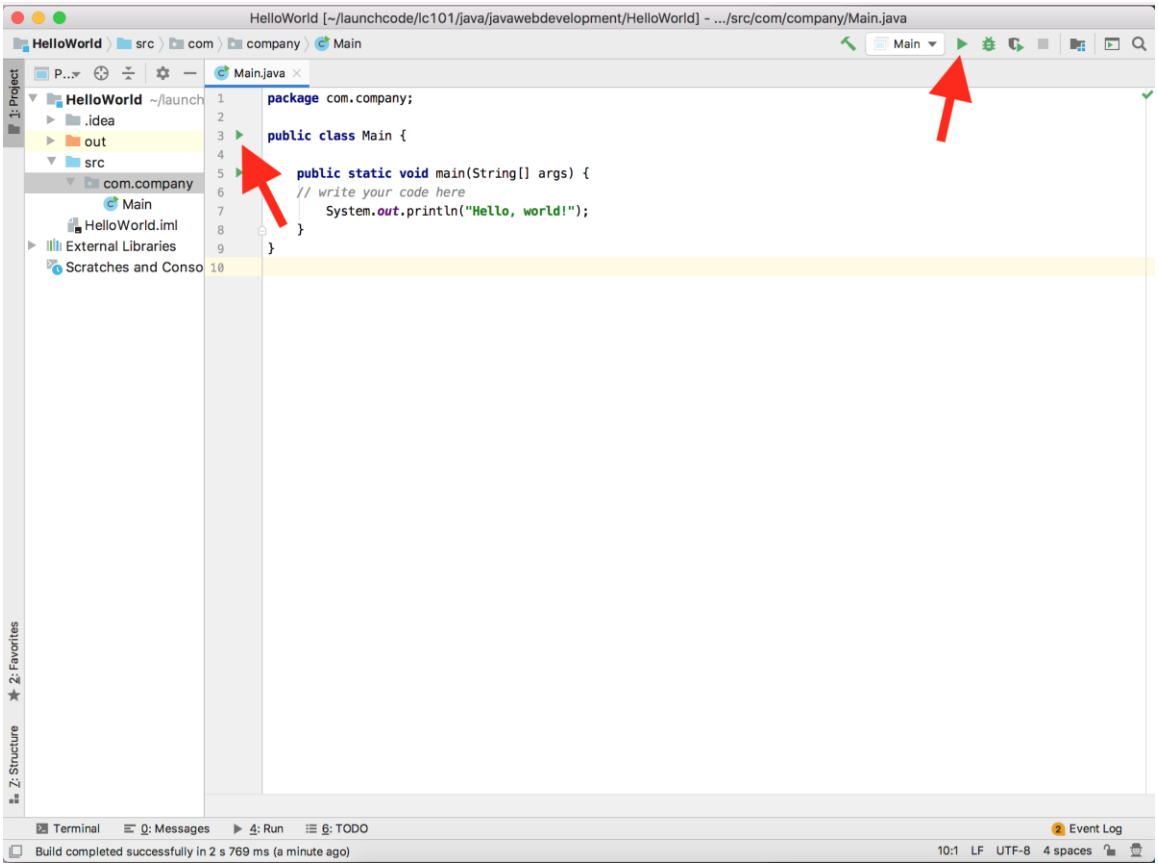
In line 1, `package com.company`, establishes a *package*, which Java uses to help organize and encapsulate our code.

7. We'll dive into the use of a `main` function and `Main` class later. At this point, let's just get right to printing our greeting. Where the project template tells you to write your code on line 6, add the following:

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

Ok sure, we haven't gone over this exact syntax yet. But you can take a guess at what this line will do.

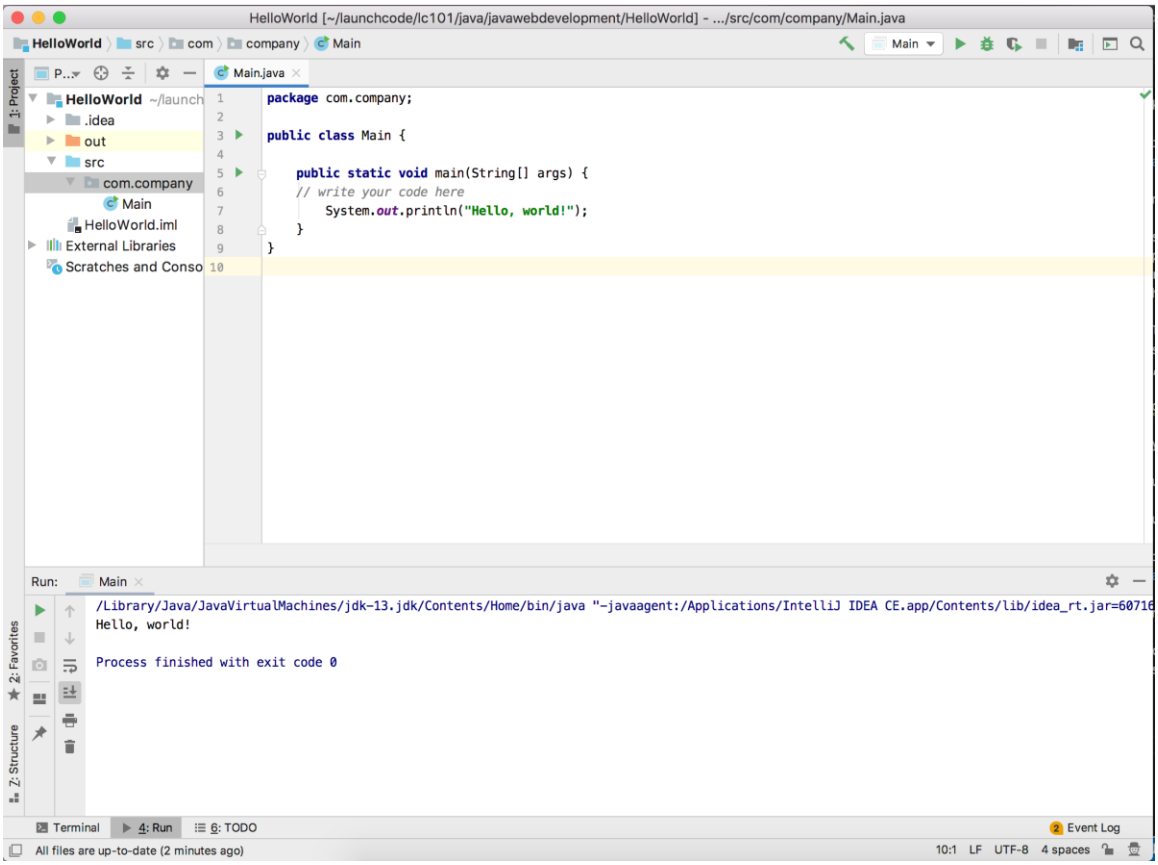
9. To run your program in IntelliJ, you have several options.



IntelliJ run code options

You can click on either of the green arrows indicated above, or choose *Run* from your top menu bar.

10. Once run, IntelliJ will generate a third panel in your view, with your program's output:



IntelliJ output

This is just the start of your relationship with IntelliJ. Not that we know the fundamentals, let's return to Java basics so we can start writing more code.

Question

Given the code below, which line is responsible for printing a message?

```
  public class HelloWorld {
1
2    public static void main(String[] args) {
3       System.out.println("Hello, World");
4    }
5
6 }
7
```

1. line 1
2. line 3
3. line 4

Question

In the sourcecode above, which line is responsible for defining the class?

1. line 1
2. line 3
3. line 4

1.3. Java Web Dev Exercises

1.3.1. Initial Setup

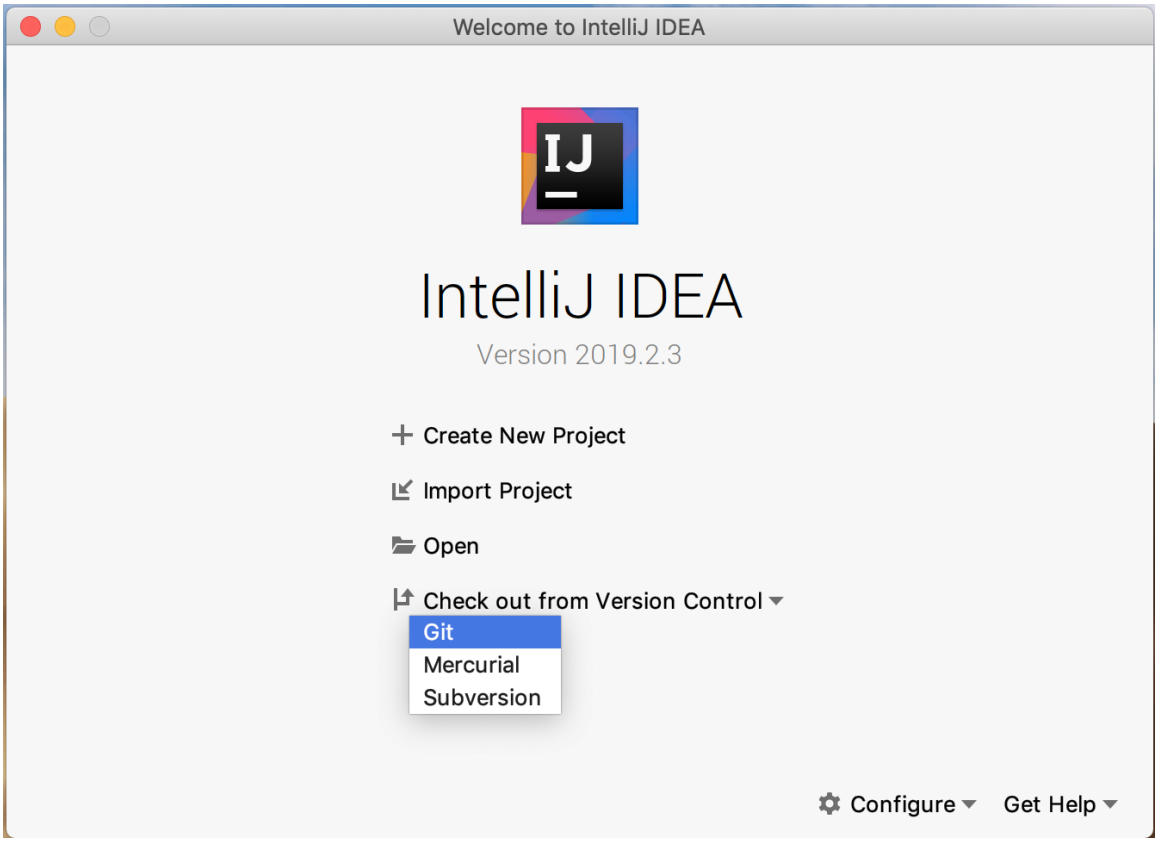
The steps here will walk you through setting up a repository that you'll use to study example code, work on studios, and complete your first assignment of this unit.

1. Create a *Fork* of LaunchCodeEducation/java-web-dev-exercises. Do not clone and create a local version of this repo just yet.
2. Open IntelliJ.

   Note

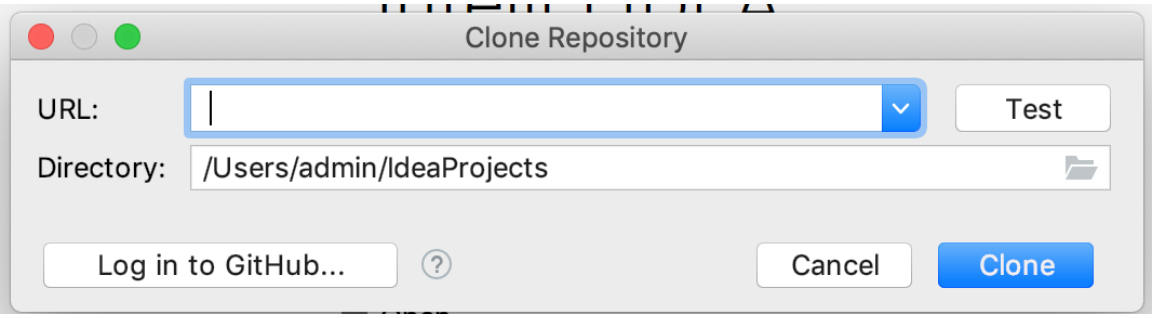
   If the app opens up to an existing project, select *IntelliJ* > *Preferences* > *Appearance & Behavior* > *System Settings* and un-check *Reopen last project on startup*. (For Windows users: *File* > *Settings* > *Appearance & Behavior* > *System Settings*.) Close and Reopen IntelliJ.

3. From the "Welcome to IntelliJ" dialog, select *Check out from Version Control* > *Git*



IntelliJ VCS

4. Click the button on the lower left corner of the dialog to log in to your Github account.

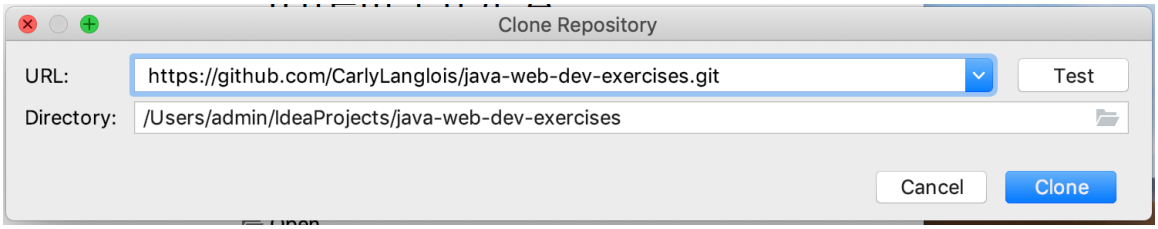


IntelliJ Github

Note

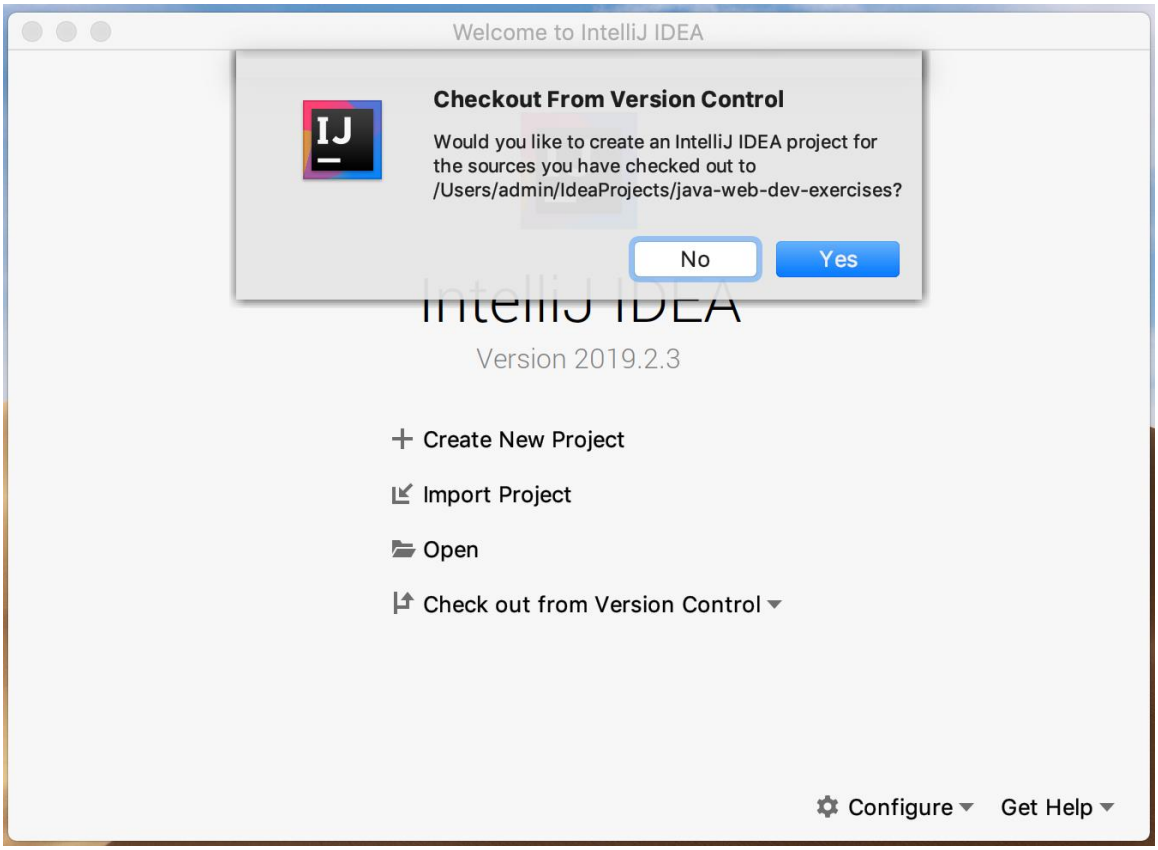
To work with a remote repository in IntelliJ, you need to configure the program to access your GitHub account. We recommend authenticating your account using a token. This takes only one brief extra step, and will prevent you from having to update IntelliJ settings when you ever change your GitHub password.

5. From the URL dropdown options, select your fork of `java-web-dev-exercises`, along with an appropriate source destination directory (i.e. a folder where you've stored other projects for this class).



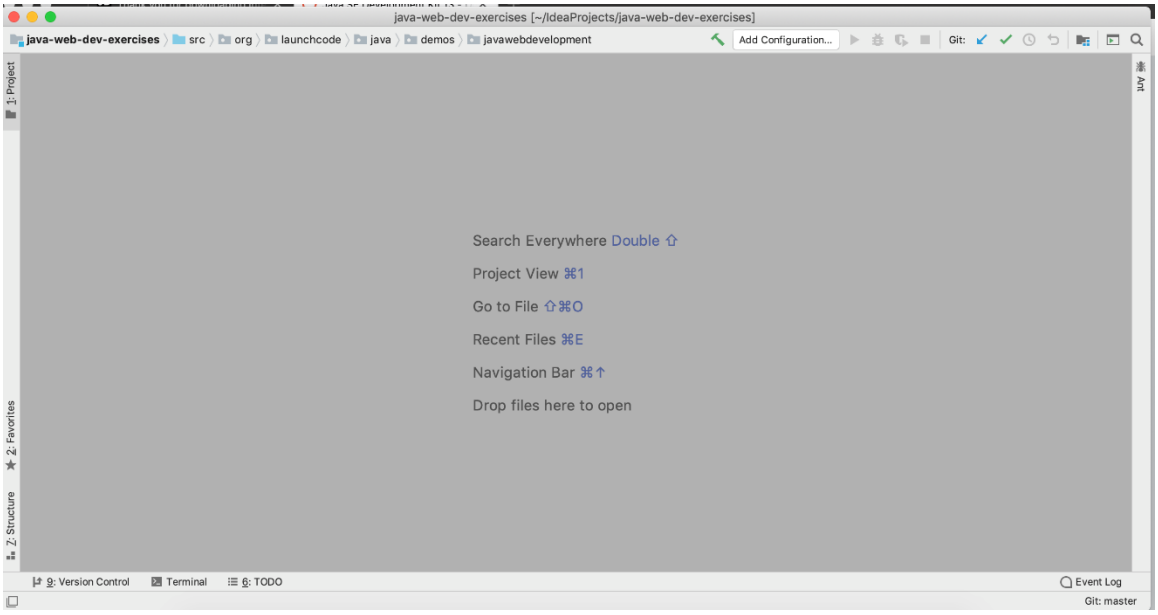
IntelliJ Repo Selection

6. When asked “Would you like to create an IDEA project…” select *Yes*.



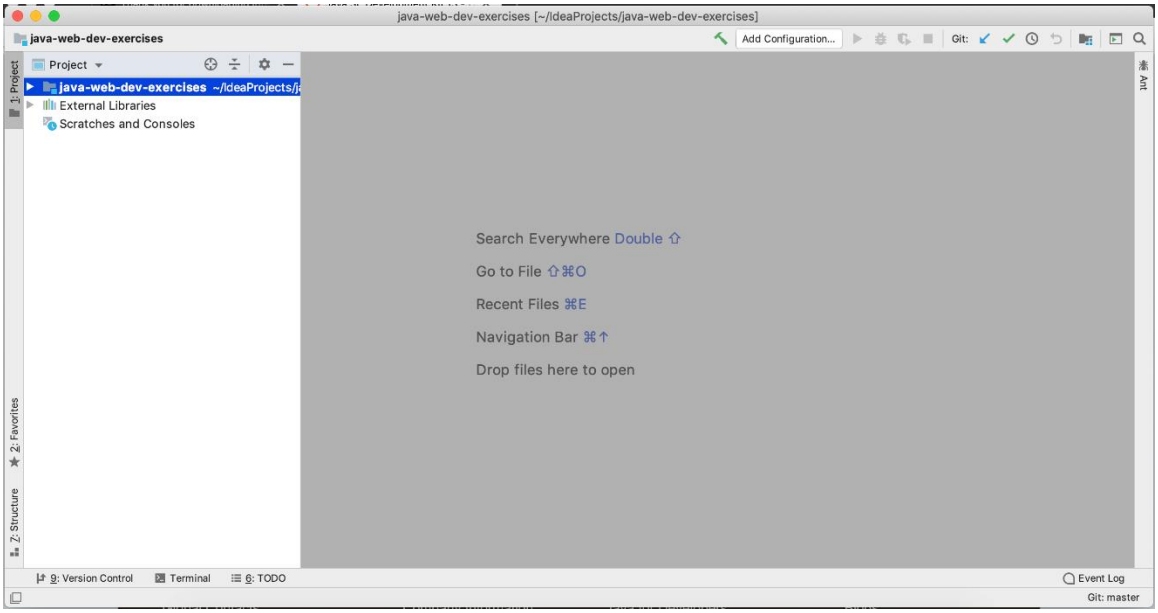
IntelliJ Add File To Git

7. You’ll then be presented with several pages that ask you about other settings for your project. Select the *Next* button on all of these pages, accepting the default settings.
8. When your project is ready, you’ll see a page that looks like the image below. Click on the area in the top left labelled *Project*.



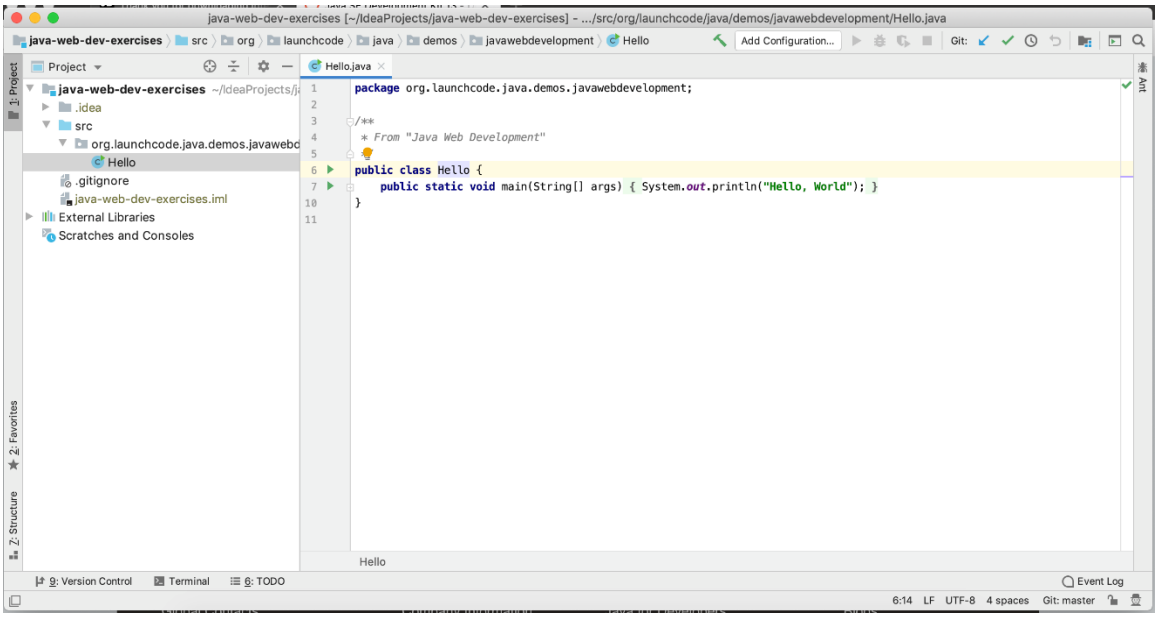
IntelliJ New Project

9. Clicking on *Project* opens a side panel, displaying the file structure of the project you have just set up.



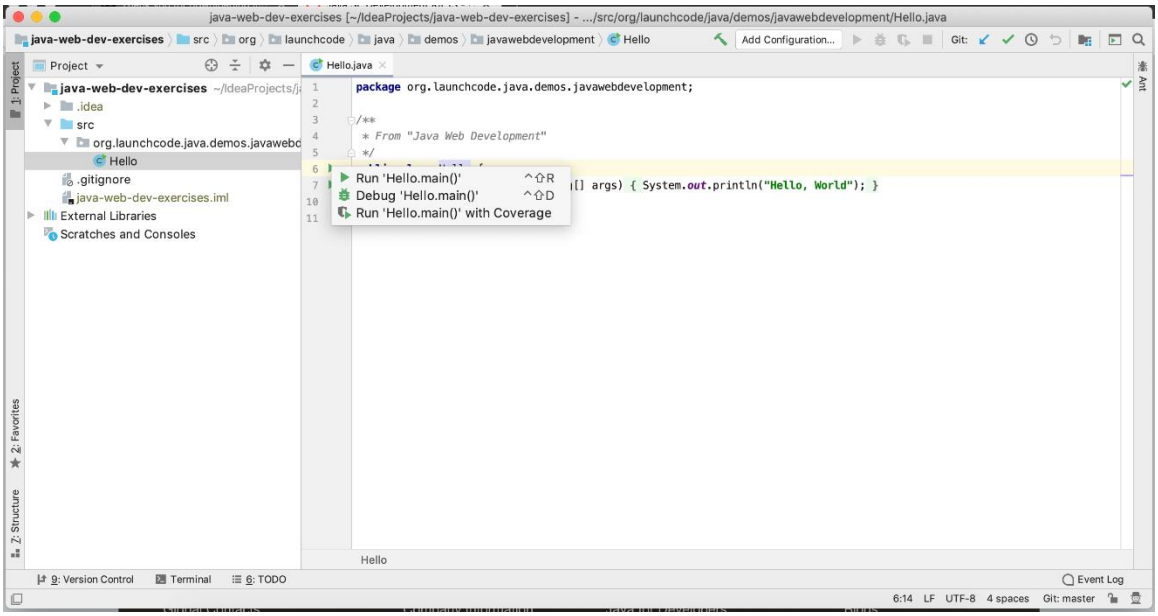
IntelliJ Project Window

10. Double-clicking on the *Hello* file opens the file to the right.



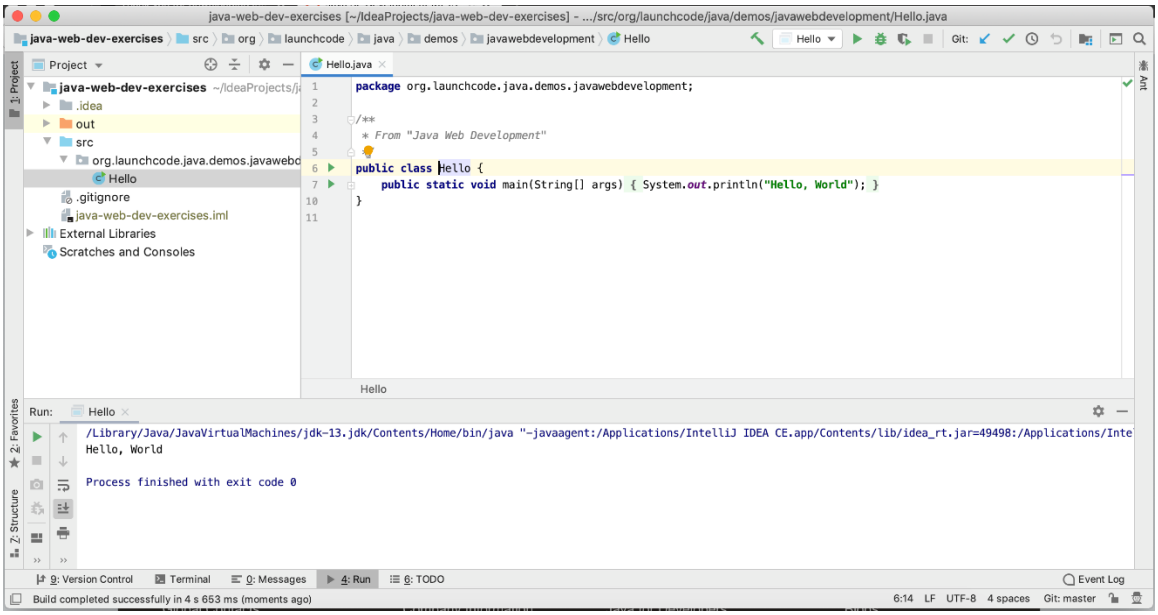
IntelliJ Open File

11. To run the *Hello* program, click on the green arrow next to the class definition and select *Run 'Hello.main()'* from the dropdown menu.



IntelliJ Run File

After a few seconds, you should see a new window appear with your program's output.



IntelliJ File Output

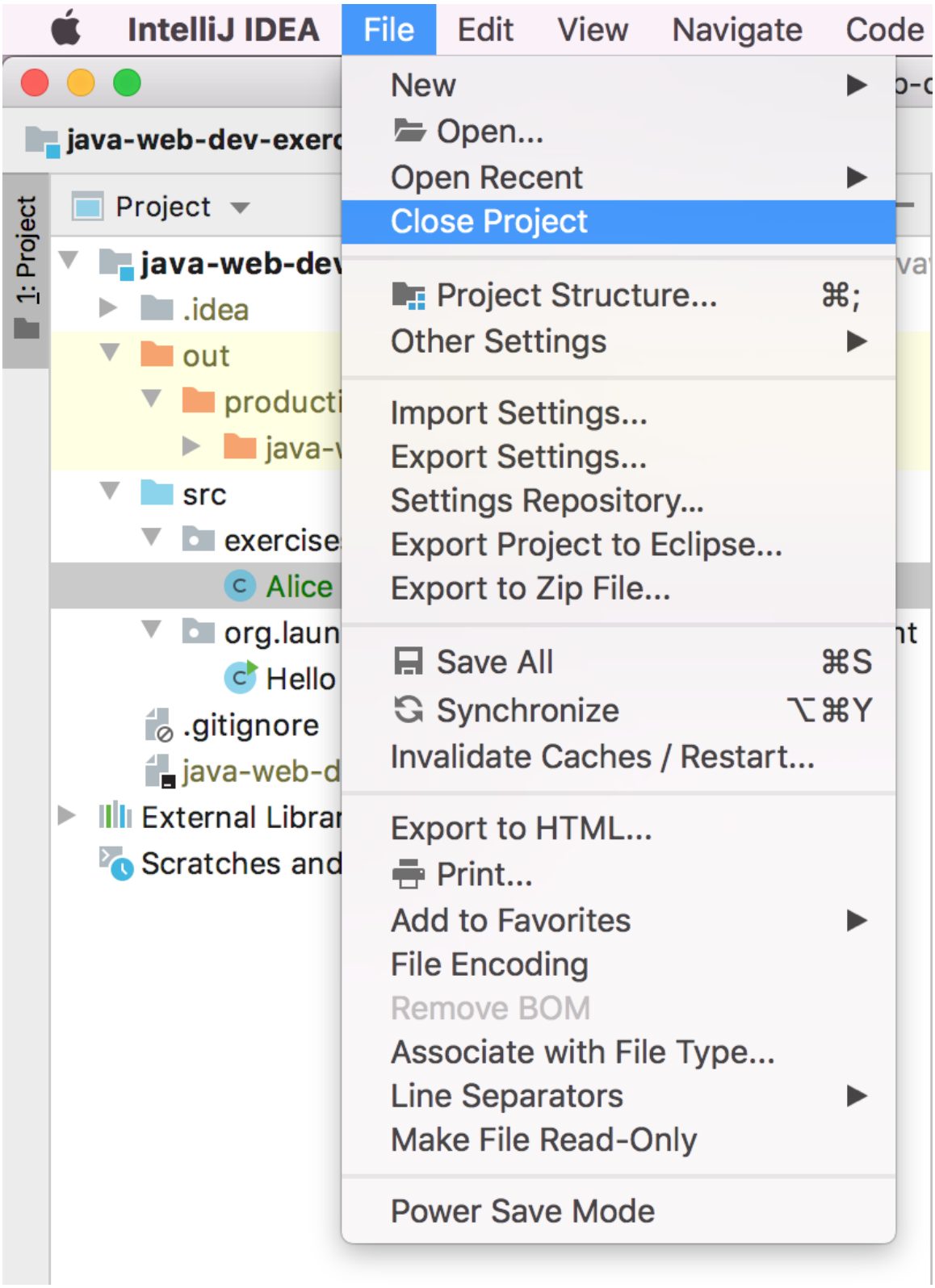
With that, you're ready to go!

1.3.2. Troubleshooting

1.3.2.1. ClassNotFoundException

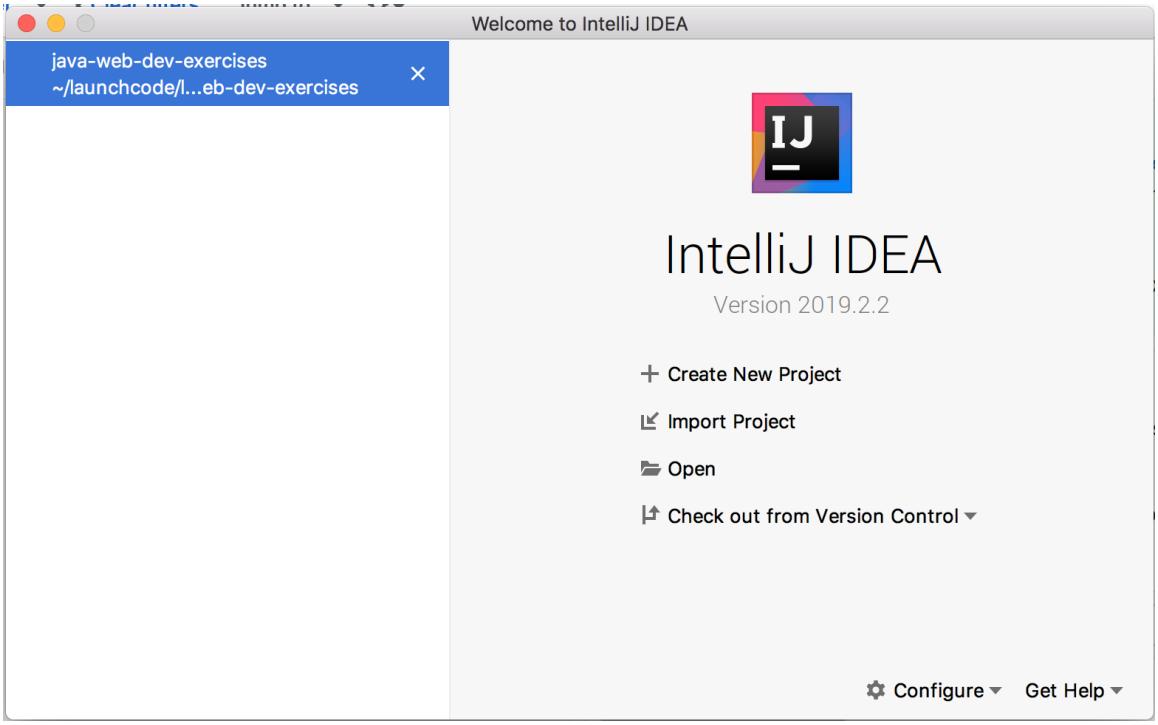
If you experience `java.lang.ClassNotFoundException` when trying to run code after setting up the project, follow these steps:

1. Select *File* > *Close Project*. If you have any other IntelliJ projects open, close them as well.



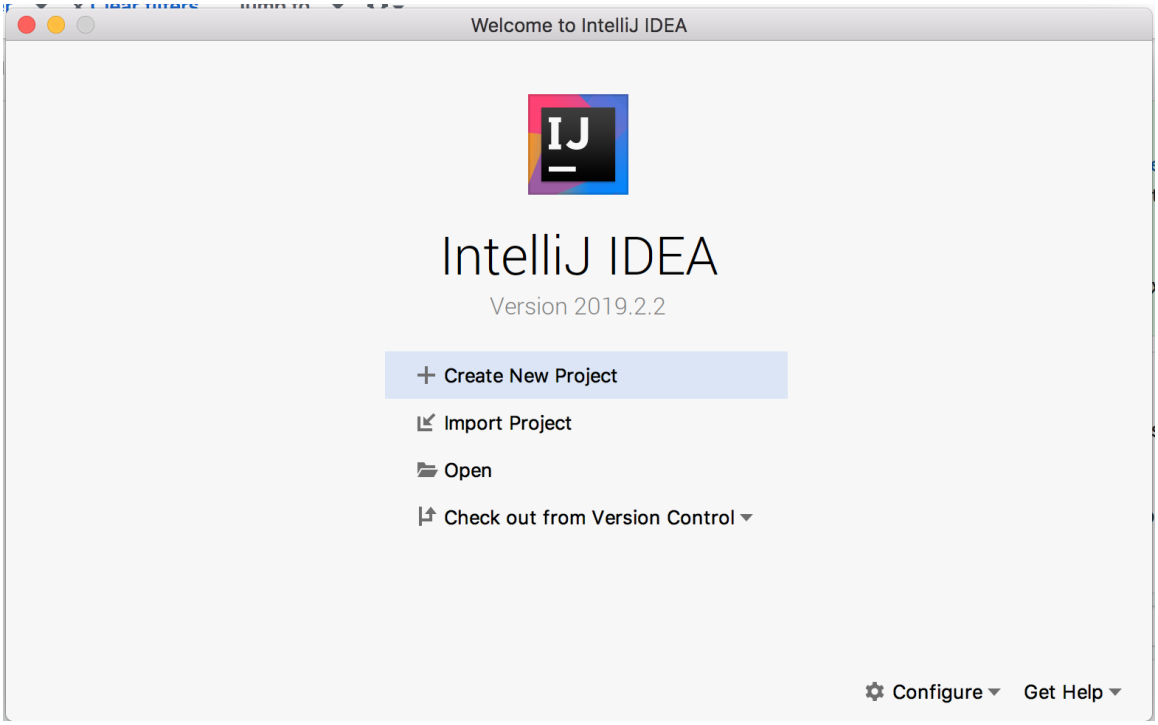
Close Project

2. You should see the IntelliJ startup window, click the *X* next to `java-web-dev-exercises` in the left-hand pane.



Startup with Project

3. From the same startup window, select *Import Project* from the right-hand pane.



Startup without Project

4. Follow the steps that IntelliJ guides you through, accepting all defaults. When prompted to overwrite IntelliJ settings files, confirm that you want to do so.

1.4. Java Naming Conventions

Java has some very straightforward naming conventions. These are universally used by Java programmers, and differ in some cases from conventions commonly used in other languages.

Again, these are conventions. Ignoring them will not prevent your code from running, as long as you are following Java's naming rules. Java's identifier naming rules are somewhat hard to parse, so a good rule-of-thumb is that you should use only letters, numbers, and the underscore character _, and they should always start with a letter.

The naming conventions are more like guidelines than rules and are what other Java coders expect to see when reading your code.

Identifier Type	Convention	Examples
Package	All lowercase	`demos.javawebdevelopment`, `org.launchcode.utilities`
Class	Start with an uppercase letter	`Scanner`, `System`, `Cello`
Method	Start with a lower case letter, and use camelCase to represent multi-word method names	`nextInt()`, `getId()`
Instance variable	Start with a lowercase letter and use camelCase	`id`, `firstName`
Constant	All uppercase letters, words separated by underscores	`MAX_INT`

Note

Constants in Java are variables created using both `static` and `final` modifiers. For example: `static final Double PI = 3.14159`

Tip

If you're not sure about all of these identifier types yet, that's ok. Keep this page in mind for future reference as you read through this book.

Oracle, the company that develops the Java language, provides some more detailed naming conventions. (From the date on this article, you'll note that these have been relatively standard for a very long time!)

2.1. Data Types

2.1.1. Static vs. Dynamic Typing

In a **dynamically typed** programming language (like JavaScript or Python), a variable or parameter can refer to a value of any data type (string, number, object, etc.) at any time. When the variable is used, the interpreter figures out what type it is and behaves accordingly.

Java is a **statically typed** language. When a variable or parameter is declared in a statically typed language, the data type for the value must be specified. Once the declaration is made, the variable or parameter cannot refer to a value of any other type.

For example, this is legal in JavaScript, a dynamically typed language:

Example

```
  let dynamicVariable = "dog";
1 console.log(typeof(dynamicVariable));
2 dynamicVariable = 42;
3 console.log(typeof(dynamicVariable));
4
```

Output

```
string
number
```

After line 1 executes, `dynamicVariable` holds a `string` data type. After line 3 runs, `dynamicVariable` becomes a `number` type. `dynamicVariable` is allowed to hold values of different types, which can be reassigned as needed when the program runs.

However, the corresponding code in Java will result in a compiler error:

Example

```
  String staticVariable = "dog";
1 staticVariable = 42;
2
```

Output

```
error: incompatible types: int cannot be converted to String
```

The compiler error occurs when we try to assign `42` to a variable of type `String`.

Take-home lesson: *We must declare the type of every variable and parameter in a statically typed language*. This is done by declaring the data type for the variable or parameter BEFORE its name, as we did in the example above: `String staticVariable = "dog"`.

Note

We only need to specify the type of a variable or parameter when declaring it. Further use of the variable or parameter does not require us to identify its type. Doing so will result in an error.

Dynamic and static typing are examples of different type systems. The **type system** of a programming language is one of the most important high-level characteristics that programmers use when discussing the differences between languages. Here are a few examples of popular languages falling into these two categories:

1. **Dynamic**: Python, Ruby, Javascript, PHP
2. **Static**: Java, C, C++, C#, TypeScript

Because we need to give plenty of attention to types when writing Java code, let's begin by exploring the most common data types in this language.

2.1.2. Strings and Single Characters

2.1.2.1. Immutability

Strings in Java are *immutable*, which means that the characters within a string cannot be changed.

2.1.2.2. Single vs. Double Quotation Marks

Java syntax requires double quotation marks when declaring strings.

Java has another variable type, `char`, which is used for a single character. `char` uses single quotation marks. The single character can be a letter, digit, punctuation, or whitespace like tab (`'\t'`).

```
  String staticVariable = "dog";
1 char charVariable = 'd';
2
```

2.1.2.3. Manipulation

The table below summarizes some of the most common string methods available in Java. For these examples, we use the string variable `String str = "Rutabaga"`.

String methods in Java

Java Syntax	Description
`str.charAt(3)`	Returns the character at index 3, (`'a'`).
`str.substring(2,4)`	Returns the characters from indexes 2 - 4, (`"tab"`).
`str.length()`	Returns the length of the string.
`str.indexOf('a')`	Returns the index for the first occurrence of 'a', (`3`).
`str.split("delimiter")`	Splits the string into sections at each `delimiter` and stores the sections as elements in an array.
`str.concat(string2).concat(string3)`	In Java, `concat` concatenates only two strings. To join multiple strings, method chaining is required.
`str.trim()`	Removes any whitespace at the beginning or end of the string.
`str.toUpperCase()`, `str.toLowerCase()`	Changes all alphabetic characters in the string to UPPERCASE or lowercase, respectively.
`str.contains("text")`	Searches for the specified text within a string and returns `true` or `false`.
`str.equals(otherString)`	Compares strings for equality and returns a boolean.

Note

We will explore the differences between using `==` and `.equals()` when we discuss Java operators. For now, use `.equals()` if you need to compare two strings.

2.1.3. Primitive Types

A primitive data type is a basic building block. Using primitive data types, we can build more complex data structures called *object* data types.

Java uses its own a set of primitive data types. The table below shows the most common types that beginners are likely to encounter. A more complete list can be found on the Oracle website.

Java Primitive Data Types

Data Type	Examples	Notes
`int`	42	Represents positive and negative whole numbers.
`float`	3.141593 and 1234.567 and 2.0	Represents positive and negative decimal numbers with up to 7 digits.
`double`	3.14159265358979 and 10000.12345678912	Represents positive and negative decimal numbers with 15-16 digits.
`char`	'a' and '9' and 'n'	A single unicode character enclosed in single quotes `''`.
`boolean`	`true` and `false`	Booleans in Java are NOT capitalized.

Warning

As we will see in a later section, the `float` data type sacrifices some accuracy for speed of calculation. Thus, evaluating 1.11111 + 3 results in an answer of 4.1111097 instead of 4.11111.

Anytime you need to perform calculations with decimal values, consider using the `double` type instead of `float`.

2.1.4. Non-primitive Types

Primitive data types are *immutable* and can be combined to build larger data structures. One example is forming the `String` "LaunchCode" from multiple `char` characters ('L', 'a', 'u', etc.).

`String` is a non-primitive data type, also called an *object type*. As we saw in the `String` table above, object types have methods which we can call using dot notation. Primitive data types do not have methods.

Note

Primitive data types in Java begin with a lower case letter, while object data types in Java begin with a capital letter.

Later in this chapter, we will explore the Array and Class object types.

2.1.5. Autoboxing

There may be situations when we call a method that expects an object as an argument, but we pass it a primitive type instead (or vice versa). In these cases, we need to convert the primitive type to an object, or convert an object type into a primitive.

In older versions of Java, it was the programmer's responsibility to convert back and forth between primitive types and object types whenever necessary. Converting from a primitive type to an object type was called **boxing**, and the reverse process (object to primitive) was called **unboxing**.

Examples

Boxing:

```
  int someInteger = 5;
1 Integer someIntegerObject = Integer.valueOf(someInteger);
2 ClassName.methodName(someIntegerObject);
3
```

1. Line 1 declares and initializes the variable `someInteger`.
2. Line 2 and converts the primitive `int` to the `Integer` object type.
3. Line 3 calls `methodName` and passes `someIntegerObject` as the argument. If `methodName` expects an object type and we tried sending an `int` instead, we would generate an error message.

Unboxing:

Let's assume that a method returns a random number of `Integer` type, and we want to combine it with a value of `int` type.

```
  int ourNumber = 5;
1 Integer randomNumber = ClassName.randomNumberGenerator();
2 int randomInt = (int) randomNumber;
3 int sum = ourNumber + randomInt;
4
```

1. Line 2 declares and initializes `randomNumber` as an `Integer` type.
2. Line 3 and converts `randomNumber` to an `int` and stores the value in the `randomInt` variable.

Converting between data types in order to pass values between methods quickly became tedious and error prone. In the newer versions of Java, the compiler is smart enough to know when to convert back and forth, and this is called **autoboxing**.

For us, the consequence of autoboxing is that in many situations, we can use primitive and object types interchangeably when calling methods or returning data from those methods.

Tip

It's a best practice to use primitives whenever possible. The primary exception to this occurs when storing values in collections, which we'll learn about in a future lesson.

Each of the primitive data types has a corresponding object type:

1. `int` —> `Integer`
2. `float` —> `Float`
3. `double` —> `Double`
4. `char` —> `Char`
5. `boolean` —> `Boolean`

2.1.6. References

1. Primitive Data Types (docs.oracle.com)
2. Autoboxing and Unboxing (docs.oracle.com)
3. Variables (docs.oracle.com)

2.1.7. Check Your Understanding

Question

Which of the following is NOT a number data type in Java:

1. `number`
2. `int`
3. `float`
4. `double`

Question

Name the Java method responsible for checking string equality:

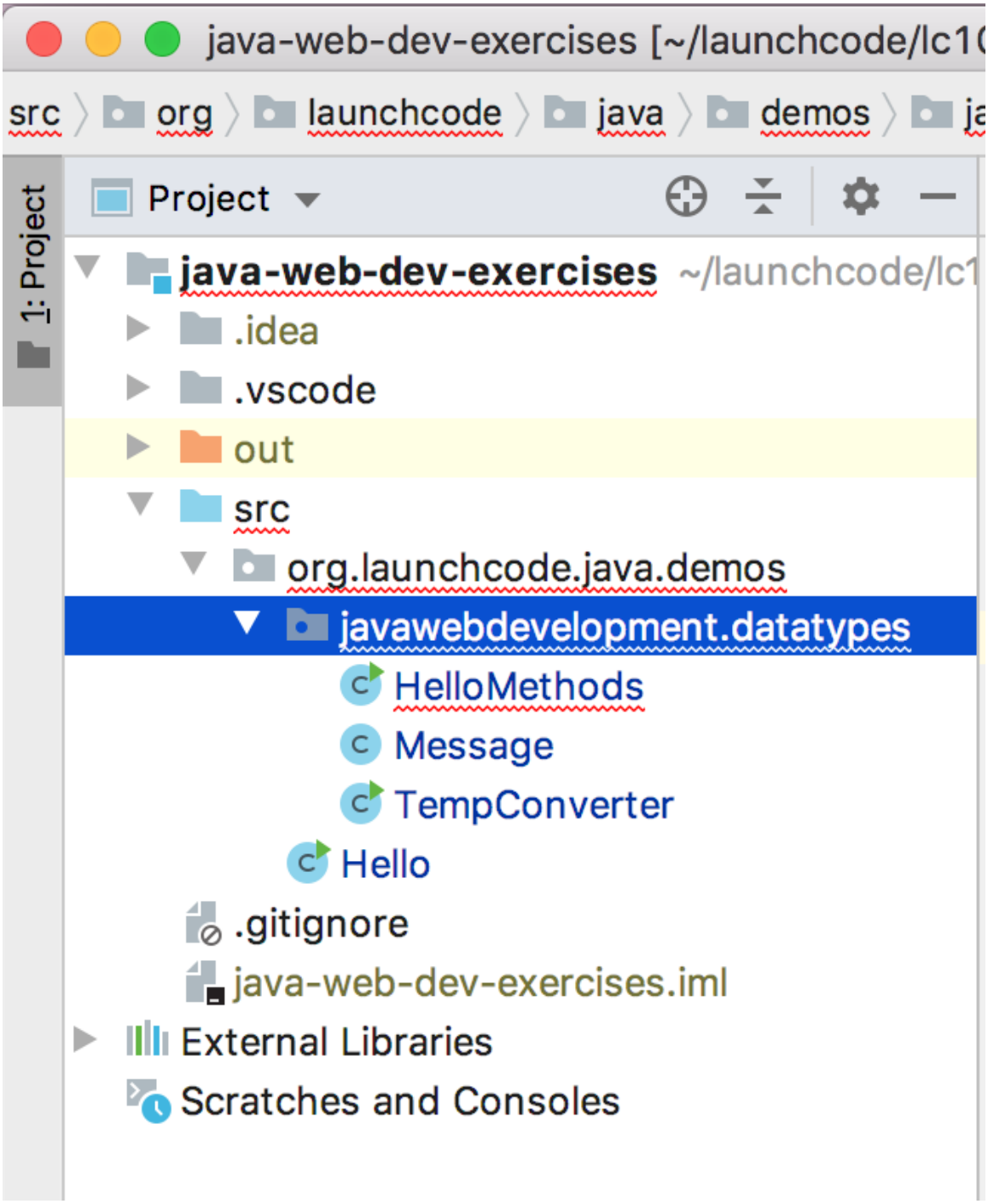
1. `.isEqualTo()`
2. `.sameAs()`
3. `.equals()`
4. `===`

2.2. Some Java Practice

Let's move beyond our "Hello, World" example and explore a simple temperature conversion program. We want our function to convert a Fahrenheit temperature to Celsius.

2.2.1. Temperature Conversion

1. Open the TempConverter file in the java-web-dev-exercises project in IntelliJ.



The TempConverter file

2. Here's what the file should look like. We will analyze the different statements in a moment.

```
  package org.launchcode.java.demos.javawebdevelopment.datatypes;
 1
 2 import java.util.Scanner;
 3
 4 public class TempConverter {
 5    public static void main(String[] args) {
 6       double fahrenheit;
 7       double celsius;
 8       Scanner input;
 9
10       input = new Scanner(System.in);
11       System.out.println("Enter the temperature in Fahrenheit: ");
12       fahrenheit = input.nextDouble();
13       input.close();
14
15       celsius = (fahrenheit - 32) * 5/9;
16       System.out.println("The temperature in Celsius is: " + celsius + "°C");
17    }
18 }
19
```

3. Build and run the program to verify that it works. Entering a Fahrenheit temperature of 212 yields the result, The temperature in Celsius is: 100.0°C.

There are several new concepts introduced in this example. We will look at them in the following order:

1. Java packages
2. The import statement
3. Declaring variables
4. Collecting input with the Scanner class

2.2.2. Java Packages

Line 1 of of the program above, package org.launchcode.java.demos.javawebdevelopment.datatypes; declares the package in which the file resides. For this simple program, your code could run without this line. However, you want to get used to always declaring the package of your Java classes.

Packages help to **encapsulate** your code. Encapsulation refers to the practice of shielding your code from outside influences. It's an essential component of good object oriented programming, and package declaration in Java is just one application of this principle. Without declaring a package, a Java class exists within the default package. In larger applications, leaving all classes in the default package risks naming conflicts and bugs.

2.2.3. import

The import statement in Java allows us to access the class, methods, and data stored a different file. import tells the compiler that we are going to use a shortened version of the class name. In this example, we are going to use the class java.util.Scanner, but we can refer to it as just Scanner. We could use the java.util.Scanner class without any problem and without any import statement, provided that we always refer to it by its full name.

This idea bears repeating. In Java, you can use any available class WITHOUT having to import it, but you must use the full name of the class. “Available” classes include:

1. All the classes you define in the .java and .class files in your current working directory.
2. All the classes that get shipped with the software.

Try It

In the sample code, remove the import statement in line 3, and change Scanner on lines 9 & 11 to java.util.Scanner. The program should still compile and run.

The class naming system in Java is very hierarchical. The *full* name of the Scanner class is really java.util.Scanner. You can think of this name as having two parts:

1. java.util is called the *package*,
2. Scanner is the class name.

We’ll talk more about the class naming system a bit later.

2.2.4. Declaring Variables

In the example above, lines 7 - 9 contain variable declarations:

```
  double fahrenheit;
7 double celsius;
8 Scanner input;
9
```

Since Java is a statically typed language, we must always declare the data type for any variable. Lines 7 & 8 establish that fahrenheit and celsius will hold values of type double. In line 9, the variable input references a Scanner object.

If later in the code we try to initialize fahrenheit with a string:

```
fahrenheit = "xyz"
```

the compiler throws an error because fahrenheit is declared to be a double.

The following error is common for new Java programmers. Suppose we forget to include the declaration for celsius. What happens when we try to compile and run our program?

Try It

1. Edit your TempConverter class by removing line 8, which declares the variable celsius.
2. Click any of the “Run” options in IntelliJ. Alternatively, use the terminal to navigate to the parent directory of your TempConverter.java class and run java TempConverter.java.

Your terminal will return some errors that resemble these:

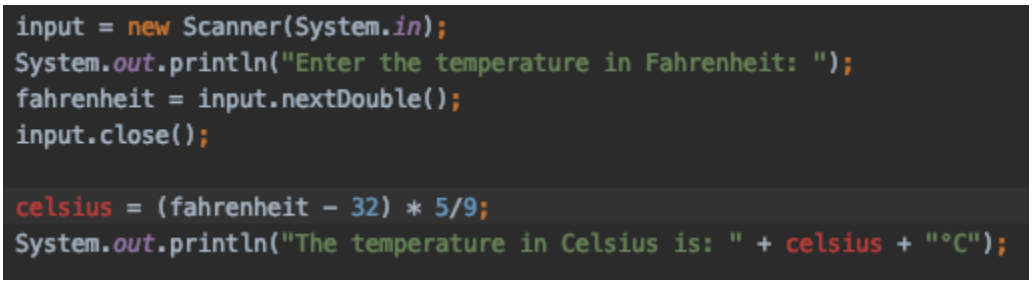
```
Error:(16, 9) java: cannot find symbol
symbol:   variable celsius
location: class TempConverter

Error:(17, 64) java: cannot find symbol
symbol:   variable celsius
location: class TempConverter
```

These two *compiler errors* occur before the program runs. The values in the parentheses () give the line number and text column where the error was found. In the first description (line 16, column 9), the celsius variable before the = is flagged. When this type of error happens, it usually means that the variable was not declared before we tried to initialize it with a value.

The second error message (line 17, column 64) occurs because we use celsius before it has been assigned a value.

Note

When using an IDE such as IntelliJ, your work is typically checked by the IDEs built-in compiler as you write your code. Errors are often visually indicated by the IDE as you type. This avoids having to explicitly compile your code before finding errors. Nice, huh?



The red coloring of the celsius variables indicate errors.

The general rule in Java is that you must decide on the data type for your variable first, and then declare that variable before you use it. There is much more to say about the static typing of Java, but for now this is enough.

Note

As in other languages, Java allows you to declare and initialize your variables in the same line:

```
double celsius = (fahrenheit - 32) * 5/9;
```

2.2.5. Add Comments to Your Code

As programs get bigger and more complicated, they get more difficult to read. Good programmers try to make their code understandable to others, but it is still tricky to look at a large program and figure out what it is doing and why.

Also, there are times when programmers need to isolate or ignore certain portions of their code as they are testing it. In the “Try It” box above, you were instructed to *remove* a line of code in order to create compiler errors. However, programmers are usually reluctant to delete lines that they might need to bring back.

Best practice encourages us to add **comments** to our programs. These are notes that clearly explain what the code is doing.

A comment is text within a program intended only for a human reader——it is completely ignored by the compiler or interpreter. In Java, the // token indicates the start of a comment, and the rest of the line gets ignored. For comments that stretch over multiple lines, the text falls between the symbols /* */.

Comments can also be used to temporarily skip a portion of the code when a program runs. Instead of removing double celsius; in TempConverter, we could *comment out* the line. This would create the same compiler errors we wanted to witness, but it would preserve the original code and allow us to easily reactivate it by removing the // token from the line.

Example

```
  import java.util.Scanner;
1
2 // Here is an example of a comment.
3
4 /* Here is how
5 to have
6 multi-line
7 comments. */
8
9 /*
10 Or
11 like
12 this.
13 */
14
15 public class HelloWorld {
16    public static void main(String[] args) {
17       Scanner input; // Comments do not have to start at the beginning of a line.
18
19       input = new Scanner(System.in);
20       System.out.println("Please enter your first name: ");
21       String name = input.next(); //Declares the 'name' variable and initializes it
22 with text from the command line.
23       input.close();
24
25       System.out.println("Hello, " + name + "!");
26
27       // System.out.println("This line will NOT print!");
28    }
29 }
```

2.2.6. Collect Input with the Scanner Class

In Java, Scanner objects make getting input from the user, a file, or even over the network relatively easy. For our temperature conversion program, we declared the variable input to be of type Scanner.

```
  Scanner input;
9
```

We want our program to prompt the user to enter in a number in the command line. We accomplish this by creating a Scanner instance using the word new and then calling the constructor and passing it the System.in object:

```
   input = new Scanner(System.in);
11
```

Notice that this Scanner object is assigned to the name input, which we declared to be a Scanner object earlier.

And you know those System statements we've been using? Like System.in above here, and System.out.println() for print statements. Well, System itself is a java class. System.in is similar to System.out except, as the name implies, it is used for input.

Note

If you are wondering why we must create a Scanner object to read data from System.in when we can write data directly to System.out using println, you are not alone. We will talk about the reasons why this is so when we dive into Java streams.

Next, line 12 asks the user to enter a number, and in line 13 we use input to read the value from the command line:

```
   System.out.println("Enter the temperature in Fahrenheit: ");
12 fahrenheit = input.nextDouble();
13
```

Here again we see the implications of Java being a strongly typed language. Notice that we must call the method nextDouble, because the variable fahrenheit was declared as a double.

Because Java is a statically typed language, we must call the appropriate method on the Scanner object to ensure the entered data is of the correct type. In this case, the compiler compares the types for fahrenheit and input.nextDouble() and throws an error if the two do not match.

The table below shows some commonly used methods of the Scanner class. There are many others supported by this class, and the Oracle website provides a complete listing of the Scanner methods.

Scanner methods

Method Name	Return Type	Description
hasNext()	boolean	Returns true if more data is present.
hasNextInt()	boolean	Returns true if the next item to read is an int data type.
hasNextFloat()	boolean	Returns true if the next item to read is a float data type.
hasNextDouble()	boolean	Returns true if the next item to read is a double data type.
nextInt()	int	Returns the next item to read as an int data type.
nextFloat()	float	Returns the next item to read as a float data type.
nextDouble()	double	Returns the next item to read as a double data type.
next()	String	Returns the next item to read as a String data type.
nextLine()	String	Returns the next line to read as a String data type.

2.2.6.1. Closing Scanner

To collect data from the command line or other source, create a Scanner object. This opens up resources in your machine to manage the input, and these resources remain open even after the required data is loaded into your program.

Leaving a Scanner open is like keeping a window open in your house 24/7. Anyone can climb into your home, and you lose $$$ by trying to heat and cool your space while it is open to the outside air. Similarly, an open Scanner can allow unintended access to your program, and it ties up resources that might be needed elsewhere.

Best practice states that if you open a Scanner object, close it after it finishes its job. Line 14 does this in our TempConverter program:

```
   input.close();
14
```

The general syntax is scannerObjectName.close().

2.2.6.2. Moving Beyond the Command Line

The Scanner class serves as a kind of adapter that gathers primitive data types as input and converts them into object types (e.g. it converts an int into Integer). We will discuss the purpose of this later, but for now, know that this adaptation makes low-level data types easier to use.

For the temperature conversion program, we collected user input from the command line, but there are other options for collecting data for our programs. In future examples, we will create a Scanner object by passing a File object as a parameter instead of System.in.

2.2.7. Check Your Understanding

Question

An import statement is required to use a Java class defined in another package.

1. True
2. False

Question

Which of the following Scanner methods should you use to return an expected String input? Check ALL that apply.

1. .hasNext()
2. .nextLine()
3. .next()
4. .nextFloat()

2.3. More Data Types

2.3.1. Arrays

In Java, an array is an ordered, fixed-size collection of elements. To comply with static typing, the items stored in an array must all be the same data type. We can create an array of integers or an array of strings, but we may NOT create an array that holds both integers and strings.

The syntax for creating an array capable of holding 10 integers is:

```
int[] someInts = new int[10];
```

Note the square brackets next to `int`. This tells Java that we want `someInts` to store a collection of integers instead of a single number.

To create an array of a different size, replace the number `10` in the brackets with the desired size. To create an array holding a different type, replace `int` (on both sides of the assignment) with the desired type, like `double` or `String`.

In addition to the example above, we can initialize an array using a literal expression:

```
int[] someOtherInts = {1, 1, 2, 3, 5, 8};
```

Here, the size of the array is implied by the number of elements in the literal expression `{1, 1, 2, 3, 5, 8}`. Also note the use of curly braces `{ }` instead of square brackets `[ ]`.

To access array elements, we use square brackets and *zero-based indexing*.

```
int anInt = someOtherInts[4];
// anInt stores the integer 5.
```

Arrays in Java may NOT change size once created. This is limiting and not very practical. Thankfully, Java provides more flexible ways to store data, which we will explore in a later lesson. These objects will allow us to rearrange, add to, or remove data.

Aside from using arrays to build some simple loop examples, we will only use them in special cases. However, they are a core part of Java, so it's good to know how they work.

2.3.2. Java Objects

In Java, **objects** are structures that have a *state* and a set of *behaviors*. The state of an object includes properties/data that the coder can define and modify. Behaviors are actions that run when requested, and they can be used to evaluate, manipulate, or return data.

An array is one example of an object. It contains *data*, which are the values stored as the individual elements. The *behaviors* are methods like `sort()` that perform actions related to the elements in the array.

The `String` data type is also an example of an object. For `String language = "Java"`, the data would be the characters. The String manipulation section gives several of the behaviors available to the `language` object. For example, `language.length()` returns the value `4`, which tells us how many characters are present in the string.

Every variable in Java refers to either a primitive data type or to an object.

2.3.3. Class Types

A **class** is a template for creating objects. In addition to the object types introduced so far, any class in Java also defines a type. We'll have much more to say about classes and objects, but for now you need to recognize the basic syntax of class types and class creation.

If we have a class `Cat`, we can declare and create an instance of `Cat` using the `new` keyword:

```
Cat myCat = new Cat();
```

1. `Cat myCat` declares the variable `myCat` and sets it to be of type `Cat`.
2. `= new Cat()` initializes the variable with a new `Cat` object.
3. Any arguments that are required to build the new `Cat` object must be included within the parentheses.

Just like a variable can be declared as a primitive data type like `char` or `double`, it can also be declared as a specific *class* type. Variables that hold objects—like `String name = "Blake"` or `myCat`—are said to be *reference variables*. Using this terminology, `name` is a reference variable of the `String` class, and `myCat` is a reference variable of type `Cat`.

Note

Java uses two general data types—primitive and object. A *class type* is NOT a new data type. Instead, it is just a specific name applied to the more general object data type.

`int` and `char` are both primitive data types, but the values they can store differ. Similarly, `String` and `Cat` are both object types, but they represent different classes.

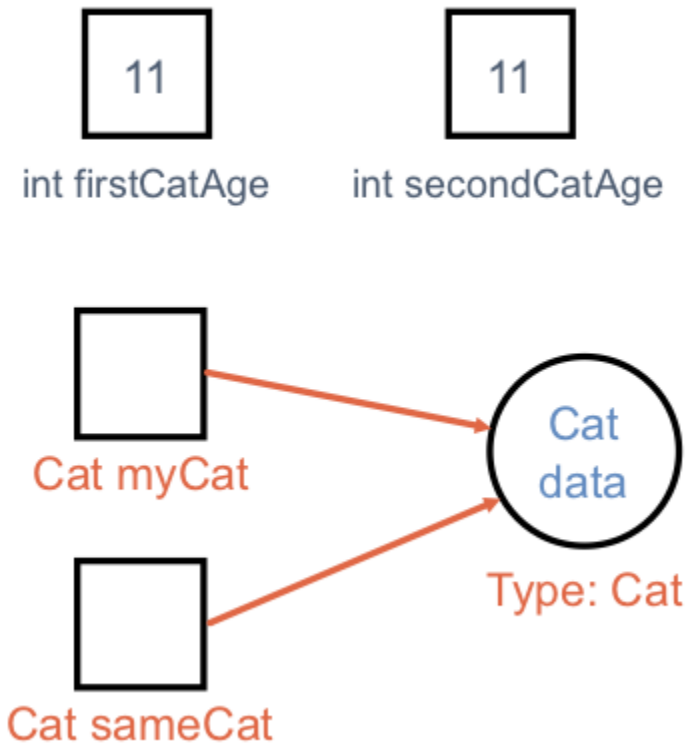
2.3.3.1. Reference Variables

Reference variables are different from primitive types in an essential way. A reference variable (such as `myCat` above) does not actually store the object in question. Instead, it stores a *reference* to the object, which is literally a memory address. We visualize a reference as an arrow pointing to the location of the object in memory.

Consider this code:

```
  int firstCatAge = 11;
1 int secondCatAge = firstCatAge;
2 Cat myCat = new Cat();
3 Cat sameCat = myCat;
4
```

Visually, we can represent these four variables as shown below.



Reference Variables vs. Primitive Variables

Since `int` is a primitive type, the variables `firstCatAge` and `secondCatAge` function like separate boxes, each one holding the integer value `11`. On the other hand, `myCat` is a reference variable, since it refers to an object of type `Cat`. The variable actually stores the *memory address* of the object, which we visualize as an arrow pointing from the variable box to where the data is stored. Instead of holding the actual `Cat` data, `myCat` stores *directions* for finding the data in memory.

When we to assign `myCat` to another variable, as in `Cat sameCat = myCat`, we do NOT create a second copy of the object or its data. Instead, we make a second arrow pointing to the same memory location.

The distinction between object types and primitives is important, if subtle. As you continue learning Java, you will see that object types are handled differently in essential and important ways.

2.3.4. Static Methods

If you are familiar with another programming language, then you most likely defined and called functions. As a pure object-oriented programming language, Java also uses functions, but it structures them in a very specific way.

In Java, functions may NOT be declared outside of a class. Even a simple function that checks if an integer is even needs to be defined within a class.

Within the context of a class, functions are referred to as **methods**, and we will adopt this terminology from now on.

Warning

Be prepared to receive a vocabulary lesson from veteran Java coders if you accidentally refer to *methods* as *functions*.

We'll dive deeper into classes and objects in Java soon enough. For now, we will explore how to write methods. In particular, we'll use **static methods**. A static method is one that can be called without creating an instance of the class to which it belongs.

Example

Define the class `Cat` and include the `static` keyword before the `makeNoise` method name:

```
public class Cat {
   public static void makeNoise(String[] args) {
      // some code
   }
}
```

Since `makeNoise` is `static`, we do NOT need to create a `Cat` object to access it.

Instead of doing this:

```
  Cat myCat = new Cat();      // Create a new Cat object.
1 myCat.makeNoise("purr");    // Call the makeNoise method.
2
```

We can call the method directly:

```
  Cat.makeNoise("roar");
1
```

Until we get further into object oriented programming, every method you write should use the `static` keyword. Leaving off `static` will prevent or complicate the process of calling the methods you defined.

We will explore exactly what `static` does in more detail in later lessons.

2.3.4.1. Static Method Examples

Let's examine two classes in Java to explore defining and using methods. The first class is defined in the `HelloMethods.java` file, and it has a `main` method. The second class is defined in a separate `Message.java` file, and it contains a `getMessage` method that we want to call from within `main`.

Examples

`HelloMethods.java`:

```
1
2
3
4
5
6
7
8
```

```
public class HelloMethods {

   public static void main(String[] args) {
      String message = Message.getMessage("fr");
      System.out.println(message);
   }

}

```

`Message.java`:

```
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
```

```
public class Message {

   public static String getMessage(String lang) {

      if (lang.equals("sp")) {
         return "¡Hola, Mundo!";
      } else if (lang.equals("fr")) {
         return "Bonjour, le monde!";
      } else {
         return "Hello, World!";
      }
   }
}

```

We won't explore every new aspect of this example, but instead focus on the two methods.

1. The `main` method in the `HelloMethods` class has the same structure as that of our temperature conversion example.
2. Take a look at the `Message` class. Note that it does NOT have a `main` method, so it can't be run on it's own. Code within the `Message` class must be called from elsewhere in order to execute.
3. The `Message` class contains the `getMessage` method. Like `main`, it has the `static` keyword. Unlike `main`, `getMessage` has a return type of `String` instead of `void`.
4. `getMessage` takes a single `String` parameter, `lang`.

Since Java is statically typed, we must declare the data type for each parameter AND the return value.

```
public static returnedDataType methodName(parameterDataType parameterName) {
   //code
}
```

One consequence of this is that a method in Java may NOT have `return` statements that send back different types of data. Note that lines 6, 8, and 10 in `Message.java` each return a string. If we try to replace line 10 with `return 42;`, we would generate a compiler error.

To call a static method, we follow a specific syntax. Line 4 in the `HelloMethods.java` shows this:

```
Message.getMessage("fr");
```

To call a static method we must use the format `ClassName.methodName(arguments)`.

Note that `getMessage` is NOT defined within the `HelloMethods` class. We can do this because `getMessage` is declared as `public`. If we wanted to restrict the method from being called by another class, we could instead use the `private` modifier. We will explore access modifiers in more depth in coming lessons.

Warning

As you have been following along with these examples, you may have noticed that each class file, for example `Message.java` and `HelloMethods.java`, is named exactly the same as the class it holds (`Message` and `HelloMethods`, respectively).

It is a rule in Java that a file containing a class marked `public` MUST be named the same as that class.

2.3.4.2. Try It

Open the `HelloMethods` and `Messages` files in the `java-web-dev-exercises` project in IntelliJ and experiment with the following:

1. Figure out how to alter the `HelloMethods` code to change the message returned.
2. Add another "Hello, World" language option.
3. Change one `public` keyword to `private` to see what happens. Repeat for each occurrence of `public`.

2.3.5. References

1. Arrays (docs.oracle.com)

2.3.6. Check Your Understanding

Question

Which of the following defines a method that takes an integer as a parameter and returns a string value?

1. `public static void methodName(String parameterName)`
2. `public static void methodName(int parameterName)`
3. `public static int methodName(String parameterName)`
4. `public static String methodName(int parameterName)`

Question

Assume that we declare the following Java array:

```
String[] someWords = new String[5];
```

Which of the following shows a correct initialization for the array?

1. `someWords = {'hello', 'world', '123', 'LaunchCode ROCKS!'}`
2. `someWords = {"hello", "world", "123", "LaunchCode ROCKS!", "Java"}`
3. `someWords = {"hello", "world", 'a', "LaunchCode ROCKS!", "Java"}`
4. `someWords = {"hello", "world", "avocado", "LaunchCode ROCKS!"}`

2.4. Exercises: Data Types

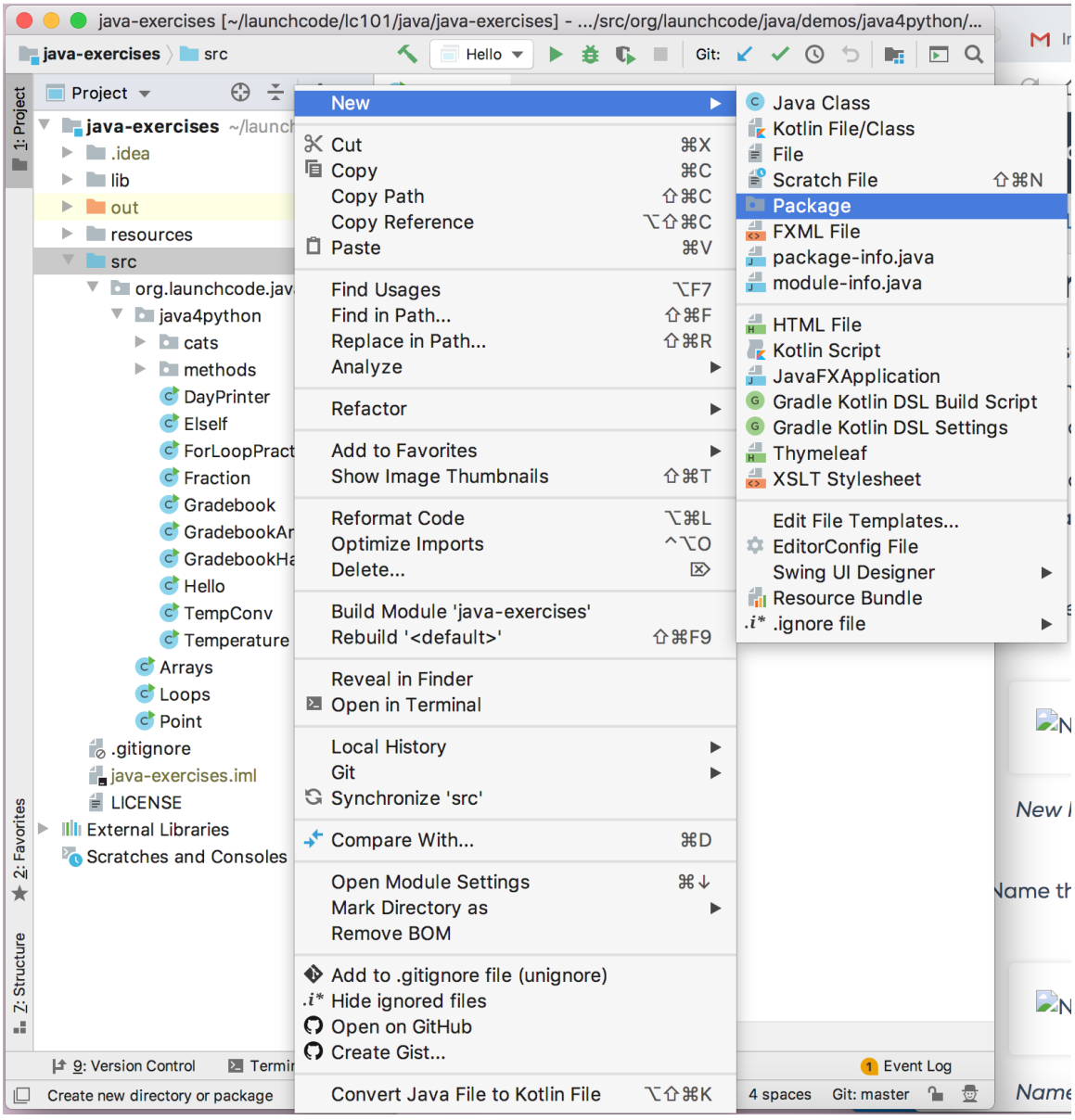
2.4.1. Instructions

Work on these exercises in the IntelliJ `java-web-dev-exercises` project. Create a new class for each numbered exercise. You may name the classes whatever you like, but use proper Java Naming Conventions and make sure that the file name matches the class name.

2.4.1.1. Creating a Package and Classes

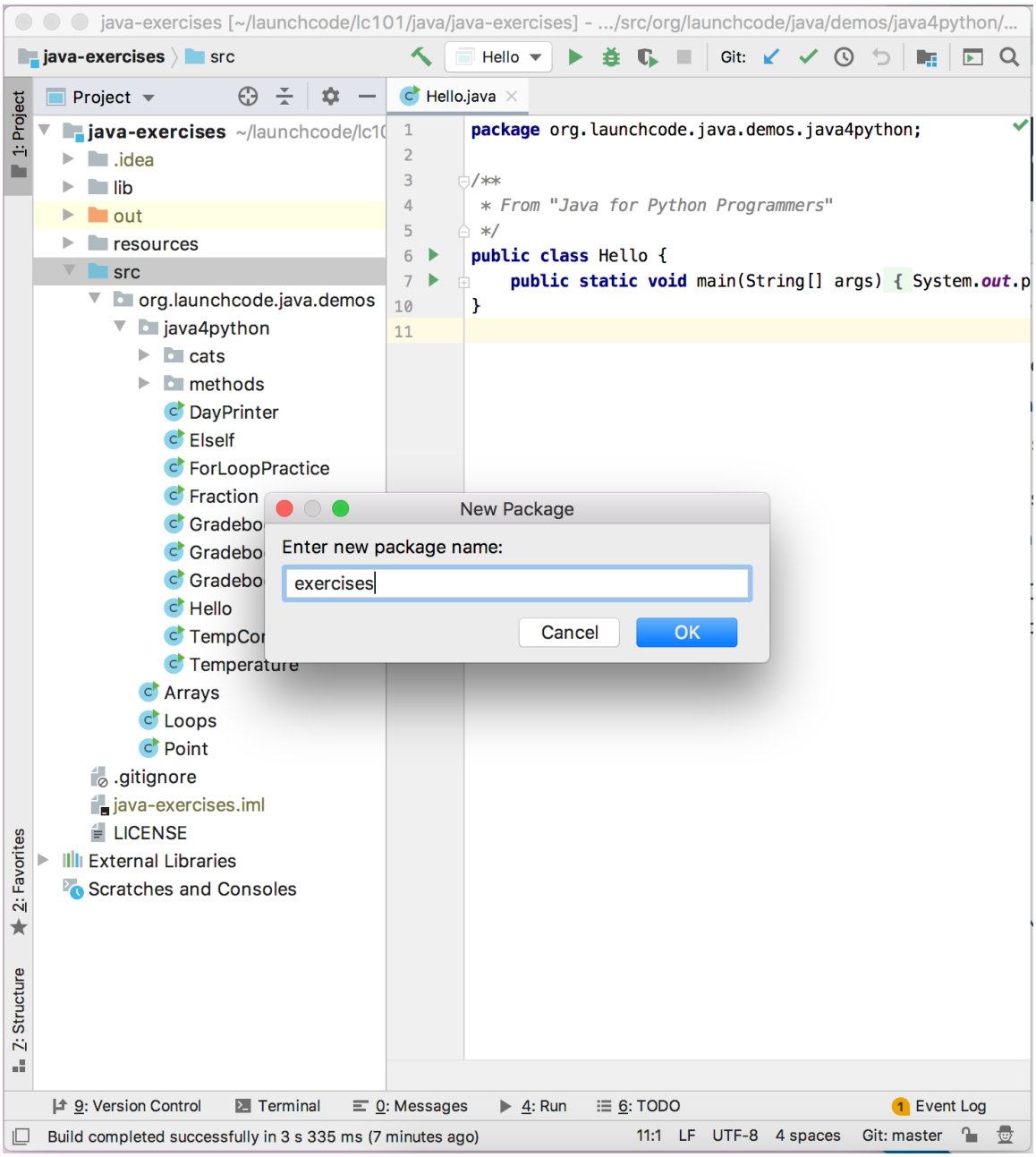
Here is how to create a new package to store these exercises, and how to create new classes within this package:

1. Click on the folder `src` in the Project pane, then right-click (or control-click for some Mac users) and select *New* and then select *Package*.



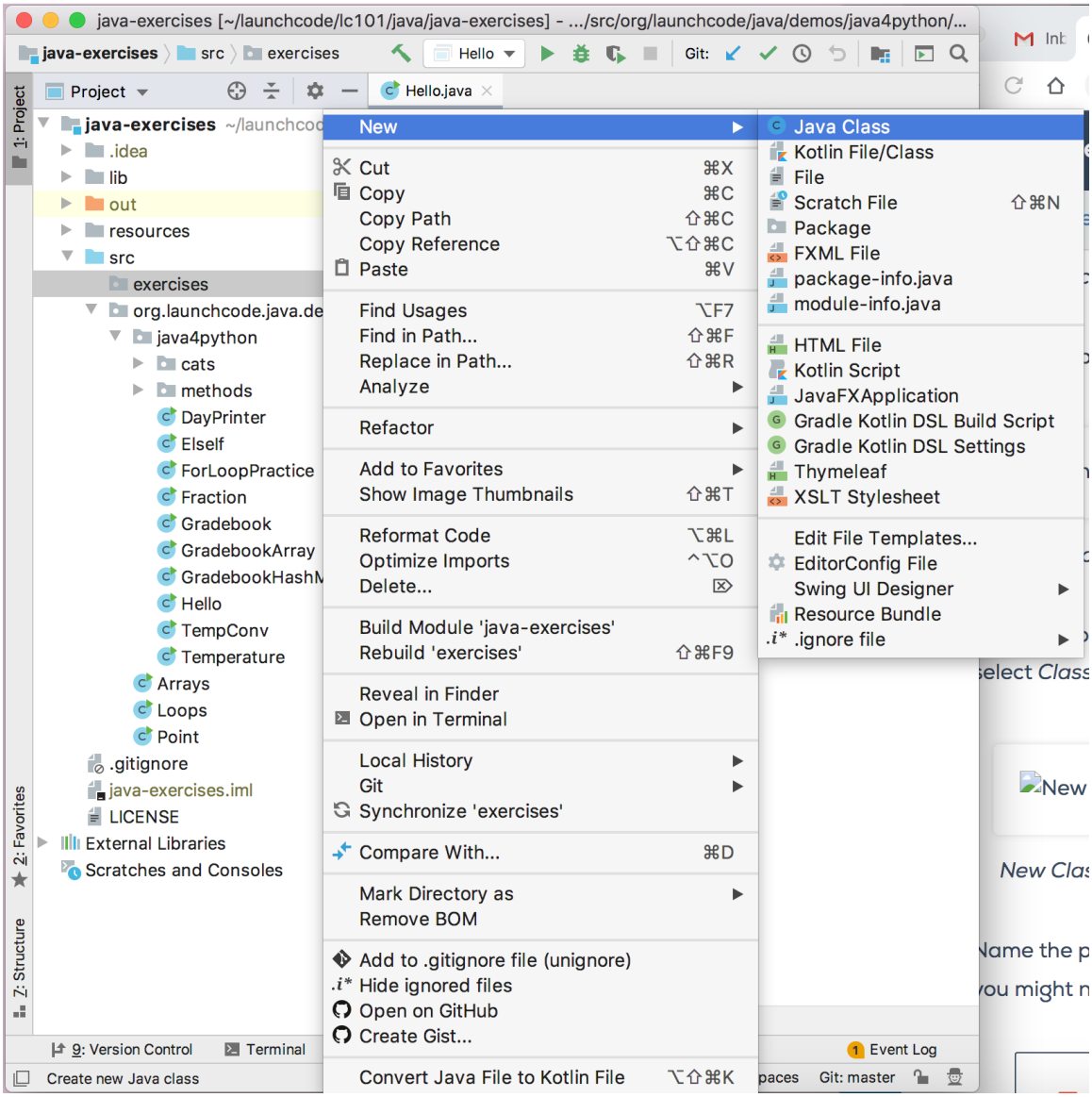
New Package

2. Name the package “exercises”.



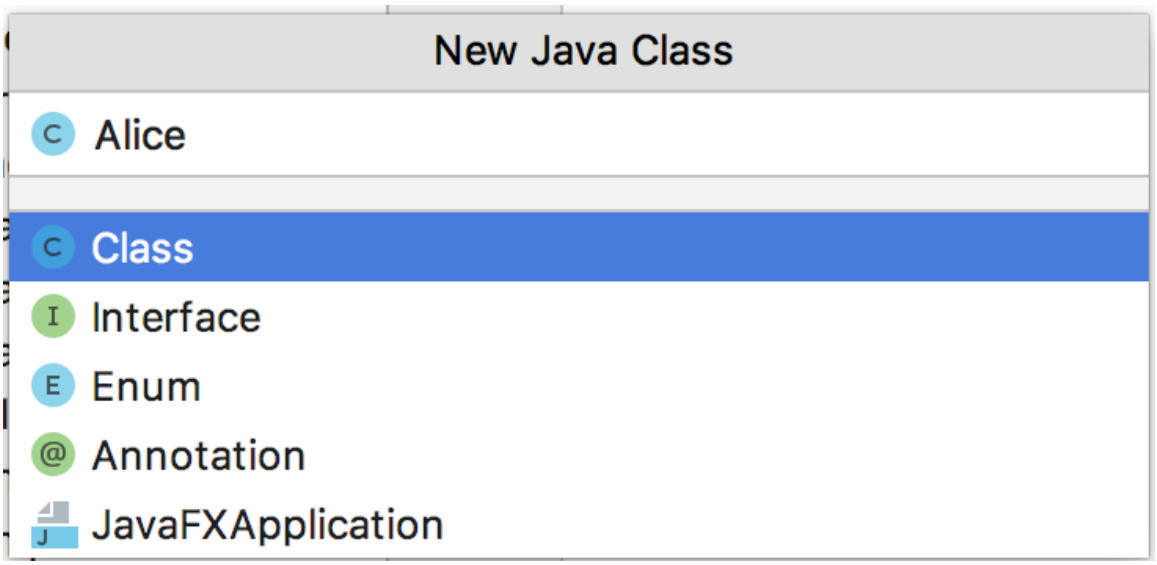
Name Package

3. Right-click/Control-click on the newly created `exercises` folder. Select *New* and then *Java Class*.



New Class

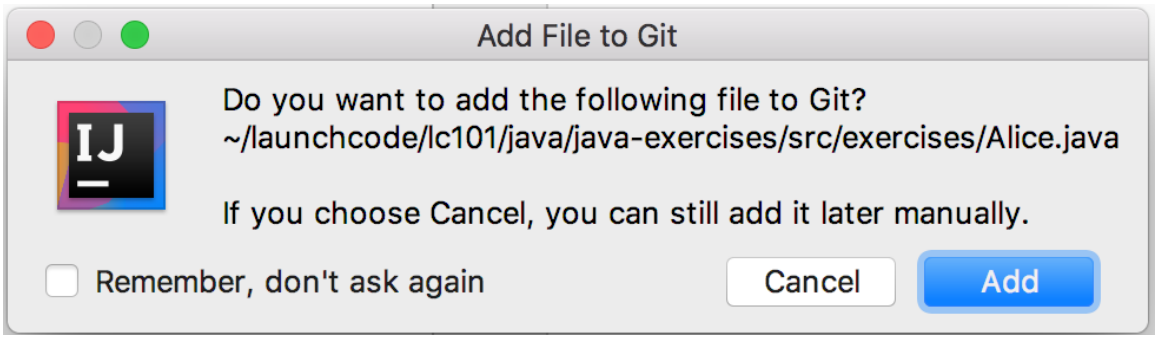
4. Name this what you will name your class (for example, in the 4th exercise below, you might name the class `Alice`).



Name Class

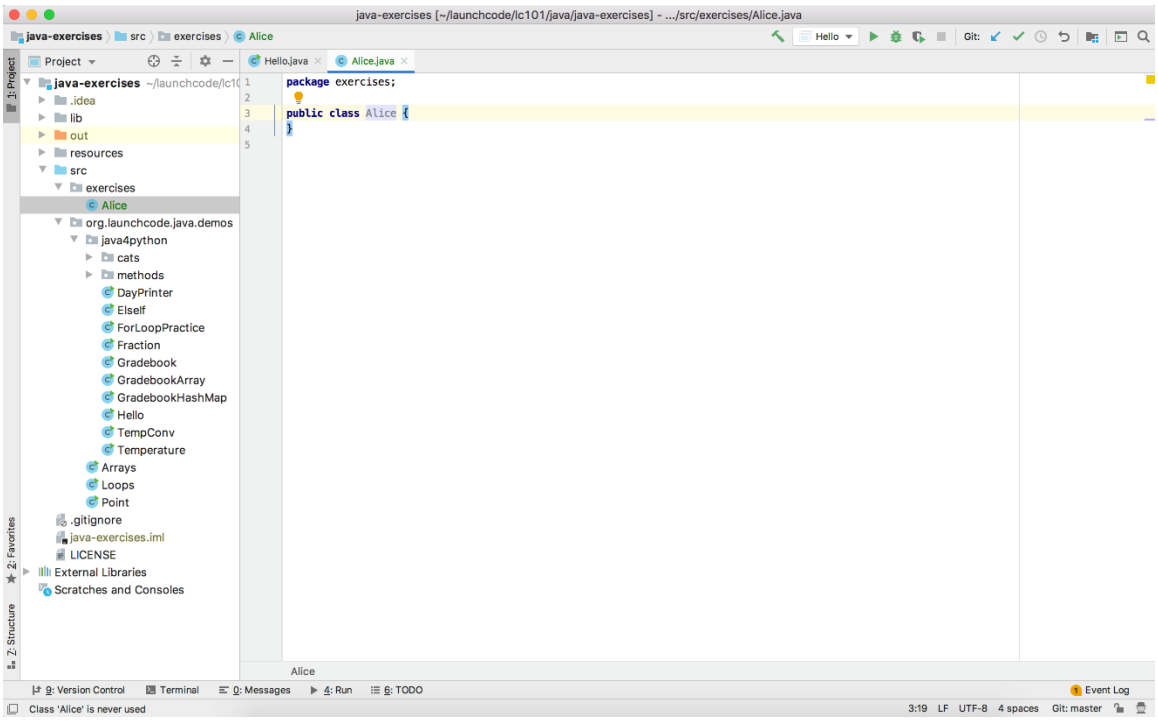
Note

You will be asked whether you want to add this file to Git. Press the "Yes" button.



Add class to Git

5. You created the new class! You can proceed to write code within it. (Don't forget to write the `main` method!)



Ready to start

2.4.2. Exercises

1. **Input/output**: Write a new "Hello, World" program to prompt the user for their name and greet them by name.
    1. Follow steps 3-5 above to create a new `HelloWorld` Class inside of your `exercises` folder.
    2. Add an import statement at the the top of the file to include `Scanner`:
    ```
    3. import java.util.Scanner;
    ```
    4. Declare a variable of type `Scanner` called `input`:
    ```
    5. Scanner input = new Scanner(System.in);
    ```
    6. Add a question to ask the user:
    ```
    7. System.out.println("Hello, what is your name:");
    ```
    8. Create a variable to store the user's response using the `Scanner`'s `.nextLine()` method
    ```
    9. String name = input.nextLine();
    ```
    10. Use concatenation to print the greeting:
    ```
    11.    System.out.println("Hello " + name);
    ```
    12. Right-click/Control-click the arrow next to your class and run the program.
2. **Numeric types**: Write a program to calculate the area of a rectangle and print the answer to the console. You should prompt the user for the dimensions. (What data types should the dimensions be?)
    1. Follow steps 3-5 above to create a new Class inside of your exercises.
    2. Add an import statement at the top of your file to use `Scanner`.
    3. Add a `Scanner` object to handle the user's input.
    4. Add a print line to prompt the user for the length of the rectangle.
    5. Define a variable to handle the user's response. Now is the time to know what type the dimension will be.

        Tip

        You'll need to use a different `Scanner` method than what we used in Exercise 1 above.

    6. Repeat the previous two steps to ask for and store the rectangle's width.
    7. Use the length and width values to calculate the rectangle's area.
    8. Print a statement using concatenation to communicate to the user what the area of their rectangle is.
    9. Run the program to verify your code.
3. **Numeric types**: Write a program that asks a user for the number of miles they have driven and the amount of gas they've consumed (in gallons), and print their miles-per-gallon.
4. **Strings**: The first sentence of *Alice's Adventures in Wonderland* is below. Store this sentence in a string, and then prompt the user for a term to search for within this string. Print whether or not the search term was found. Make the search case-insensitive, so that searching for "alice", for example, prints `true`.

    ```
    Alice was beginning to get very tired of sitting by her sister on the bank, and of
    having nothing to do: once or twice she had peeped into the book her sister was
    reading, but it had no pictures or conversations in it, 'and what is the use of a
    book,' thought Alice 'without pictures or conversation?'
    ```

    Note

    You may want to write the string above on more than one line in your solution. Java 13 and IntelliJ gives you a few options to do so. The easiest, thanks to your IDE, is to press *Enter* as you type the string. IntelliJ will close the string and concatenate it with the next line to create one longer string.

5. **Strings**: Extend the previous exercise. Assume the user enters a word that is in the sentence. Print out its index within the string and its length. Next, remove the word from the string and print the sentence again

to confirm your code. Remember that strings are *immutable*, so you will need to reassign the old sentence variable or create a new one to store the updated phrase.

2.5. Studio: Area of a Circle

Get cosy with Java syntax by writing a console program that calculates the area of a circle based on input from the user.

2.5.1. Creating your class

Since you're still new to Java and IntelliJ, we'll provide some extra direction the first few coding exercises.

First, make a new folder, or package, to hold your studio exercises. Create a new package named `org.launchcode.java.studios.areaofacircle` by right-clicking (or ctrl-clicking for some Mac users) on the `src` directory in `java-web-dev-exercises` and selecting *New* > *Package*. Be sure to enter the full name, or your package won't be created in the correct location.

Create your class in the `java-web-dev-exercises` project within the package `org.launchcode.java.studios.areaofacircle` by right-clicking/ctrl-clicking on the `studios.areaofacircle` package/folder and selecting *New* > *Java Class*. Enter the name `Area`. Select the option to add the file to Git when the window appears.

2.5.2. Your first task

Write a class, `Area`, that prompts the user for the radius of a circle and then calculate its area and print the result.

Tip

Recall that the area of a circle is `A = pi * r * r` where `pi` is 3.14 and `r` is the radius.

Note

Unlike some other languages, Java does not have an exponent operator.

Here's an example of how your program should work:

```
Enter a radius: 2.5
The area of a circle of radius 2.5 is: 19.625
```

Some questions to ask yourself:

1. What data type should the radius be?
2. What is the best way to get user input into a variable `radius` of that type?

Warning

Be sure to create a `main` method to place your code within. It's signature *must* be:

```
public static void main(String[] args)
```

2.5.3. Your next task

Add a second Java file to your program to delegate the area calculation away from the printing task.

1. Add a new class in your `studios.areaofacircle` package called `Circle`.
2. Create a method called `getArea` inside of `Circle` that takes a Double `radius` as its only parameter and returns another `Double`, the result of the area calculation.

```
3. public static Double getArea(Double radius) {
4.   return 3.14 * radius * radius;
5. }
```

6. Back in `Area`, replace your area calculation line with a call to `Circle.getArea()`.

   Tip

   Check out the `HelloMethods` and `Message` example from Static Methods for a reference on how to use a class from another file.

2.5.4. Bonus Missions

1. Add validation to your program. If the user enters a negative number? a non-numeric character? the empty string? Print an error message and quit. You'll need to peek ahead to learn about conditional syntax in Java.
2. Extend your program further by using a while or do-while loop, so that when the user enters a negative number they are re-prompted.

3.1. Conditionals

Control flow statements in Java — conditionals and loops — are very straightforward.

3.1.1. `if` Statements

Let's consider an **if statement** with no `else` clause.

In Java this pattern is simply written as:

```
  if (condition) {
1    statement1
2    statement2
3    ...
4 }
5
```

You can see that in Java the curly braces define a block. Parentheses around the condition are required.

3.1.2. `if else`

Adding an **else clause**, we have:

```
  if (condition) {
1    statement1
2    statement2
3    ...
4 } else {
5    statement1
6    statement2
7    ...
8 }
9
```

3.1.3. `else if`

An **else if** construction in Java:

```
   import java.util.Scanner;
 1
 2 public class ElseIf {
 3     public static void main(String args[]) {
 4         Scanner in = new Scanner(System.in);
 5         System.out.println("Enter a grade: ");
 6         int grade = in.nextInt();
 7         if (grade < 60) {
 8             System.out.println('F');
 9         } else if (grade < 70) {
10             System.out.println('D');
11         } else if (grade < 80) {
12             System.out.println('C');
13         } else if (grade < 90) {
14             System.out.println('B');
15         } else {
16             System.out.println('A');
17         }
18     }
19 }
20
```

3.1.4. `switch` Statements

Java also supports a **switch** statement that acts something like an `else if` statement under certain conditions, called **cases**. The `switch` statement is not used very often, and we generally recommend you avoid using it. It is not as powerful as the `else if` model because the `switch` variable can only be compared for equality with a very small class of types.

Here is a quick example of a `switch` statement:

```
  import java.util.Scanner;
 1
 2 public class DayPrinter {
 3    public static void main(String[] args) {
 4       Scanner in = new Scanner(System.in);
 5       System.out.println("Enter an integer: ");
 6       int dayNum = in.nextInt();
 7
 8       String day;
 9       switch (dayNum) {
10          case 0:
11             day = "Sunday";
12             break;
13          case 1:
14             day = "Monday";
15             break;
16          case 2:
17             day = "Tuesday";
18             break;
19          case 3:
20             day = "Wednesday";
21             break;
22          case 4:
23             day = "Thursday";
24             break;
25          case 5:
26             day = "Friday";
27             break;
28          case 6:
29             day = "Saturday";
30             break;
31          default:
32             // in this example, this block runs if none of the above blocks match
33             day = "Int does not correspond to a day of the week";
34       }
35       System.out.println(day);
36    }
37 }
38
```

In the example above, here's the output if a user enters the number `4`.

```
Enter an integer: 4
Thursday
```

And the output if that user enters `10`? Below:

```
Enter an integer: 10
Int does not correspond to a day of the week
```

Here's how the above example looks using the `else if` construction:

```
  import java.util.Scanner;
 1
 2 public class DayPrinter {
 3    public static void main(String[] args) {
 4       Scanner in = new Scanner(System.in);
 5       System.out.println("Enter an integer: ");
 6       int dayNum = in.nextInt();
 7
 8       String day;
 9       if (dayNum == 0) {
10         day = "Sunday";
11       } else if (dayNum == 1){
12         day = "Monday";
13       } else if (dayNum == 2){
14         day = "Tuesday";
15       } else if (dayNum == 3){
16         day = "Wednesday";
17       } else if (dayNum == 4){
18         day = "Thursday";
19       } else if (dayNum == 5){
20         day = "Friday";
21       } else if (dayNum == 6){
22        day = "Saturday";
23       } else {
24        day = "Int does not correspond to a day of the week";
25       }
26       System.out.println(day);
27    }
28 }
29
```

3.1.4.1. Fallthrough

Additionally, if **break statements** are omitted from the individual cases on accident, a behavior known as fallthrough is carried out. **Fallthrough** can be quite unintuitive, and is only desirable in very specific circumstances. We will discuss `break` statements in more detail in the loop section below. For now, just know that when used in a `switch` block, they terminate the `switch` statement they are in, so the flow of control in your program moves to the next statement after the switch block.

Here's a quick example of how fallthrough works:

```
  import java.util.Scanner;
 1
 2 public class DayPrinter {
 3    public static void main(String[] args) {
 4
 5       System.out.println("Enter an integer: ");
 6       Scanner in = new Scanner(System.in);
 7       int dayNum = in.nextInt();
 8
 9       String day;
10       switch (dayNum) {
11          case 0:
12             day = "Sunday";
13          case 1:
14             day = "Monday";
15          case 2:
16             day = "Tuesday";
17          case 3:
18             day = "Wednesday";
19          case 4:
20             day = "Thursday";
21          case 5:
22             day = "Friday";
23          case 6:
24             day = "Saturday";
25          default:
26             // in this example, this block runs even if one of the above blocks match
27             day = "Int does not correspond to a day of the week";
28       }
29       System.out.println(day);
30    }
31 }
32
```

This time, without the `break` statements in each `case`, if the user enters `4`, they will see the default output:

```
Enter an integer: 4
Int does not correspond to a day of the week
```

This is because after the `switch` statement matches the `case` for `4` and assigns the value `Thursday` to the variable `day`, it proceeds to execute every statement in every case that follows, all the way through the `default` case. So the `String` that ends up being printed will reflect the last executed statement in the `switch` block.

Along similar lines, consider this variation on the code block above:

```
  import java.util.Scanner;
 1
 2 public class DayPrinter {
 3    public static void main(String[] args) {
 4
 5       System.out.println("Enter an integer: ");
 6       Scanner in = new Scanner(System.in);
 7       int dayNum = in.nextInt();
 8
 9       String day;
10       switch (dayNum) {
11          case 0:
12             day = "Sunday";
13          case 1:
14             day = "Monday";
15          case 2:
16             day = "Tuesday";
17          case 3:
18             day = "Wednesday";
19          case 4:
20             day = "Thursday";
21          case 5:
22             day = "Friday";
23          case 6:
24             day = "Saturday";
25             break;
26          default:
27             day = "Int does not correspond to a day of the week";
28       }
29       System.out.println(day);
30    }
31 }
32
```

Here, we have a `break` statement in `case 6` after `day = "Saturday";`. If the user enters `4`, the execution will fallthrough until it reaches that `break` statement and `Saturday` is printed instead of `Thursday`. The output:

```
Enter an integer: 4
Saturday
```

3.1.5. References

- The if-then and if-then-else Statements (docs.oracle.com)
- The switch Statement (docs.oracle.com)

3.1.6. Check Your Understanding

Question

When does fallthrough occur in Java?

1. Omitting an `else` clause from a conditional.
2. Omitting an `else` clause from switch statement.
3. Omitting a `default` case from a `switch` statement.
4. Omitting a `break` line from a `switch` statement.

Question

```
  import java.util.Scanner;
 1
 2 public class QuizQuestion {
 3    public static void main(String[] args) {
 4
 5       System.out.println("Are you a space cadet? yes or no");
 6       Scanner in = new Scanner(System.in);
 7       String response = in.next();
 8
 9       switch (response) {
10          case "yes":
11             System.out.println("Greetings cadet.");
12          case "no":
13             System.out.println("Greetings normie.");
14          default:
15             System.out.println("Are you an alien?");
16       }
17    }
18 }
19
```

Given the code above, what prints if the user enters `no` after the prompt?

1. `Greetings cadet.`
2.
3. `Greetings normie.`
4.
5. `Greetings normie.`
6. `Are you an alien?`
7.
8. `Greetings cadet.`
9. `Greetings normie.`
10.

3.2. Loops

3.2.1. `for` Loop

In Java we write a definite loop (aka a **for loop**) as:

```
  for (int i = 0; i < 10; i++ ) {
1    System.out.println(i);
2 }
3
```

Output:

```
0
1
2
3
4
5
6
7
8
9
```

Note

You may not be familiar with the expression `i++` since it is not found in all languages. The `++` is an increment operator that has the same effect as `i += 1`. In this example, since the `++` comes after `i`, we call it a postfix increment operator. There is also a `--` decrement operator in Java. For more information, see Increment and Decrement Operators.

The Java `for` loop gives you explicit control over the starting, stopping, and stepping of the loop variable inside the parentheses. You can think of it this way:

```
  for (start clause; stop clause; step clause) {
1    statement1
2    statement2
3    ...
4 }
5
```

If you want to start at 100, stop at 0 and count backward by 5, the loop is written as:

```
  for (int i = 100; i >= 0; i -= 5) {
1    System.out.println(i);
2 }
3
```

Output:

```
100
95
90
...
```

3.2.2. `for-each` Loop

Java also provides a syntax to iterate over any sequence or collection, such as an Array:

```
  int nums[] = {1, 1, 2, 3, 5, 8, 13, 21};
1
2 for (int i : nums) {
3    System.out.println(i);
4 }
5
```

Here, the loop variable moves through the items in the Array of integers, `nums[]`. The syntax here uses a colon symbol, `:`. This type of loop is known as a **for-each loop**.

Tip

When considering this structure, it can be helpful to read the code sample above to yourself as "For each integer in `Array nums...`".

This loop version also works with a String, where we can convert the String to an Array of characters:

```
  String msg = "Hello World";
1
2 for (char c : msg.toCharArray()) {
3    System.out.println(c);
4 }
5
```

As you see, to iterate through a String in this way, Java requires an extra String method, `.toCharArray()`, to convert the String to an Array of characters.

3.2.3. `while` Loop

Java also supports the **while loop**, or indefinite loop. A `while` loop in Java:

```
  int i = 0;
1 while (i < 3) {
2    i++;
3 }
4
```

3.2.4. `do-while` Loop

Java adds an additional, if seldom used, variation of the `while` loop called the **do-while loop**. The `do-while` loop is very similar to `while` except that the condition is evaluated at the end of the loop rather than the

beginning. This ensures that a loop *will be executed at least one time*. Some programmers prefer this loop in some situations because it avoids an additional assignment prior to the loop.

For example:

```
  do {
1    System.out.println("Hello, World");
2 } while (false);
3
```

Output:

```
Hello, World
```

Above, the message prints despite the condition never being met.

3.2.5. Break Statements in Loops

There are instances where you may want to terminate a loop if a given condition is met. In these instances, the `break` statement comes in handy. For example, say you want to loop through an Array of integers to search for a given value. Once that number is found, you want to quit the loop. You can do the following:

```
   public class testBreak {
 1
 2    public static void main(String [] args) {
 3       int[] someInts = {1, 10, 2, 3, 5, 8, 10};
 4       int searchTerm = 10;
 5       for (int oneInt : someInts) {
 6          if (oneInt == searchTerm) {
 7             System.out.println("Found it!");
 8             break;
 9          }
10       }
11    }
12 }
13
```

In the code above, instead of the `for` loop iterating through all the integers in the array, it will stop after it finds the first matching instance. So once it finds the first `10` in the array, it prints “Found it!” and then terminates the loop. If the `break` statement weren’t there, the loop would continue and when it found the second `10`, it would print “Found it!” a second time.

Note that the `break` statement terminates the innermost loop that it is contained within. So if you have nested loops and use a `break` statement within the innermost loop, then it will only terminate that loop and not the outer one. If a `break` is present in the outer loop, it — and any other block nested within it — is terminated when the `break` runs.

3.2.6. Continue Statements in Loops

The **continue** statement is similar to, but importantly different from, the `break` statement. Like `break`, it interrupts the normal flow of control of the loop. But unlike `break`, the `continue` statement only terminates the

current iteration of the loop. So the loop will continue to run from the top (as long as the boolean expression that controls the loop is still true) after a `continue` statement. Here is an example:

```
  public class testContinue {
1
2    public static void main(String [] args) {
3       int[] someInts = {1, 10, 2, 3, 5, 8, 10};
4       int searchTerm = 10;
5       for (int oneInt : someInts) {
6          if (oneInt == searchTerm) {
7             System.out.println("Found it!");
8             continue;
9          }
10         System.out.println("Not here");
11      }
12   }
13 }
14
```

The above program will print "Not here" on every iteration of the `for` loop *except* where the number has been found. So the output looks like this:

```
Not here
Found it!
Not here
Not here
Not here
Not here
Found it!
```

Because of the `continue` statement, the final print statement in the for loop is skipped. If the `continue` statement weren't there, the output would look like this instead (notice the extra "Not here" printouts):

```
Not here
Found it!
Not here
Not here
Not here
Not here
Not here
Found it!
Not here
```

3.2.7. References

- The for statement (docs.oracle.com)
- The while and do-while Statements (docs.oracle.com)
- Break and Continue Statements (docs.oracle.com)
- Summary of Control Flow Statements (docs.oracle.com)

3.2.8. Check Your Understanding

Question

```
  char chars[] = {'p', 'l', 'r', 's', 't'};
1
2 for (<loop-statement>) {
3    System.out.println(i);
4 }
5
```

What does the missing <loop-statement> need to be to print each item in `chars`?

1. `char i : chars`
2. `char i : chars[]`
3. `char i in chars`
4. `char i in chars[]`

Question

```
  do {
1    System.out.println("Hello world!");
2 } while (3 < 2);
3
```

How many times does the message print and why?

1. 0 — The `while` condition is never true.
2. 1 — The print statement is evaluated before the conditional.
3. infinite times — 3 is less than 2, and the condition is never changed in the loop.

3.3. Collections

3.3.1. Data Structures

A **data structure** lets us hold on to lots of data in a single place. It is a programming construct to aggregate lots of values into one value. Many types of data structures exist in various languages. A few examples are lists, dictionaries, arrays, tuples, etc.

3.3.2. Java Collections Framework

Java provides powerful and flexible structures to store data, known as **collections**. The **Java collections framework** refers to the various interfaces the language provides for implementing collection types.

Here, we'll discuss a collection called `ArrayList` and compare it to the `Array` class. We'll then introduce a third collection type called `HashMap`. These three collection types will be sufficient for our basic Java needs. For more, refer to the official Java documentation on collections.

3.3.3. Gradebook, Three Ways

We'll explore collections in Java by looking at different versions of the same program. The program functions as a gradebook, allowing a user (a professor or teacher) to enter the class roster for a course, along with each student's grade. It then prints the class roster along with the average grade. In each variation of this program, the grading system could be anything numeric, such as a 0.0-4.0 point scale, or a 0-100 percentage scale.

A test run of the program might yield the following:

```
Enter your students (or ENTER to finish):
Chris
Jesse
Sally

Grade for Chris: 3.0
Grade for Jesse: 4.0
Grade for Sally: 3.5

Class roster:
Chris (3.0)
Jesse (4.0)
Sally (3.5)

Average grade: 3.5
```

We'll look at the gradebook using an `Arraylist` first.

3.4. ArrayList

To write an **ArrayList** version of the program, we will have to introduce several new Java concepts, including the class `ArrayList`. We will also review different kinds of `for` loops used in Java.

Before going any further, we suggest you run the `ArrayListGradebook` program in IntelliJ. You can view this program in `java-web-dev-exercises`. Once you've done that, let's look at what is happening in the Java source code.

```
  package org.launchcode.java.demos.collections;
 1
 2 import java.util.ArrayList;
 3 import java.util.Scanner;
 4
 5 public class ArrayListGradebook {
 6
 7    public static void main(String[] args) {
 8
 9       ArrayList<String> students = new ArrayList<>();
10       ArrayList<Double> grades = new ArrayList<>();
11       Scanner input = new Scanner(System.in);
12       String newStudent;
13
14       System.out.println("Enter your students (or ENTER to finish):");
15
16       // Get student names
17       do {
18          newStudent = input.nextLine();
19
20          if (!newStudent.equals("")) {
21             students.add(newStudent);
22          }
23
24       } while(!newStudent.equals(""));
25
26       // Get student grades
27       for (String student : students) {
28          System.out.print("Grade for " + student + ": ");
29          Double grade = input.nextDouble();
30          grades.add(grade);
31       }
32
33       // Print class roster
34       System.out.println("\nClass roster:");
35       double sum = 0.0;
36
37       for (int i = 0; i < students.size(); i++) {
38          System.out.println(students.get(i) + " (" + grades.get(i) + ")");
39          sum += grades.get(i);
40       }
41
42       double avg = sum / students.size();
43       System.out.println("Average grade: " + avg);
44    }
45 }
46
```

Here we declare and initialize two objects, `students` and `grades`, which appear to be of type `ArrayList<String>` and `ArrayList<Double>`, respectively. An `ArrayList` in Java is very similar to an Array. Like an `Array`, we must let the compiler know what kind of objects our `ArrayList` is going to contain. In the case of `students`, the `ArrayList` will contain values of type `String` (representing the names of the students), so we use the `ArrayList<String>` syntax to inform the compiler that we intend to fill our list with Strings. Similarly, `grades` will hold exclusively values of type `Double` and is declared to be of type `ArrayList<Double>`.

Warning

Notice that we declared `grades` to be of type `ArrayList<Double>`, using the wrapper class `Double` rather than the primitive type `double`. All values stored in Java collections must be objects, so we'll have to use object types in those situations.

In lines 10 and 11, we also initialize each list by creating a new, empty list. Note that when we call the `ArrayList` constructor, as in `new ArrayList<>()`, we don't need to specify type (it's implicit in the left-hand side of the assignment).

Note

You will sometimes see the `ArrayList` class written as ArrayList<E>, where `E` represents a placeholder for the type that a programmer will declare a given list to hold. This is especially true in documentation. You can think of `E` as representing an arbitrary type.

3.4.1. `ArrayList` Iteration

3.4.1.1. `do-while`

We then use a `do-while` loop to collect the names of each of the students in the class.

```
   // Get student names
18 do {
19    newStudent = input.nextLine();
20
21    if (!newStudent.equals("")) {
22       students.add(newStudent);
23    }
24
25 } while(!newStudent.equals(""));
26
```

Recall that a `do-while` loop is very similar to a `while` loop, but the execution condition is checked at the end of the loop block. This has the net effect that the code block will always run at least once. In this example, we prompt the user for a name, which Java processes via `input.nextLine()` when the user hits the enter key. To finish entering names, the user enters a blank line.

For each student that is entered (that is, each non-empty line), we add the new `String` to the end of our list with `students.add(newStudent)`. The `.add()` method is provided by the ArrayList Class. There are lots of other ArrayList methods to get familiar with, some of which we will discuss in more detail below.

Note that our program imports `java.util.ArrayList` to take advantage of this Java provided class.

3.4.1.2. `for-each`

Below the `do-while` loop are two different loops that demonstrate two ways you can loop through a list in Java. Here's the first, which collects the numeric grade for each student:

```
   // Get student grades
27 for (String student : students) {
28    System.out.print("Grade for " + student + ": ");
29    Double grade = input.nextDouble();
30    grades.add(grade);
31 }
32
```

This, you may recall, is Java's `for-each` loop syntax. You may read this in your head, or even aloud, as: `for each student in students`. As you might expect at this point, we must declare the iterator variable `student` with its data type.

3.4.1.3. `for`

The next loop on display prints out each student's name and grade:

```
   // Print class roster
34 System.out.println("\nClass roster:");
35 double sum = 0.0;
36
37 for (int i = 0; i < students.size(); i++) {
38    System.out.println(students.get(i) + " (" + grades.get(i) + ")");
39    sum += grades.get(i);
40 }
41
```

Here, we introduce the syntax `students.size()` which utilizes the `size()` method of `ArrayList`. This method returns the integer representing the number of items in the list. This is similar to String's `.length()` method.

In this `for` loop, we use a *loop index* to define the starting point, ending point, and increment for iteration. It may be helpful for you to consider this kind of construction as something like, `for integer i in the range of the number of items in students...`. The first statement inside the parenthesis declares and initializes a loop index variable `i`. The second statement is a Boolean expression that is our exit condition. In other words, we will keep looping as long as this expression evaluates to `true`. The third statement is used to increment the value of the loop index variable at the end of iteration through the loop.

Again, the syntax `i++` is Java shorthand for `i = i + 1`. Java also supports the shorthand `i--` to decrement the value of `i`. We can also write `i += 2` as shorthand for `i = i + 2`.

In the final lines of the program, we compute the average grade for all students:

```
   double avg = sum / students.size();
43 System.out.println("Average grade: " + avg);
44
```

3.4.2. ArrayList Methods

Let's gather up a few of the `ArrayList` methods that we've encountered so far, along with a few new ones. While these will be the most common methods and properties that you use with this class, they by no means represent a complete list. Refer to the official documentation on the ArrayList class for such a list, and for more details.

To demonstrate the use of these methods, we'll create a new `ArrayList` called `planets`.

```
ArrayList<String> planets = new ArrayList<>();
```

Ok, we've got an empty ArrayList. We need to use the class's `.add()` method to populate this collection with items.

Note

There are other means to declare and initialize an ArrayList in fewer lines. These require knowledge of other collections types, so we'll stick with `.add()` for the time being.

Using `.add()` to populate `planets`:

```
  planets.add("Mercury");
1 planets.add("Venus");
2 planets.add("Earth");
3 planets.add("Mars");
4 planets.add("Jupiter");
5 planets.add("Saturn");
6 planets.add("Uranus");
7 planets.add("Neptune");
8
```

Thus, the first item in this table:

ArrayList methods in Java

Java Syntax	Description	Example
`add()`	Adds an item to the ArrayList	`planets.add("Pluto")` adds `Pluto` to `planets`
`size()`	Returns the number of items in an ArrayList, as an `int`	`planets.size()` returns `9`
`contains()`	Checks to see if the ArrayList contains a given item, returning a Boolean	`planets.contains("Earth")` returns `true`
`indexOf()`	Looks for an item in n ArrayList, returns the index of the first occurrence of the item if it exists, returns -1 otherwise	`planets.indexOf("Jupiter")` returns `4`

Here's a couple more methods that require slightly longer descriptions:

Collections.sort()

Java Syntax	Description	Example
`Collections.sort()`	Sorts a Collection in ascending order, returns the sorted Collection	`Collections.sort(planets)` returns `["Earth", "Jupiter", "Mars", "Mercury", "Neptune", "Pluto", "Saturn", "Uranus", "Venus"]`

This method is technically used on Java's `Collections` class and thus requires a different `import` statement:

```
import java.util.Collections;
```

`Collections` is itself a member of the collections framework but not all members of the framework are instances of this class. We include this method here because, should you be in the market for a sorting method, this is a helpful one to know.

toArray()

Java Syntax	Description	Example
`toArray()`	Returns an Array containing the elements of the ArrayList	`planets.toArray(planetsArray)` returns `{"Earth", "Jupiter", "Mars", "Mercury", "Neptune", "Pluto", "Saturn", "Uranus", "Venus"}`

Perhaps you recall that in Java, you must know the size of the Array when you create it. So we'll need to first define the new Array before we can use `toArray()`.

```
  String planetsArr[] = new String[planets.size()];
1 planets.toArray(planetsArr);
2
```

Speaking of Arrays, let's see the Array version of Gradebook next.

3.4.3. References

- Java Collections (docs.oracle.com)
- ArrayList Class (docs.oracle.com)

3.4.4. Check Your Understanding

Question

The number of entries in an `ArrayList` may not be modified.

1. True
2. False

Question

Create an `ArrayList` called `charStars` containing `a`, `b`, and `c`.

1.

```
 ArrayList<String> charStars = new ArrayList<>();
1chars.add('a');
2chars.add('b');
3chars.add('c');
4
```

2.

```
 ArrayList<Char> charStars = new ArrayList<>();
1chars.add('a');
2chars.add('b');
3chars.add('c');
4
```

3. `ArrayList<char> charStars = new ArrayList<char>('a', 'b', 'c');`
4.
5.

```
 ArrayList<String> charStars = new ArrayList<>();
1chars.add("a");
2chars.add("b");
3chars.add("c");
4
```

3.5. Array

We learned about arrays in Java in a previous lesson, so let's spend a moment comparing them to `ArrayLists`. `ArrayLists` are generally easier to use that Java's `Array`. Let's see why this is.

Why does Java have both `Arrays` and `ArrayLists`? The answer is historical, at least in part. Java is a C-style language, and arrays are the most basic data structure in C. Using an `Array` over an `ArrayList` might be preferred in some circumstances, primarily for performance reasons (array operations are generally faster than `ArrayList` operations). Also note that *Arrays are of fixed size*. You cannot expand or contract an `Array` after it is created, so you must know exactly how many elements it will need to hold when you create it. This fact is reason enough to use `ArrayLists` in most scenarios.

To illustrate `Array` usage, here is a version of the Gradebook program using `Arrays` instead of `ArrayLists`:

```
  package org.launchcode.java.demos.collections;
 1
 2 import java.util.Scanner;
 3
 4 public class ArrayGradebook {
 5
 6    public static void main(String[] args) {
 7
 8       // Allow for at most 30 students
 9       int maxStudents = 30;
10
11       String[] students = new String[maxStudents];
12       double[] grades = new double[maxStudents];
13       Scanner input = new Scanner(System.in);
14
15       String newStudent;
16       int numStudents = 0;
17
18       System.out.println("Enter your students (or ENTER to finish):");
19
20       // Get student names
21       do {
22          newStudent = input.nextLine();
23
24          if (!newStudent.equals("")) {
25             students[numStudents] = newStudent;
26             numStudents++;
27          }
28
29       } while(!newStudent.equals(""));
30
31       // Get student grades
32       for (int i = 0; i < numStudents; i++) {
33          System.out.print("Grade for " + students[i] + ": ");
34          double grade = input.nextDouble();
35          grades[i] = grade;
36       }
37
38       // Print class roster
39       System.out.println("\nClass roster:");
40       double sum = 0.0;
41
42       for (int i = 0; i < numStudents; i++) {
43          System.out.println(students[i] + " (" + grades[i] + ")");
44          sum += grades[i];
45       }
46
47       double avg = sum / numStudents;
48       System.out.println("Average grade: " + avg);
49    }
50
51 }
52
```

Note that we have to decide up front how large our `Arrays` `students` and `grades` are going to be. Thus, this program sets an arbitrary maximum amount of students, likely larger than any user will enter. It may seem obvious, then, that `Array` has no equivalent add() method. The only way to access and alter an element in an `Array` is with **bracket notation**, using an explicit index. For example, gradebook defines a counter variable,

`numStudents`. When the first student is entered by the user, the value is stored in `newStudent`. If the value is not the empty string, then the item in `students` at position 0, the initial value of `numStudents` is assigned to the `newStudent` value. The next time the `do-while` loop executes, the value of `students` at position 1 will be assigned. And so on… Because we must always access and assign `Array` elements using an explicit index, our code can seem littered with `Array` counter variables (like our friends `i` and `j`) and is more difficult to read (not to mention more error-prone).

Like `ArrayLists`, however, we can loop through an `Array` using a `for-each` loop as long as we don't need to use the index of the current item. If we only wanted to print each student's name, and not their grade, at the end of our program, we could do the following:

```
  for (String student : students) {
1    System.out.println(student);
2 }
3
```

We'll use `Arrays` in Java from time-to-time, but for the most part you should rely on `ArrayLists` to store collections of values, or ordered data.

3.5.1. References

- Arrays Tutorial (docs.oracle.com)

3.5.2. Check Your Understanding

Question

`Array` size and element values cannot be changed once defined.

1. True
2. False

Question

Given the `Array` below, which of the following options is a valid action?

```
int[] randomNumbers = new int[5];
```

1. `randomNumbers.add(3);`
2. `randomNumbers.add("one");`
3. `randomNumbers[0] = "three";`
4. `randomNumbers[0] = 1;`

3.6. HashMap

Java also provides us a structure to store data as key/value pairs. Java calls these objects **hashmaps** (or **maps**, more generally), and they are provided by the `HashMap` class.

Considering the gradebook example, we can improve our program using a map. We'll store the students' grades along with their names in the same data structure. The names will be the keys, and the grades will be the values.

As with the other collection structures, in Java we must specify the types of the objects we'll be storing when we declare a variable or parameter to be a map. This means specifying both key and value data types, which are allowed to be different types for a given map.

```
 1 package org.launchcode.java.demos.collections;
 2
 3 import java.util.HashMap;
 4 import java.util.Map;
 5 import java.util.Scanner;
 6
 7 public class HashMapGradebook {
 8
 9    public static void main(String[] args) {
10
11       HashMap<String, Double> students = new HashMap<>();
12       Scanner input = new Scanner(System.in);
13       String newStudent;
14
15       System.out.println("Enter your students (or ENTER to finish):");
16
17       // Get student names and grades
18       do {
19
20          System.out.print("Student: ");
21          newStudent = input.nextLine();
22
23          if (!newStudent.equals("")) {
24             System.out.print("Grade: ");
25             Double newGrade = input.nextDouble();
26             students.put(newStudent, newGrade);
27
28             // Read in the newline before looping back
29             input.nextLine();
30          }
31
32       } while(!newStudent.equals(""));
33
34       // Print class roster
35       System.out.println("\nClass roster:");
36       double sum = 0.0;
37
38       for (Map.Entry<String, Double> student : students.entrySet()) {
39          System.out.println(student.getKey() + " (" + student.getValue() + ")");
40          sum += student.getValue();
41       }
42
43       double avg = sum / students.size();
44       System.out.println("Average grade: " + avg);
45    }
46 }
```

Notice how a `HashMap` called `students` is declared on line 11:

```
   HashMap<String, Double> students = new HashMap<>();
11
```

Here, `<String, Double>` defines the data types for this map's `<key, value>` pairs. Like the `ArrayList`, when we call the `HashMap` constructor on the right side of the assignment, we don't need to specify type.

We can add a new item with a `.put()` method, specifying both key and value:

```
   students.put(newStudent, newGrade);
26
```

And while we don't do so in this example, we may also access `HashMap` elements using the `get` method. If we had a key/value pair of `"jesse"/4.0` in the `students` map, we could access the grade with:

```
Double jesseGrade = students.get("jesse");
```

Variables may be used to access elements:

```
  String name = "jesse";
1 Double jesseGrade = students.get(name);
2
```

Looping through a map is slightly more complex than it is for ordered lists. Let's look at the `for-each` loop from this example:

```
   for (Map.Entry<String, Double> student : students.entrySet()) {
38    System.out.println(student.getKey() + " (" + student.getValue() + ")");
39    sum += student.getValue();
40 }
41
```

The iterator variable, `student`, is of type `Map.Entry<String, Double>`. The class `Map.Entry` is specifically constructed to be used in this fashion, to represent key/value pairs within HashMaps. Each `Map.Entry` object has a `getKey` method and a `getValue` method, which represent (surprisingly enough!), the key and value of the map item.

If you only need to access the key of each item in a map, you can construct a simpler loop:

```
  for (String student : students.keySet()) {
1    System.out.println(student);
2 }
3
```

A similar structure applies if you only need the values, using `students.values()`:

```
  for (double grade : students.values()) {
1    System.out.println(grade);
2 }
3
```

3.6.1. HashMap Methods

Let's collect some `HashMap` methods as we have for `ArrayList`. As we said about `ArrayLists`, this is by no means a comprehensive list. For full details on all properties and methods available, see the reference section below for official documentation on the `HashMap` class.

For the purposes of this table, we'll create a map to hold our solar system's planets and the number of moons associated with each.

```
                HashMap<String, Integer> students = new HashMap<>();
1               moons.put("Mercury", 0);
2               moons.put("Venus", 0);
3               moons.put("Earth", 1);
4               moons.put("Mars", 2);
5               moons.put("Jupiter", 79);
6               moons.put("Saturn", 82);
7               moons.put("Uranus", 27);
8               moons.put("Neptune", 14);
9
```

Java Syntax	Description	Example
`size()`	Returns the number of items in the map, as an `int`.	`moons.size()` returns `8`
`keySet()`	Returns a collection containing all keys in the map. This collection may be used in a `for-each` loop just as lists are, but the map *may not be modified* within such a loop.	`moons.keySet()` returns `["Earth", "Mars", "Neptune", "Jupiter", "Saturn", "Venus", "Uranus", "Mercury"]`
`values()`	Returns a collection containing all values in the map. This collection may be used in a `for-each` loop just as lists are.	`moons.values()` returns `[1, 2, 14, 79, 82, 0, 27, 0]`
`put()`	Add a key/value pair to a map.	`moons.put("Pluto", 5)` adds `"Pluto": 5` to the `moons`
`containsKey()`	Returns a boolean indicating whether or not the map contains a given key.	`moons.containsKey("Earth")` returns `true`
`containsValue()`	Returns a boolean indicating whether or not the map contains a given value.	`moons.containsValue(79)` returns `true`

We have only brushed the surface of how arrays, `ArrayLists`, and maps work. We leave it to you to refer to the official documentation linked below for more details. You'll certainly be using `ArrayLists` and maps in more ways than those covered in this lesson, but with the knowledge you have now, you should be able to use Java collections and learn new uses as you go.

3.6.2. References

- HashMap Class (docs.oracle.com)

3.6.3. Check Your Understanding

Question

Given our `HashMap`,

```
  moons = {
1    "Mercury" = 0,
2    "Venus" = 0,
3    "Earth" = 1,
4    "Mars" = 2,
5    "Jupiter" = 79,
6    "Saturn" = 82,
7    "Uranus" = 27,
8    "Neptune" = 14
9 }
10
```

What is the method to return the key names?

1. `Map.keys(moons);`
2. `moons.keys();`
3. `moons.keySet(moons);`
4. `moons.keySet();`

Question

Given our `HashMap`,

```
  moons = {
1    "Mercury" = 0,
2    "Venus" = 0,
3    "Earth" = 1,
4    "Mars" = 2,
5    "Jupiter" = 79,
6    "Saturn" = 82,
7    "Uranus" = 27,
8    "Neptune" = 14
9 }
10
```

What will `moons.get("Mars");` return?

1. `3`
2. `{Mars: 3}`
3. `3.0`
4. `"Mars"`

3.7. Exercises: Control Flow and Collections

Work on these exercises in the IntelliJ `java-web-dev-exercises` project, creating a new class for each item. You may call these classes whatever you like, but remember to use the proper Java naming conventions.

3.7.1. Array Practice

1. Create and initialize an array with the following values in a single line: `1, 1, 2, 3, 5, 8`.
2. Loop through the array and print out each value, then modify the loop to only print the odd numbers.
3. For this exercise, use the string `I would not, could not, in a box. I would not, could not with a fox. I will not eat them in a house. I will not eat them with a mouse.` Use the `split` method to divide the string at each space and store the individual words in an array. If you need to review the method syntax, look back at the String methods table.
4. Print the array of words to verify that your code works. The syntax is:
5. `System.out.println(Arrays.toString(arrayName));`
6. Repeat steps 3 and 4, but change the delimiter to split the string into separate sentences.

Note

Some characters, like a period `"."`, have special meanings when used with the `split` method. They cannot be used as-is for the deliminator.

To use these characters as the deliminator, we must *escape* their special meanings. Instead of `.split(".")`, we need to use `.split("\\.")`.

3.7.2. ArrayList Practice

1. Write a static method to find the sum of all the even numbers in an ArrayList. Within `main`, create a list with at least 10 integers and call your method on the list.
2. Write a static method to print out each word in a list that has exactly 5 letters.
3. Modify your code to prompt the user to enter the word length for the search.
4. BONUS: Instead of creating our own list of words, what if we want to use the string from the *Array Practice* section? Search "Java convert String to ArrayList" online to see how to split a string into the more flexible `ArrayList` collection.

3.7.3. HashMap Practice

Make a program similar to `GradebookHashMap` that does the following:

1. It takes in student names and ID numbers (as integers) instead of names and grades.
2. The keys should be the IDs and the values should be the names.
3. Modify the roster printing code accordingly.

3.8. Studio: Counting Characters

In this studio, you will write a program to count the number of times each character occurs in a string and then print the results to the console.

Feel free to prompt the user for a string. However, for the sake of simplicity, you might want to start by hard-coding some text and storing it in a variable. For your convenience, here is a quote from the movie *Hidden Figures*:

If the product of two terms is zero then common sense says at least one of the two terms has to be zero to start with. So if you move all the terms over to one side, you can put the quadratics into a form that can be factored allowing that side of the equation to equal zero. Once you've done that, it's pretty straightforward from there.

Tip

Remember, you can turn a `String` object into an array of characters using:

```
char[] charactersInString = myString.toCharArray();
```

3.8.1. Some Items to Ponder Before Starting

1. There are multiple ways to approach this task, but one way involves the following steps:
    1. Loop through the string one character at a time,
    2. Store and/or update the count for a given character using an appropriate data structure.
    3. Loop through the data structure to print the results (one character and its count per line).
2. Which type of data structure (`ArrayList`, `HashMap`, or `Array`) should you use to store character counts? Any can be made to work, but there is a BEST choice.
3. You'll need to *initialize* the counts for the characters in some fashion. It's probably better to do this as you go through the string instead of doing so before you loop through it. (*WHY?*)
4. If you need to review how to create a new class, revisit the instructions in Studio: Area of a Circle.
5. Don't forget to check out the *Bonus Missions* below.

3.8.2. Sample Output

For the example string above, your output should look something like:

```
I: 1
O: 1
S: 1
': 2
 : 66
a: 20
b: 2
c: 7
d: 7
e: 32
f: 9
g: 2
h: 13
i: 11
l: 6
,: 2
m: 8
n: 12
.: 3
```

```
o: 31
p: 3
q: 3
r: 18
s: 16
t: 38
u: 8
v: 3
w: 5
y: 5
z: 3
```

3.8.3. Bonus Missions

Try these modifications on your code:

1. Prompt the user to enter the string in the command line.
2. Make the character counts case-insensitive.
3. Exclude non-alphabetic characters.

3.8.3.1. Super Bonus

Read the string in from a file.

Note

This is a hard one. We won't talk about reading from files in Java in this course, so be ready for a tough challenge if you accept this mission.

4.1. Classes for Java

In previous programming studies, we have come across **classes** and **objects**. Classes and objects in Java are similar to classes and objects in other languages.

4.1.1. A Minimal Class and Object

Classes may contain **fields** and **methods**. Fields contain our data for the class and methods define actions a class can take. We say that fields and methods are **members** of a class.

Example

Let's create a class called HelloWorld with one field, message, and one method, sayHello(). message will be a string and have a value of "Hello World". sayHello() will not return a specific value and instead print out the value of message.

```
 public class HelloWorld {
1
2    String message = "Hello World";
3
4    void sayHello() {
5          System.out.println(message);
6    }
7
8 }
9
```

The only field in the HelloWorld class is the string message, while the only method is sayHello, which prints the value of the message field and doesn't return anything.

Note

There is no main method, which is required to run a Java program. Without it, we have to do some additional work to get our program to run!

To execute sayHello, we'll need to create an **instance** of the class HelloWorld. We refer to an object created from a particular class as an instance of that class.

Here's how this might look with our HelloWorld class:

Example

```
 public class HelloWorldRunner {
1
2    public static void main(String[] args) {
3          HelloWorld hello = new HelloWorld();
4          hello.sayHello();
5    }
6 }
7
```

In order to call the sayHello method of HelloWorld, we must first have an instance of HelloWorld, which we create using the syntax new HelloWorld(). As with built-in classes, classes that we create define their own types. So the object hello is a variable of type HelloWorld.

We introduced this HelloWorld class as a means of illustrating the simplest representation of some basic concepts in Java. The goal of the next few lessons is to build up the machinery to create a wide variety of interesting classes that can be used to create complex programs and elegantly solve difficult problems.

4.1.2. The this Keyword

In HelloWorld above, we could have written sayHello this way, with the same net effect:

```
public void sayHello() {
    System.out.println(this.message);
}
```

In this context, inside of the class, we can refer to fields (and methods) that belong to the class using the special object, this. Whenever you use this, it *always* refers to the object that the given code is currently within. In other words, this will always be an instance of the given class. Since it is not legal to create code outside of a class in Java, this nearly always makes sense to use (there's one exception, that we'll encounter soon).

You are allowed to create local variables (variables declared within a method) with the same name as a field of the given class. In this case, in order to refer to the field, we *must* use this.

Example

Let's look at how this works with our HelloWorld class:

```
  public class HelloWorld {
1
2    String message = "Hello World";
3
4    public void sayHello() {
5
6       String message = "Goodbye World";
7
8       // The line below prints "Goodbye World"
9       System.out.println(message);
10
11      // The line below prints "Hello World"
12      System.out.println(this.message);
13   }
14 }
15
```

Warning

When a local variable has the same name as a field, we say that the local variable **shadows** the field. Errors caused by shadowing can be tricky to spot, so it's best to avoid doing this in your code.

Note

If you want to learn more about this subject, check out the Oracle Documentation on using the this keyword.

4.1.3. Check Your Understanding

Question

The following code block contains several bugs. Mark all of the lines that contain a bug in the code.

```
 public class Greeting {
1
2    String name = "Jess"
3
4    public void sayHello() {
5       System.out.println("Hello " + here.name+"!");
6
7 }
8
```

1. line 7
2. line 3
3. line 6
4. line 1

4.2. Modifiers in Java

4.2.1. Access Modifiers

For fields in classes, the **access level** determines who can get or set the value of the field. For methods, the **access level** determines who can call the method. The access level of a class member is determined by an **access modifier**.

We've encountered access modifiers so far in our code. In our examples, you frequently see the keyword, `public`. `public` makes the field or method to be accessible by anyone working with our code. Another common access modifier is `private`, which restricts access to fields or methods so they can only be used within the class. Two additional access modifiers are available in Java, though they are used much less often than `public` and `private`.

Example

Let's take a look at our `HelloWorld` class again from the first section of this chapter.

```
 public class HelloWorld {
1
2    String message = "Hello World";
3
4    void sayHello() {
5          System.out.println(message);
6    }
7
8 }
9
```

In this HelloWorld class, we omit the access modifier, therefore implicitly giving the message field and the sayHello method **default access**.

We should avoid giving everything default access when creating classes in Java and instead think carefully about what level of access each field and method should have.

The table below details whether or not information can be accessed at different levels based on the access modifier. For example, a field with the `private` access modifier can be accessed within the class, but cannot be accessed outside the class at the world-level. In Java, world-level is the level of the whole application and contains all of the packages and classes. While we will discuss later how to decide which access modifier to use for different scenarios, you should save this table now as reference for those conversations.

Is information accessible at certain levels with certain access modifiers?

Modifier	Class	Package	World
`public`	Yes	Yes	Yes
`protected`	Yes	Yes	No
(no modifier, aka *package-private* or *default*)	Yes	Yes	No
`private`	Yes	No	No

Note

If you would like to learn more about access modifiers, you should check out the Oracle documentation on the subject.

Let's take a look at our `HelloWorld` class again and add some access modifiers.

Example

```
  public class HelloWorld {
1
2    private String message = "Hello World";
3
4    public void sayHello() {
5       System.out.println(message);
6    }
7
8 }
9
```

Since `message` only needs to be used by `sayHello`, we declare it to be `private`. Since we want `sayHello` to be usable by anybody else, we declare it to be `public`.

Note

In Java, you should always use the most restrictive access modifier possible. Minimizing access to class members allows code to be refactored more easily in the future, and hides details of how you implement your classes from others.

Note

If you want to learn more about controlling access to members of a class.

4.2.2. Check Your Understanding

Question

For this question, refer to the code block below.

```
public class Greeting {

   String name = "Jess";

   public void sayHello() {
      System.out.println("Hello " + this.name + "!");
   }
}
```

What access modifier would you give `name`?

1. no access modifier
2. `public`
3. `private`
4. `protected`

4.3. Encapsulation

Our discussion of classes and objects is integral to us using **object-oriented programming**. Object-oriented programming stands on four pillars: abstraction, encapsulation, inheritance, and polymorphism.

4.3.1. Encapsulation

Encapsulation is the bundling of related data and behaviors that operate on that data, usually with restricted access to internal, non-public data and behaviors. In object-oriented programming, classes and objects allow us to encapsulate, or isolate, data and behavior to only the parts of our program to which they are relevant. Restricting access allows us to expose only that data and behavior that we want others to be able to use.

Let's take a look at this by developing a new class called `Student`.

4.3.2. `Student` Class

4.3.2.1. Fields

We previously defined a **field** as a variable, or piece of data, that belongs to a class. For our `Student` class, let's think about the data that is typically associated with a student (in the sense of a high school or college student). There are a lot of possibilities, but here are the most important:

1. Name
2. Student ID
3. Number of credits
4. GPA

In order to declare these fields within our class, we'll need to determine the best data type for each. A field may be of any primitive or object type. In this case, the following types will work best:

1. Name: `String`
2. Student ID: `int`
3. Number of credits: `int`
4. GPA: `double`

Let's put these inside of a class. While they may be declared anywhere within a class, fields should always be declared at the top of the class. When we're ready to add methods, we'll add them below the fields.

```
  public class Student {
1
2     String name;
3     int studentId;
4     int numberOfCredits;
5     double gpa;
6
7 }
8
```

Like variables within a method, fields may be initialized when they are declared. For example, we could provide default values for `numberOfCredits` and `gpa` (default values for `name` and `studentId` don't make sense since they should be different for each student).

```
int numberOfCredits = 0;
double gpa = 0.0;
```

Fields are also referred to as **instance variables**, since they belong to an instance of a class.

4.3.2.2. Getters and Setters

As declared, our four fields are package-private, which means that they can be read or changed by any code within the same package. As a rule-of-thumb, *fields should always be private unless you have a very, very, very good reason to not make them so.* So, let's make our fields private.

```
  public class Student {
1
2     private String name;
3     private int studentId;
4     private int numberOfCredits = 0;
5     private double gpa = 0.0;
6
7 }
8
```

In order to provide access to private fields, **getter and setter methods** are used. Getters and setters do what you might guess: get and set a given field. If we make the getter and/or setter method for a given property public, then others will be able to access or modify the field in that way.

Here is a getter/setter pair for `name` (you can imagine how the others would be written).

```
  public String getName() {
1     return name;
2 }
3
4 public void setName(String aName) {
5     name = aName;
6 }
7
```

Note

Prefixing a parameter that is intended to set an instance variable with `a` is a relatively common convention, and one that we'll adopt to avoid shadowing and having to use `this` in our setters. You can think of the `a` as denoting the "argument" version of the variable.

An astute question to ask at this point would be, "Why make the fields private if you're just going to allow people to get and set them anyway!?" Great question. There are lots of reasons to use getters and setters to control access. Here are just a few:

1. Sometimes you'll want to implement behavior that happens every time a field is accessed (get) or changed (set). Even if you can't think of such a reason when writing your class, you might later have the need to add such behavior. If you don't use getters and setters, you'll have to do a lot more refactoring if you ever decide to add such behaviors.
2. You can perform validation within a setter. For example, we might want to ensure that a student's name contains only certain characters, or that their student ID is positive.
3. You can use different access modifiers on getters and setters for the same field, based on desired usage. For example, you might want to allow anyone to be able to read the value of a field, but only classes within the same package to modify it. You could do this with a public getter and a package-private setter, but not as a field without getters and setters, which could only be public to everyone or package-private to everyone.

As an example of reason 2, let's take a short detour to look at a `Temperature` class. A valid temperature can only be so low ("absolute zero"), so we wouldn't want to allow somebody to set an invalid value. In `setFahrenheit` we print out if an invalid value is provided.

```
  public class Temperature {
 1
 2     private double fahrenheit;
 3
 4     public double getFahrenheit() {
 5         return fahrenheit;
 6     }
 7
 8     public void setFahrenheit(double aFahrenheit) {
 9
10         double absoluteZeroFahrenheit = -459.67;
11
12         if (aFahrenheit < absoluteZeroFahrenheit) {
13             System.out.println("Value is below absolute zero");
14         }
15
16         fahrenheit = aFahrenheit;
17     }
18 }
19
```

Note

When writing getters and setters, the convention for a field named `field` is to name them `getField` and `setField`. This is more than just a convention, as some libraries you use will *expect* names to be of this format, and won't work as desired if you don't follow the convention.

4.3.2.3. Properties

A **property** in Java is a characteristic that users can set. Our `Student` class had properties `name`, `studentId`, `numberOfCredits`, and `gpa`, while our `Temperature` class had only one property, `fahrenheit`.

Most often, properties will be fields that have public setters, though they need not have a corresponding field. Let's look at an example of a property that doesn't directly correspond to a field. If we wanted to add a `celsius` property to the `Temperature` class above, we might do it as follows:

```
  public double getCelsius() {
1     return (fahrenheit - 32) * 5.0 / 9.0;
2 }
3
4 public void setCelsius(double celsius) {
5     double fahrenheit = celsius * 9.0 / 5.0 + 32;
6     setFahrenheit(fahrenheit);
7 }
8
```

Since there's a link between `fahrenheit` and `celsius`, we want to make sure that when one is updated, so is the other. In this case, we only store one field value (`fahrenheit`) and make the appropriate calculation when getting or setting the `celsius` property.

Note

There are slight variations among Java developers when it comes to colloquial usage of the term *property*. People will sometimes define the term in a slightly more specific or narrow way, to mean a private field with public getters and setters.

Using properties, getters/setters, and fields, we can *encapsulate* the information we need in our student class.

4.3.3. Check Your Understanding

Question

What is a method that is used to give a private field a value?

1. getter
2. method
3. property
4. setter

4.4. Constructors

We'll often want to initialize, or set the initial value of, some of our fields when creating a new object from a class. **Constructors** allow us to do so.

In Java, constructors have the same name as the class and are most often declared public (though they can have any other valid access modifier). They are declared *without a return type*. Any function that is named the same as the class and has no return type is a constructor.

Here is an example of a constructor definition within the `HelloWorld` class:

```
  public class HelloWorld {
1
2     private String message = "Hello World";
3
4     public HelloWorld(String message) {
5         this.message = message;
6     }
7
8     public String getMessage() {
9         return message;
10    }
11
12    public void setMessage(String aMessage) {
13        message = aMessage;
14    }
15
16 }
17
```

This constructor allows us to create `HelloWorld` objects with custom messages. The assignment `this.message = message` assigns the value passed into the constructor to the field `message`. Here's how we might use it:

```
HelloWorld goodbye = new HelloWorld("Goodbye World");
System.out.println(goodbye.getMessage()); // prints "Goodbye World"
```

It's not required that every class have a constructor. If you don't provide one, the Java compiler will generate an empty constructor for you, known as a **default constructor**. For example, when we left out a constructor in our `HelloWorld` class above, the compiler created the following constructor for us:

```
public HelloWorld() {}
```

While this can be convenient, you almost always want to provide a constructor to properly initialize your objects.

4.4.1. Overloading Constructors

We can provide multiple constructors for a given class in order to allow for different initialization scenarios. This is known as **constructor overloading**. When providing multiple constructors, we must ensure that each has a different collection of arguments, as determined by the number, order, and type of the constructor arguments.

Let’s expand upon our `Student` class.

Example

```
  public class Student {
 1
 2    private String name;
 3    private final int studentId;
 4    private int numberOfCredits;
 5    private double gpa;
 6
 7    public Student(String name, int studentId,
 8              int numberOfCredits, double gpa) {
 9       this.name = name;
10       this.studentId = studentId;
11       this.numberOfCredits = numberOfCredits;
12       this.gpa = gpa;
13    }
14
15    public Student(String name, int studentId) {
16       this.name = name;
17       this.studentId = studentId;
18       this.numberOfCredits = 0;
19       this.gpa = 0.0;
20    }
21
22    /* getters and setters omitted */
23
24 }
25
```

The first constructor allows for the creation of `Student` objects where the code creating the object provides initial values for each of the fields. The second allows for the creation of `Student` objects with only `name` and `studentId`. The first constructor would be most useful for creating a transfer student, where credits and a GPA might already be non-zero. However, for all new students, it would be safe to initialize `numberOfCredits` and `gpa` to be 0.

A better way to write the above constructors would be this:

Example

```
  public class Student {
1
2     private String name;
3     private final int studentId;
4     private int numberOfCredits;
5     private double gpa;
6
7     public Student(String name, int studentId,
8               int numberOfCredits, double gpa) {
9        this.name = name;
10       this.studentId = studentId;
11       this.numberOfCredits = numberOfCredits;
12       this.gpa = gpa;
13    }
14
15    public Student(String name, int studentId) {
16       this(name, studentId, 0, 0);
17    }
18
19    /* getters and setters omitted */
20
21 }
22
```

In the example above on line 15, we use `this()` to invoke another constructor within the same class. In this case, the second constructor calls the first with the default values for `numberOfCredits` and `gpa`. If you use this syntax, the call to `this()` must be the first line in the constructor. This is a good practice not only because it makes your code shorter, but also because it allows any initialization behavior that may be carried out beyond just initializing variables to be contained in a smaller number of constructors. In other words, constructors can share initialization code. Notice from this example that a constructor doesn't need to require an initial value for each field as an argument.

When defining constructors, think about:

1. Which fields must be initialized for your class to work properly? Be sure you initialize every such field.
2. Which fields should be initialized by the user creating an object, and which should be initialized by the class itself?
3. What are the use-cases for your class that you should provide for?

4.4.2. Check Your Understanding

Question

A constructor is required for every class.

1. True
2. False

Question

Let's take a look at a class called `Dog`.

```
  public class Dog {
1    private String name;
2    private String breed;
3
4    public Dog(String name, String breed) {
5       this.name = name;
6       this.breed = breed;
7    }
8 }
9
```

What line of code would be appropriate for us to declare an instance of the `Dog` class called `myDog` and give it the name, "Bernie", and the breed, "Beagle"?

1. `Dog myDog = new Dog(Bernie,beagle);`
2. `Dog myDog = new Dog("Bernie","beagle");`
3. `Dog Bernie = new Dog("Bernie","beagle");`

4.5. Methods

4.5.1. Calling Methods on Objects

A **method** is a function that belongs to a class. In Java, all procedures must be part of a class. Let's revisit our `HelloWorld` class.

```
  public class HelloWorld {
1
2     private String message = "Hello World";
3
4     void sayHello() {
5         System.out.println(message);
6     }
7
8 }
9
```

There is one method in this class, `sayHello()`. In order to call this method, we must have an object created from the `HelloWorld` class template. In other words, we must have an instance of `HelloWorld`.

Here's how you call methods on an object.

```
HelloWorld hello = new HelloWorld();
hello.sayHello();
```

It is not possible to call `sayHello()` without having a `HelloWorld` object. This begins to make more sense when you note that the `message` field is used within `sayHello()`, and unless we are calling `sayHello()` on an instantiated object, there will be no `message` field available to print.

Note

As mentioned before, class members should have the most restrictive level of access possible. This goes for methods as well as fields. For example, if you create a utility method that should only be used within your class, then it should be `private`. You can think of `private` methods as those that are not useful *outside* of the class, but that can contribute internally to helping the class behave as desired or expected.

4.5.2. Instance Methods

So far, we've only looked at examples of methods that are relatively specialized: constructors, getters, and setters. Every class you create will have these methods. What will make your classes different from each other, and thus fulfill the purpose of creating each class, are the specific behaviors that are unique to your classes.

Let's say we want to add a method to get the grade level of a student to our `Student` class. This method is an **instance method** since it will belong to each `Student` object created, and will use the data of each such object.

```
  public class Student {
1
2     private static int nextStudentId = 1;
3     private String name;
4     private int studentId;
5     private int numberOfCredits;
6     private double gpa;
7
8     public Student(String name, int studentId, int numberOfCredits, double gpa) {
9         this.name = name;
10        this.studentId = studentId;
11        this.numberOfCredits = numberOfCredits;
12        this.gpa = gpa;
13    }
14
15    public Student(String name, int studentId) {
16        this(name, studentId, 0, 0);
17    }
18
19    public Student(String name) {
20        this(name, nextStudentId);
21        nextStudentId++;
22    }
23
24    public String studentInfo() {
25        return (this.name + " has a GPA of: " + this.gpa);
26    }
27
28    /* getters and setters omitted */
29
30 }
31
```

We will make use of instance methods more in the next chapter, however, we wanted to share more about them now in our first conversation about classes. Sometimes when we create a class, we will need that class to have more behaviors than using only constructors, setters, and getters can provide. When we do want to add additional behaviors to our classes, we can use instance methods!

4.5.3. Check Your Understanding

Question

Fill in the blanks with the appropriate terms.

A _____ gives a class property a field. A _____ gives a programmer access to the value of a private class property. A _____ creates a new instance of a class with values for the fields. A _____ is a method that belongs to each instance of a class.

4.6. Single Responsibility Principle

As we wrap up our whirlwind tour of classes, we want you think a bit about how to go about building good classes. Doing so is more of an art than a science, and it will take you lots of practice and time. However, there are a few rules that we've pointed out to help guide you. Here's one more.

From Wikipedia:

> The **single responsibility principle** states that every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class.

It isn't always clear what "responsibility over a single part of the functionality" means. However, it is often very clear what it doesn't mean. For example, we wouldn't think of adding functionality to the `Student` class that tracked all of the data for each of the student's classes, such as catalog number, instructor, and so on. Those are clearly different areas of responsibility. One way to interpret the single responsibility principle is to say that "classes should be small."

As you go forth and create classes, the main thing to keep in mind is that your skill and judgement in creating Java classes will improve over time. The best way to improve is to write lots of code, ask lots of questions, and continue learning!

If you are interested in learning more about Single Responsibility Principle, you can check out the entry on Wikipedia.

4.7. Exercises: Classes and Objects

Work on these exercises in the IntelliJ `java-web-dev-exercises` project.

1. Open up the file, `Student.java`, and add all of the necessary getters and setters. Think about the access level each field and method should have, and try reducing the access level of at least one setter to less than public.
2. Instantiate the `Student` class using yourself as the student. For the `numberOfCredits` give yourself `1` for this class and a GPA of `4.0` because you are a Java superstar!
3. In the `school` package, create a class `Course` with at least three fields. Before diving into IntelliJ, try using pen and paper to work through what these might be. At least one of your fields should be an `ArrayList` or `HashMap`, and you should use your `Student` class.
4. In the `school` package, create a class `Teacher` with four fields: `firstName`, `lastName`, `subject`, and `yearsTeaching`. Add getters and setters to the class and add the access level to each field and methd in the class. When adding your getters and setters, think about what we read about in the section on Encapsulation.
    1. What access modifier restricts access the most for what we need?
    2. If it is a field, could we restrict the access to `private` and use getters and setters?
    3. If we do set fields to `private`, what access level do we have to give our getters and setters?

4.8. Studio: Classes and Objects

Let's practice designing classes using the following scenario. You've been hired to create a web application for a local restaurant. They want to both display their current menu and edit it through an admin panel.

You're not going to build an actual application in this studio. Instead, you will focus on the *design* of a portion of this application. Object-oriented programming in Java requires intentional, up-front planning. While this may seem tedious, outlining your ideas before you code helps reduce the errors you need to fix later.

4.8.1. Design

You know you'll need to create classes within the web application to facilitate this behavior and represent the various components of the menu. After talking to the owner, you have these details:

1. The menu consists of several menu items
2. Each menu item has a price, description, and category (appetizer, main course, or dessert)
3. It should be possible to display whether a menu item is new or not
4. The app should know when the menu was last updated, so visitors can see that the restaurant is constantly changing and adding exciting new items

Starting with pen and paper (or your favorite notes application on your laptop), sketch out the design for two classes, `Menu` and `MenuItem`. List the fields that each should have, along with the data type and access level for each. Also consider what constructors the classes might need.

Note

For this studio, we are focusing on class design for these two classes. You do not need to be concerned with how the classes would be used in an application. At this stage, don't think about how the application will work or behave; you should focus on the way that data will be represented within these classes, and how they should relate to each other.

You may find it useful to use one or more of the classes provided by Java, such as Date.

4.8.2. Presenting Your Design

Once you have sketched out your fields and properties, pair with a classmate and take turns presenting your designs. Class design can be subjective, so it's important to properly think and talk through your choices before coding.

While your partner is presenting their design, ask questions about why they made the decisions they did. Consider other use cases that might come up, and see if their design fits with those.

4.8.3. Implementation

In IntelliJ, create a new project, `Restaurant Menu`. Within the project, create a new package named `restaurant`. Add the `Menu` and `MenuItem` classes and code the design that you created above. Be sure to add getters and setters as appropriate.

4.8.4. Submitting Your Work

Create a repository on your Github account and push up your project. Submit the link to your repository on Canvas.

5.1. Customizing Fields

In the previous chapter, we used *fields* to store data within a class, and we explored how to access and modify the values of those fields.

Now, we will explore several ways to configure fields based on their intended use.

5.1.1. Final Fields

A **final field** is one that cannot be changed once it is initialized. This means slightly different things for primitive and class types. We create final fields by declaring them with the final keyword.

We cannot change the value of a **final primitive field** (final int, final double, etc.) after it is initialized.

Similarly, we cannot assign a new object to a **final object field** (final String, final Double, final ClassName, etc.) after initialization. However, we can change the values within the object itself.

Here are some examples to illustrate. Each class would normally be in its own file, but we present them side-by-side for convenience. Additionally, we declare each field public to minimize the example code and more clearly demonstrate where compiler errors would occur.

Examples

```
public class FortyTwo {

   public int intValue = 42;

}

public class FinalFields {

   public final int intValue = 42;
   public final double doubleValue;
   public final FortyTwo objectValue = new FortyTwo();

   public static void main(String[] args) {

      FinalFields demo = new FinalFields();

      // This would result in a compiler error
      demo.intValue = 6;

      // This is allowed since we haven't initialized doubleValue yet
      demo.doubleValue = 42.0;

      // However, this would result in a compiler error
      demo.doubleValue = 6.0;

      // This would result in a compiler error, since we're trying to
      // give objectValue a different object value
      demo.objectValue = new FortyTwo();

      // However, this is allowed since we're changing a field
      // inside the final object, and not changing which object
      // objectValue refers to
      demo.objectValue.intValue = 6;
   }
}

```

Final fields help to prevent accidentally (or intentionally) changing the value of a field after it is initialized. As such, final fields may NOT have setters.

5.1.2. Static Fields

A **static field** is one that is *shared by all instances of the class*, and it is declared with the static keyword.

For example, in our Temperature class there is no reason for each Temperature object to hold its own copy of the double absoluteZeroFahrenheit. That value remains constant from class to class and object to object. Because of this, we make it a static field.

Previous examples used the static keyword with both fields and methods, but since this discussion is focused on data, let's focus on static fields for now.

```
  public class Temperature {
1
2    private double fahrenheit;
3
4    private static double absoluteZeroFahrenheit = -459.67;
5
6    public double getFahrenheit() {
7       return fahrenheit;
8    }
9
10   public void setFahrenheit(double aFahrenheit) {
11
12      if (aFahrenheit < absoluteZeroFahrenheit) {
13         throw new IllegalArgumentException("Value is below absolute zero");
14      }
15
16      fahrenheit = aFahrenheit;
17   }
18
19   /* rest of the class... */
20
21 }
22
```

There are multiple ways to refer to a static field.

Examples

Within a class:

```
  // Use a static field the same way as a normal, non-static field
1 System.out.println("Absolute zero in F is: " + absoluteZeroFahrenheit);
2
3 // We can also be more explicit
4 System.out.println("Absolute zero in F is: " + this.absoluteZeroFahrenheit);
5
```

Outside of a class:

```
   // If the static field is public, we can do this
 6 System.out.println("Absolute zero in F is: " + Temperature.absoluteZeroFahrenheit);
 7
 8 // Or if we have an object named "temp" of type Temperature
 9 System.out.println("Absolute zero in F is: " + temp.absoluteZeroFahrenheit);
10
```

When accessing a field from outside of its class, line 7 shows the preferred technique. The syntax makes it explicit that the field is static. Line 10 does not make this point clear.

Example

As another example, we might also provide a third constructor for our Student class that only requires the student's name. Theoretically, the studentId would (or could) be generated by the class itself.

```
  public class Student {
 1
 2    private static int nextStudentId = 1;
 3    private String name;
 4    private final int studentId;
 5    private int numberOfCredits;
 6    private double gpa;
 7
 8    public Student(String name, int studentId,
 9              int numberOfCredits, double gpa) {
10       this.name = name;
11       this.studentId = studentId;
12       this.numberOfCredits = numberOfCredits;
13       this.gpa = gpa;
14    }
15
16    public Student(String name, int studentId) {
17       this(name, studentId, 0, 0);
18    }
19
20    public Student(String name) {
21       this(name, nextStudentId);
22       nextStudentId++;
23    }
24
25    /* getters and setters omitted */
26
27 }
28
```

In line 3, we add a static integer field that will keep track of the next student ID to be assigned to a student. Then, our new constructor (line 21) takes only a name as a parameter and assigns the student the next available ID. This works because static fields are shared across all objects created from the Student class, so it functions as a counter of sorts for the number of Student objects created.

5.1.3. Constants

Unlike some other languages, Java doesn't have a special keyword to declare a constant, or unchanging, variable. However, we can achieve the same result using a combination of static and final.

```
  public class Constants {
1    public static final double PI = 3.14159;
2    public static final String FIRST_PRESIDENT = "George Washington";
3 }
4
```

Throughout the rest of this course, when we say *constant* we will mean a static final variable.

Three things to note from this example:

1. We use a different naming convention for constants than for other variables. Constants should be in ALL CAPS, with an underscore to separate words.
2. There is no strong reason to make constants private, since restricting access would force us to re-declare the same values in different classes. We'll generally make our constants public.
3. We must declare and initialize a constant at the same time. If we do not declare and initialize the constant in the same statement, we cannot assign it a value later. The constant's value remains empty.

A good use of a constant can be seen in our Temperature class. Since absolute zero will never change, we can ensure that nobody ever alters it (intentionally or by mistake) by adding final to make it a constant.

```
  public class Temperature {
1
2    private double fahrenheit;
3
4    public static final double ABSOLUTE_ZERO_FAHRENHEIT = -459.67;
5
6    /* rest of the class... */
7
8 }
9
```

5.1.4. References

1. Declaring Member Variables (docs.oracle.com)
2. Initializing Fields (docs.oracle.com)
3. Constructors (docs.oracle.com)

5.1.5. Check Your Understanding

Question

Assume that we define a Pet class that uses the fields name, age, mass, and species.

Assuming you do not give your pet away, which of these fields should be declared final? (There may be more than one).

1. name
2. age
3. mass
4. species

Should any of the fields be declared static?

1. Yes

2. No

Question

Assume we define several fields in a Circle class. Which of the following is the BEST choice to be declared static?

1. radius
2. area
3. pi
4. circumference

Question

Which of the following is the BEST syntax for defining a variable to hold the (constant) speed of light in a vacuum?

1. `public static final int SPEED_OF_LIGHT = 299792458;`
2. `private static final int SPEED_OF_LIGHT = 299792458;`
3. `public static final int SPEED_OF_LIGHT;`
4. `private static final int SPEED_OF_LIGHT;`

5.2. Instance and Static Methods

We explored configuring *data* within classes with fields and properties. Now let's turn our attention back to class *behavior* (methods).

5.2.1. Quick Method Review

In the last chapter, we learned that:

1. A *method* belongs to a class and performs an action.
2. Methods cannot stand on their own—they must be part of a class.
3. To call a method on an object, use dot notation:

```
4. objectName.methodName(arguments);
```

5. Access modifiers apply to methods:
    1. `private` methods as those that are NOT useful outside of the class but contribute internally to helping the class behave as desired or expected.
    2. `public` methods contain code that other classes need to use when they implement the class containing those methods. Make methods `public` only when you expect other classes to use them, and when you are committed to maintaining those methods for other programs.

Let's take a closer look at two different types of methods—both of which we have used in earlier examples.

5.2.2. Instance Methods

As we learned in the last chapter, *instance methods* define the behaviors that are *unique* or *specialized* to each class. Every object created from a class will carry a copy of these methods.

Instance methods depend on the data stored in an individual object. If two objects call the same method, the results will vary when the objects contain different data.

Let's add a couple more instance methods to our `Student` class.

What are the behaviors that our `Student` class should have? To start, it makes sense that when a student takes a class and earns a grade, their data should be updated accordingly. Additionally, it would be nice to easily identify the grade level of a student—freshman, sophomore, junior, or senior.

The framework for these new methods is shown in the `Student` class below, but each method is missing some code. Filling in that code is left for you to do in the chapter exercises.

```
  public class Student {
1
2     private static int nextStudentId = 1;
3     private String name;
4     private int studentId;
5     private int numberOfCredits;
6     private double gpa;
7
8     public Student(String name, int studentId,
9           int numberOfCredits, double gpa) {
10       this.name = name;
11       this.studentId = studentId;
12       this.numberOfCredits = numberOfCredits;
13       this.gpa = gpa;
14    }
15
16    public Student(String name, int studentId) {
17       this(name, studentId, 0, 0);
18    }
19
20    public Student(String name) {
21       this(name, nextStudentId);
22       nextStudentId++;
23    }
24
25    public String studentInfo() {
26       return (this.name + " has a GPA of: " + this.gpa);
27    }
28
29    public void addGrade(int courseCredits, double grade) {
30       // Update the appropriate fields: numberOfCredits, gpa
31    }
32
33    public String getGradeLevel() {
34       // Determine the grade level of the student based on numberOfCredits
35    }
36
37    /* getters and setters omitted */
38
39 }
40
```

Tip

When creating your classes, think about the behaviors that you want to make available, as well as the access level of those methods.

5.2.3. Static Methods

We've already used static methods quite a bit in this course, all the way back to our first Java method:

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

Now let's examine them in the context of what we've recently learned about classes.

Just like static fields, **static methods** belong to the class as a whole, and not to any of the specific instances of the class. Thus, they are sometimes also called **class methods**. A static method is essentially the opposite of an instance method, since the two cases are mutually exclusive. *Instance methods* rely on each object's specific data, while *static methods* must NOT rely on data from a specific object.

We call a static method by preceding it with the class name and using dot-notation. Here's an example that we looked at previously.

Examples

`HelloMethods.java`

```
 public class HelloMethods {
1
2    public static void main(String[] args) {
3       String message = Message.getMessage("fr");
4       System.out.println(message);
5    }
6
7 }
8
```

`Message.java`

```
  public class Message {
 1
 2    public static String getMessage(String lang) {
 3
 4       if (lang.equals("sp")) {
 5             return "¡Hola, Mundo!";
 6       } else if (lang.equals("fr")) {
 7             return "Bonjour, le monde!";
 8       } else {
 9             return "Hello, World!";
10       }
11    }
12 }
13
```

The call occurs in line 4: `Message.getMessage("fr")`. We call the static `getMessage` without needing an instance of the `Message` class. Instead, we use the name of the class itself.

Warning

It is technically allowed to call a static method using an instance of a class: `myObject.someStaticMethod()`. However, best practice recommends using the class name instead: `ClassName.someStaticMethod()`. This makes it clear to other coders that you are calling a static method.

A method should be static when it does NOT refer to any instance fields of the containing class (it *may* refer to static fields, however). These methods tend to be utility-like (e.g. carrying out a calculation, or using or fetching some external resource).

5.2.4. Accessing Static vs. Instance Fields

One common error new Java coders encounter reads something like *non-static variable cannot be referenced from a static context*. This occurs when a *static method* tries to call an *instance variable*.

Why can't we do this? Static methods can be called from anywhere (depending on their access modifier), and they do NOT require us to create an object for a particular class. However, these methods must be independent of any values unique to a particular object.

For example, if we have a `Circle` class, then we can define and call a static `area` method without creating a new object: `Circle.area(radius)`. Since the area of a circle is just, `PI*radius*radius`, we can pass in the argument when we call the method. The method does not depend on any value stored within a specific `Circle` object.

Now let's assume we define a `Car` class with an instance variable for `color`. The value of this field will NOT be the same for every `Car` object we create. Thus, trying to call a static method like `Car.printColor()` results in an error. Since there is no single value for `color` that applies to every object, trying to access it from outside of the class does not work. To print the color of a `Car` object, we must call the method on that specific object: `myCar.printColor()`.

Instance fields can only be called by instance methods.

Note

While static methods cannot access instance variables, an instance method CAN access a static variable. Why?

5.2.5. References

1. Encapsulation (wikipedia.org)
2. Defining Methods (docs.oracle.com)
3. Passing Data to a Method or Constructor (docs.oracle.com)

5.2.6. Check Your Understanding

Question

Assume that we add two methods to a `Pet` class—`public String makeNoise()` and `public static void increaseAge()`. Which of the following statements is true?

1. The `makeNoise()` method can be accessed outside of the `Pet` class, while the `increaseAge()` method cannot.
2. Each `Pet` object carries a copy of the `makeNoise()` method but NOT a copy of the `increaseAge()` method.
3. The `increaseAge()` method can be accessed outside of the `Pet` class, while the `makeNoise()` method cannot.
4. Each `Pet` object carries a copy of the `increaseAge()` method but NOT a copy of the `makeNoise()` method.

Question

Explain why it IS possible for an instance method to access a static field.

5.3. Special Methods

Every class has a few special methods that belong to it, regardless of whether or not we define them. Exactly *how* every class obtains these methods will be explored in a future lesson. For now, let's look at two important examples of these methods.

5.3.1. `toString`

The `toString` method returns a string representation of a class. Calling `toString` on a class that you've written will result in something like this:

Example

```
  Student person = new Student("Violet");
1 System.out.println(person.toString());
2
```

Console Output

```
org.launchcode.java.demos.classes2.Student@61bbe9ba
```

Here, we called `toString` on a `Student` object. The default `toString` implementation is generally not very useful. Most of the time, you'll want to write your own `toString` method to *override* the default and provide better results.

Here's how we might do it for `Student` to produce a much more friendly message:

Example

```
  public String toString() {
1    return name + " (Credits: " + numberOfCredits + ", GPA: " + gpa + ")";
2 }
3
4 Student person = new Student("Violet");
5 System.out.println(person.toString());
6
```

Console Output

```
Violet (Credits: 0, GPA: 0.0)
```

In the example, we define the `toString` method to return a string that reports the class fields `name`, `numberOfCredits`, and `gpa` in a clear manner.

Note that `toString` is often implicitly called for you. For example, the output above could have been generated by the following code, which calls `toString` on `person` within `System.out.println`.

```
  Student person = new Student("Violet");
1 System.out.println(person);
2
```

5.3.2. `equals`

Suppose we have two objects of type `Student`, say `student1` and `student2`, and we want to determine if they are equal. If we try to compare them using `==`, we will likely get a result we did not expect. This is because `student1` and `student2` are reference variables, which means they hold a reference to, or the *address* of, the actual `Student` objects. `student1` and `student2` evaluate as equal only when they have the same memory address.

To state that again: `student1` and `student2` will be equal (`==`) only when they refer to, or point at, the exact same object. Consider the example below, which creates two `Student` objects:

Example

```
  Student student1 = new Student("Maria", 1234);
1 Student student2 = new Student("Maria", 1234);
2
3 System.out.println(student1.name + ", " + student1.id + ": " + student1);
4 System.out.println(student2.name + ", " + student2.id + ": " + student2);
5 System.out.println(student1 == student2);
6
```

Console Output

```
Maria, 1234: org.launchcode.java.demos.classes2.Student@3e3abc88
Maria, 1234: org.launchcode.java.demos.classes2.Student@6ce253f1
false
```

Even though the objects have the exact same keys and values, `student1` and `student2` point to different memory locations. Therefore, the `==` check returns `false`.

This is not usually how we want to compare objects. For example, we might want to consider two `Student` objects equal if they have the same name, email, or student ID.

The `equals()` method determines if one object is equal to another in this sense. We introduced the method when discussing strings, but it also applies to all other classes.

The code below shows how to use `equals()` to compare two students. Note that they have different names but the same student ID, indicating they are actually the same person.

```
  Student bono1 = new Student("Paul David Hewson", 4);
1 Student bono2 = new Student("Bono", 4);
2
3 if (bono1.equals(bono2)) {
4    System.out.println(bono1.getName() +
5       " is the same as " + bono2.getName());
6 }
7
```

If we don't provide our own `equals()` method, the default option only considers two objects equal if they are the *exact same object*, which means they point to the same memory address. This is identical to the behavior we see when using the == operator: `bono1 == bono2`.

In the example above we created two different `Student` objects, so the expression `bono1.equals(bono2)` evaluates to `false`. In order to compare two objects based on their *properties*, rather than their memory references, we need to define our own `equals()` method.

The difference between the comparison carried out by the default `equals()` method (and by the == operator), and how we would like to compare our classes, is the difference between *identity* and *equality*.

1. Two objects are *identical* if they both point to the same memory address. In essence, they are the same object. If `object1` and `object2` are identical, then changing one property value in `object1` also changes that value for `object2`.
2. Two objects are *equal* if the values they store are the same at the time of comparison. `student1` and `student2` point to different memory addresses, but their values are all the same. Thus, we can consider them equal, even though they are not identical.

The default `equals()` method and the == operator test for *identity*, whereas we want to test for *equality* instead. We can do so by **overriding** the `equals()` method. We will discuss overriding in more detail later, but for now just recognize that it involves defining different behavior for an existing method.

Two things can be considered *equal* even if they do NOT have all the same values. In the case of the `Student` class, we might specify that two `Student` objects are equal if they have the same ID numbers. We would then write a new method definition for `equals()` as follows:

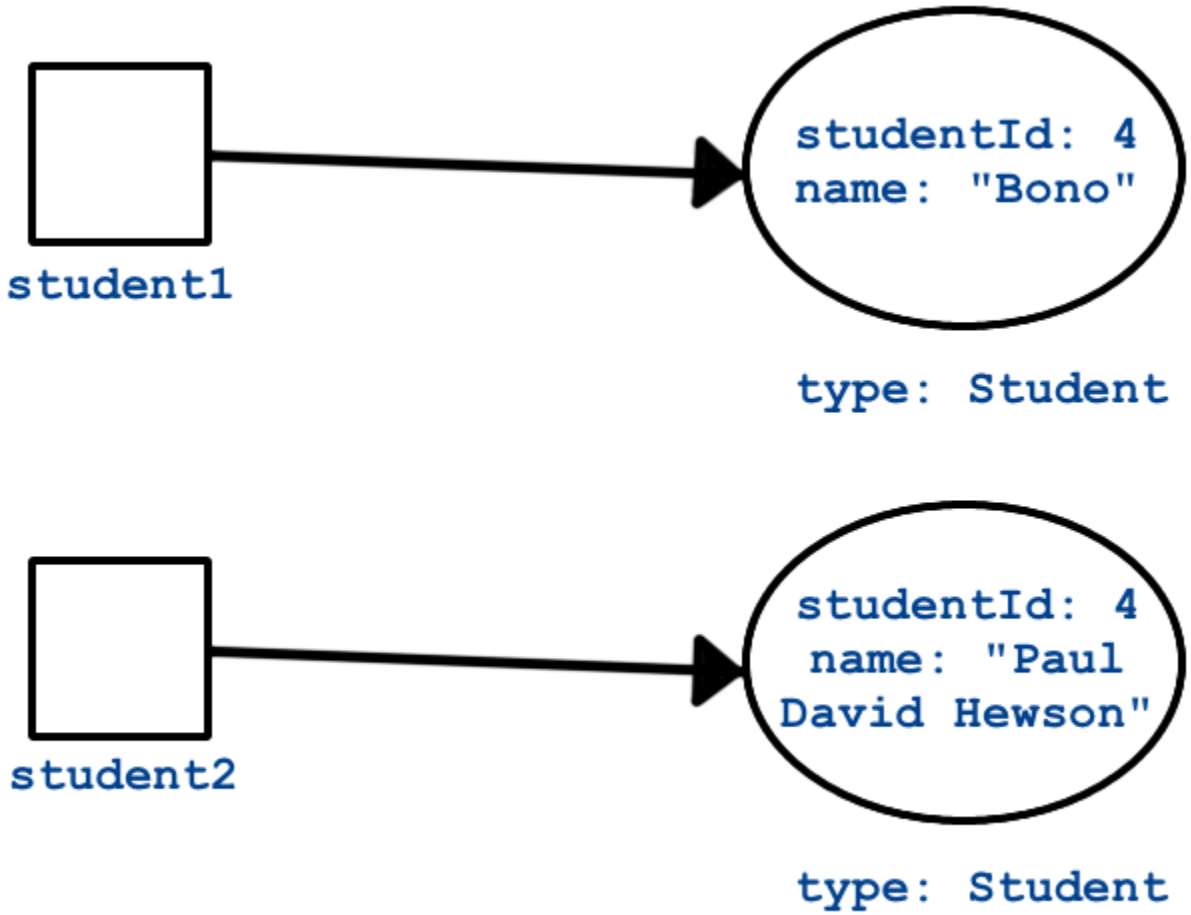
```
  public boolean equals(Object toBeCompared) {
1    Student theStudent = (Student) toBeCompared;
2    return theStudent.getStudentId() == getStudentId();
3 }
4
```

Now if we evaluate `bono1.equals(bono2)` we will get a result of true, since the student IDs match.

One catch of working with `equals()` is that its input parameter must be of type `Object`, even if we're working in a class like `Student`. The reason why will become more clear in the next lesson, when we introduce the `Object` class. For now, the practical implication is that we must convert, or **cast**, the input `toBeCompared` to be of type `Student` with the syntax `(Student) toBeCompared`. Then we compare the converted student's ID (`bono2.id`) to that of the current student (`bono1.id`).

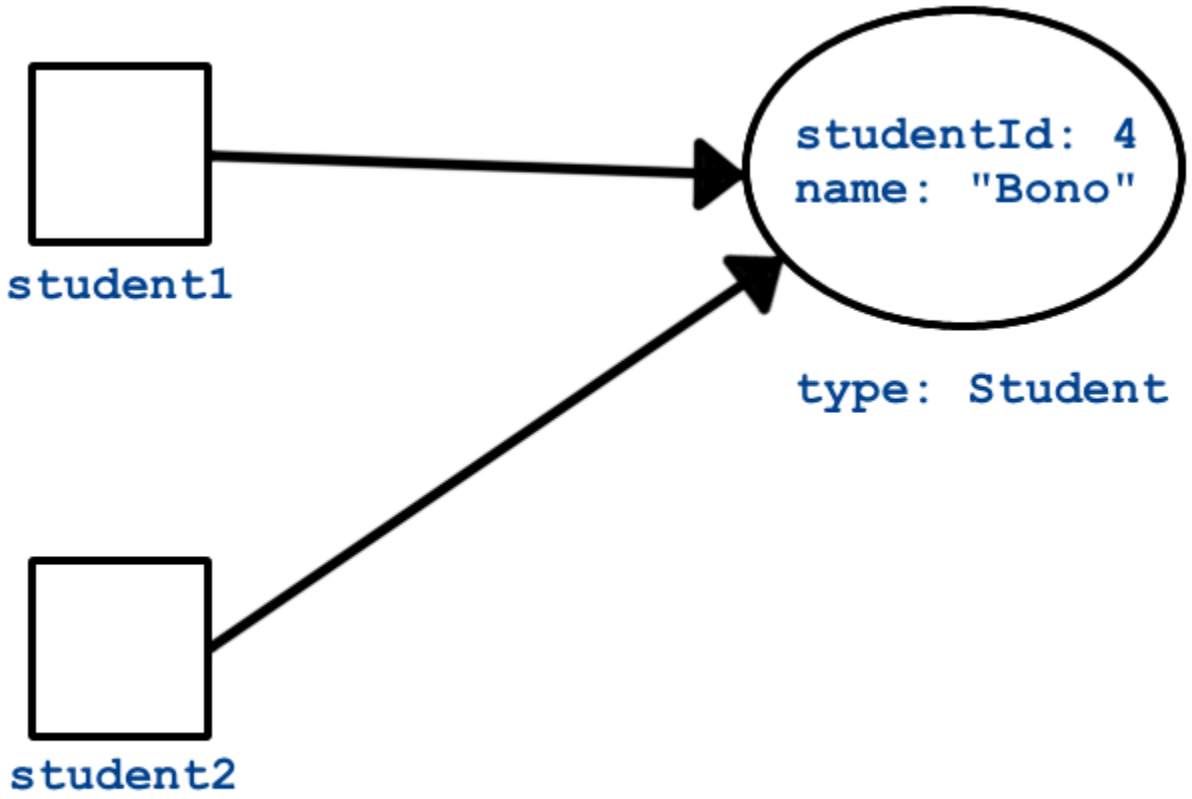
Here’s a visualization of the concepts of equality and identity:

Equality



When you test for equality, you look at two different objects and compare some aspect of them to each other.

Identity



When you test for identity, you look at two variables to see if they reference the exact same object.

5.3.2.1. Coding a New `equals` Method

You'll often want to implement `equals()` yourself. When you do, be sure you understand the best practices around how the method should behave. These are a little more involved compared to coding a new `toString` method.

In fact, the `equals()` method we defined above isn't very good by most Java programmers' standards. Let's improve it.

5.3.2.1.1. Problem #1

The method argument cannot be converted to a `Student` instance.

When we attempt to cast the argument `toBeCompared` to type `Student`, we'll get an exception if `toBeCompared` can't be properly converted. This happens if something other than a `Student` object gets passed into `equals()`. To prevent this from happening, we'll return `false` if `toBeCompared` was not created from the `Student` class. To check this, we use the `getClass` method, which is available to every object (similarly to `toString`).

```
  public boolean equals(Object toBeCompared) {
1
2    if (toBeCompared.getClass() != getClass()) {
3       return false;
4    }
5
6    Student theStudent = (Student) toBeCompared;
7    return theStudent.getStudentId() == getStudentId();
8 }
9
```

Lines 3 - 5 ensure that the two objects that we want to compare were created from the same class.

5.3.2.1.2. Problem #2

`toBeCompared` might be `null`.

If `toBeCompared` is `null` then `toBeCompared.getClass()` throws an exception. This is an easy issue to fix—just compare the object to `null`. If the comparison evaluates to `true`, then we know the object is null and `equals()` should return `false`.

```
   public boolean equals(Object toBeCompared) {
 1
 2    if (toBeCompared == null) {
 3       return false;
 4    }
 5
 6    if (toBeCompared.getClass() != getClass()) {
 7       return false;
 8    }
 9
10    Student theStudent = (Student) toBeCompared;
11    return theStudent.getStudentId() == getStudentId();
12 }
13
```

Line 3 checks `toBeCompared` for `null`, preventing an error in line 7. Line 7 checks the class of `toBeCompared`, preventing an error in line 11.

5.3.2.1.3. Problem #3

The two objects to compare are *the same* object (identical).

This is less of a problem and more of a way to improve our `equals()` method. If `toBeCompared` is the same literal object that we are comparing it to, then we can make a quick determination and save a few checks.

```
  public boolean equals(Object toBeCompared) {
1
2    if (toBeCompared == this) {
3       return true;
4    }
5
6    if (toBeCompared == null) {
7       return false;
8    }
9
10   if (toBeCompared.getClass() != getClass()) {
11      return false;
12   }
13
14   Student theStudent = (Student) toBeCompared;
15   return theStudent.getStudentId() == getStudentId();
16 }
17
```

Line 3 checks for identity. If `true`, then the remaining checks become unnecessary.

5.3.2.2. Components of `equals`

Almost every `equals` method you write will look similar to the last example above. It will contain the following segments in this order:

1. **Reference check:** If the two objects are the same, return `true` right away.
2. **Null check:** If the argument is `null`, return `false`.
3. **Class check:** Compare the classes of the two objects to ensure a safe cast. Return `false` if the classes are different.
4. **Cast:** Convert the argument to the type of our class, so getters and other methods can be called.
5. **Custom comparison:** Use custom logic to determine whether or not the two objects should be considered equal. This will usually be a comparison of properties or fields.

5.3.2.3. Characteristics of `equals`

Now that we know how to write an `equals()` method, let's look at some characteristics that every such method should have. Following the general outline above makes it easier to ensure that your `equals()` method has these characteristics.

1. **Reflexivity:** For any non-null reference value `x`, `x.equals(x)` should return `true`.
2. **Symmetry:** For any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` also returns `true`.
3. **Transitivity:** For any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
4. **Consistency:** As long as `x` and `y` do not change `x.equals(y)` should always return the same result.
5. **Non-null:** For any non-null reference value `x`, `x.equals(null)` should return `false`.

If you think back to what your math classes had to say about equality, then these concepts make sense.

Using the general approach outlined above to implement `equals()` will make it easier to meet these characteristics. However, always check your method! Missing one or more characteristic can be disastrous for your Java applications.

Tip

Seasoned Java developers will tell you that every time you implement your own version of `equals()` you should also implement your own version of `hashCode()`. `hashCode()` is another special method that every class has. Understanding `hashCode()` would take us a bit far afield at this point, but we would be remiss to not mention it. If you want to read more, check out this article and this stack overflow.

5.3.2.4. Take Away

You may not need to write your own `equals()` method for every class you create. However, as a new Java programmer, remember the following:

Always use `equals()` to compare objects.

This is especially true when working with objects of types provided by Java, such as `String`. A class that is part of Java or a third-party library will have implemented `equals()` in a way appropriate for the particular class, whereas `==` will only check to see if two variables refer to the same literal object.

5.3.3. References

1. How to Implement Java's equals Method Correctly
2. How to Implement Java's hashCode Correctly

5.3.4. Check Your Understanding

Question

Given the code:

```
public class Pet {

   private String name;

   Pet(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }
}

String firstPet = "Fluffy";
Pet secondPet = new Pet("Fluffy");
Pet thirdPet = new Pet("Fluffy");

```

Which of the following statements evaluates to `true`?

1. `firstPet == secondPet;`
2. `secondPet == thirdPet;`
3. `thirdPet.equals(secondPet);`
4. `thirdPet.getName() == firstPet;`
5. `thirdPet.equals(firstPet);`

Question

We add the following code inside the `Pet` class:

```
public boolean equals(Object petToCheck) {

   if (petToCheck == this) {
      return true;
   }

   if (petToCheck == null) {
      return false;
   }

   if (petToCheck.getClass() != getClass()) {
      return false;
   }

   Pet thePet = (Pet) petToCheck;
   return thePet.getName() == getName();
}

```

Which of the following statements evaluated to `false` before, but now evaluates to `true`?

1. `firstPet == secondPet;`
2. `secondPet == thirdPet;`
3. `thirdPet.equals(secondPet);`
4. `thirdPet.getName() == firstPet;`
5. `thirdPet.equals(firstPet);`

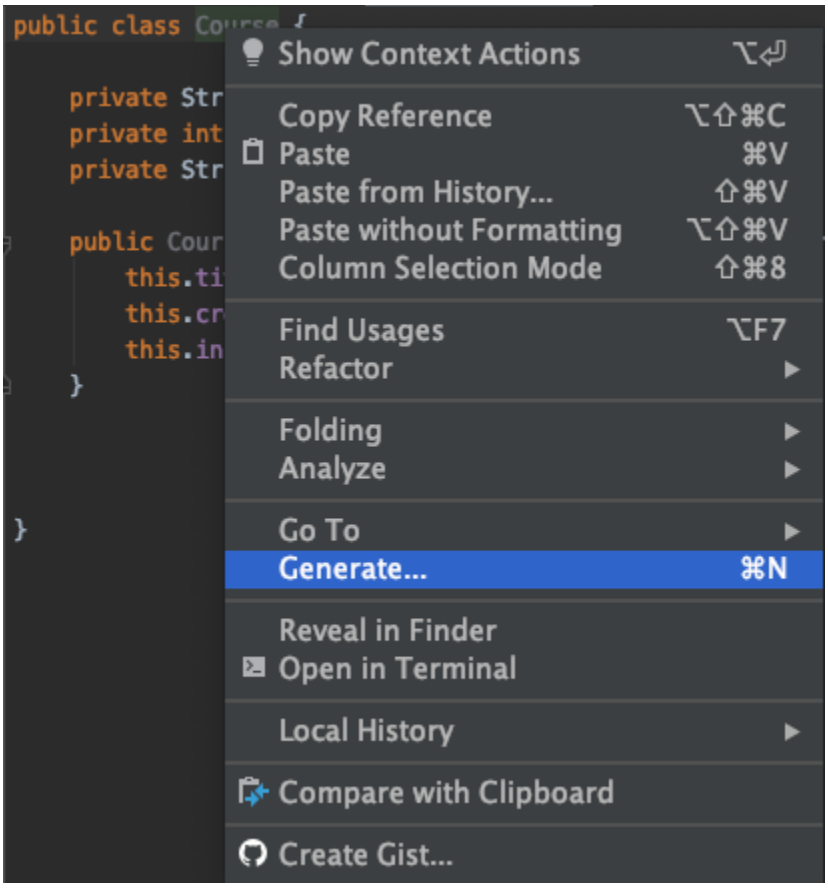
5.4. IntelliJ Generator Shortcut

Instead of cutting, pasting, and refactoring old code to ensure that you create a well-structured `hashCode()` method whenever you define your own `equals()` method, you can use IntelliJ's code generation tool! Just right-click within your class file and select *Generate* > *equals and hashCode* and follow the prompts.

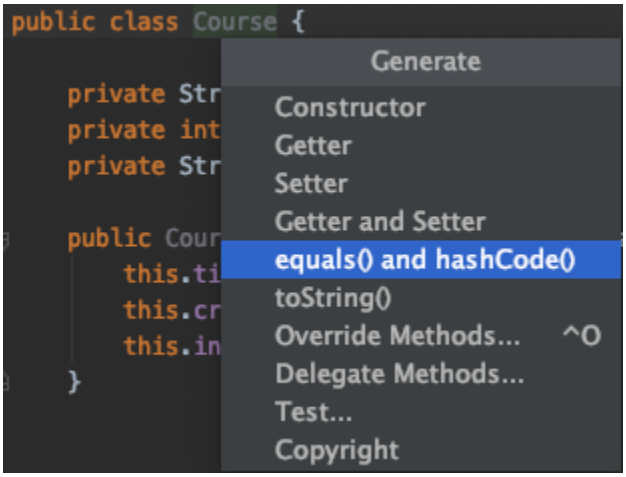
Let's use a `Course` class to demonstrate:

```
  public class Course {
1
2    private String title;
3    private int credits;
4    private String instructor;
5
6    public Course (String title, int credits, String instructor) {
7       this.title = title;
8       this.credits = credits;
9       this.instructor = instructor;
10   }
11 }
12
```

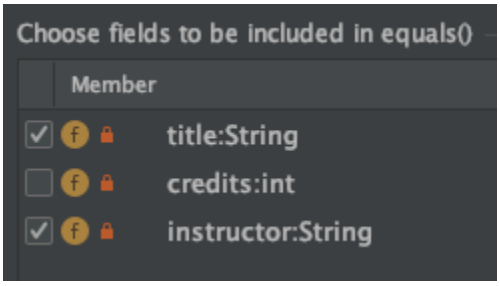
1. In the IntelliJ editor, right-click in the editor (or on the `Course` class name to be really deliberate), then select *Generate* from the menu.



2. Select the *equals() and hashCode()* option:



3. Select the default options until you are asked to choose the fields you want `equals` to consider. Let's assume that two `Course` objects will be equal if they have the same `title` and `instructor` values.



4. Repeat the selections in the next two prompts for the `hashCode` and `null` fields.

5. Click *Finish*. IntelliJ generates the `equals` and `hashCode` methods:

```
  import java.util.Objects;
 1
 2 public class Course {
 3
 4    private String title;
 5    private int credits;
 6    private String instructor;
 7
 8    public Course (String title, int credits, String instructor) {
 9       this.title = title;
10       this.credits = credits;
11       this.instructor = instructor;
12    }
13
14    @Override
15    public boolean equals(Object o) {
16       if (this == o) return true;
17       if (!(o instanceof Course)) return false;
18       Course course = (Course) o;
19       return title.equals(course.title) &&
20                instructor.equals(course.instructor);
21    }
22
23    @Override
24    public int hashCode() {
25       return Objects.hash(title, instructor);
26    }
27 }
28
```

Looking at the new `equals` method shows that it includes all of the best-practice components:

1. Line 17 performs the reference check on the object `o`.
2. Line 18 performs the `null` check and class check on `o`.
3. Line 19 casts `o` as a `Course` object.
4. Line 20 compares the `title` and `instructor` fields of the two objects.

5.4.1. Try It!

Use the *Generate* option to add getters, setters, and a `toString` method to the `Course` class.

COOL!!!!!

5.5. Exercises: Objects and Classes, Part 2

Work on these exercises in the IntelliJ `java-web-dev-exercises` project. You will update your `Student.java` file by implementing the `addGrade` and `getGradeLevel` methods that were sketched out in the Instance Methods section.

5.5.1. The `getGradeLevel` Method

This method returns the student's level based on the number of credits they have earned: Freshman (0-29 credits), Sophomore (30-59 credits), Junior (60-89 credits), or Senior (90+ credits).

5.5.2. The `addGrade` Method

This method accepts two parameters—a number of course credits and a numerical grade (0.0-4.0). With this data, you need to update the student's GPA.

5.5.2.1. GPA Information

GPA is computed via the formula:

gpa = (total quality score) / (total number of credits)

1. The *quality score* for a class is found by multiplying the letter grade score (0.0-4.0) by the number of credits.
2. The *total quality score* is the sum of the quality scores for all classes.

For example, if a student received an "A" (worth 4 points) in a 3-credit course and a "B" (worth 3 points) in a 4-credit course, their total quality score would be: 4.0 * 3 + 3.0 * 4 = 24. Their GPA would then be 24 / 7 = 3.43.

5.5.2.2. Determine the New GPA

To update the student's GPA:

1. Calculate their *current* total quality score by using the formula `gpa * numberOfCredits`.
2. Use the new course grade and course credits to update their total quality score.
3. Update the student's total `numberOfCredits`.
4. Compute their new GPA.

5.5.3. `toString` and `equals`

1. Add custom `equals()` and `toString()` methods to the `Student` class.
2. Add custom `equals()` and `toString()` methods to the `Course` class which you started in the exercises for the previous chapter.

5.6. Studio: Restaurant Menu Continued

We began designing and implementing our `Menu` and `MenuItem` classes in the last studio. Let's continue working on these classes by adding some methods.

5.6.1. Design

To review, here are the details you have from the restaurant owner:

1. The menu consists of several menu items.
2. Each menu item has a price, description, and category (appetizer, main course, or dessert).
3. It should be possible to display whether or not a menu item is new.
4. The app should know when the menu was last updated, so visitors can see that the restaurant is constantly changing and adding exciting new items.

Based on these details, you need to include some *instance* methods:

1. A way to add and remove menu items from the menu.
2. A way to tell if a menu item is new.
3. A way to tell when the menu was last updated.
4. A way to print out both a single menu item as well as the entire menu.
5. A way to determine whether or not two menu items are equal.

Starting with pen and paper (or your favorite notes application on your laptop), sketch out the methods that you need to add to these classes. List the method names and access levels, along with the types of all input and return parameters. Also, consider whether any methods should be `static`.

5.6.2. Share Your Design

Once you have sketched out your methods, pair with a classmate and take turns presenting your designs. Class design can be subjective, so it's important to properly think and talk through your choices before coding.

While your partner is presenting their design, ask questions about why they made the decisions they did. Consider other use cases that might come up, and see if their design fits with those.

5.6.3. Implementation

1. In IntelliJ, open your `Restaurant Menu` project. Open the terminal in IntelliJ and create a branch in your repository for your Lesson 4 Studio solution. For the studio, make sure all of your work is in that branch.
2. Within the `restaurant` package, add the methods you designed to your `Menu` and `MenuItem` classes.
3. Create a class called `Restaurant`, and add a `public static void main(String[] args)` method.
4. Use the `main` method to test your classes:
    1. Create several items and add them to a menu.
    2. Print the entire, updated menu to the screen.
    3. Print an individual menu item to the screen.
    4. Delete an item from a menu, then reprint the menu.

5.6.4. Bonus Mission

If a user tries to add an item that is already on the menu, print a message that warns the user about the duplicate. Also, prevent the duplicate from being added to the menu.

5.6.5. Submitting Your Work

In the previous studio, you should have created a repository on GitHub for your restaurant project.

Since you just modified that project in this studio, commit your changes and push them up to the same repository. However, you must still submit the URL for your repository on Canvas.