THE GREAT COURSES®

Topic
Science

Subtopic
Engineering & Technology

Learning Java Programming

Course Guidebook

Paulo Dichone
Java Programmer

THE GREAT COURSES®

GREAT COURSES LEADERSHIP

President & CEO	PAUL SUIJK
Chief Financial Officer	BRUCE G. WILLIS
Chief Marketing Officer	CALE PRITCHETT
SVP, Marketing	JOSEPH PECKL
VP, Customer Engagement	KONSTANTINE GELFOND
VP, Technology Services	MARK LEONARD
VP, Product Development	JASON SMIGEL
VP, General Counsel	DEBRA STORMS
VP, People	AUDREY WILLIAMS
Sr. Director, Creative & Production	KEVIN BARNHILL
Sr. Director, Content Development	KEVIN MANZEL
Director, Business Operations & Planning	GAIL GLEESON
Director, Editorial & Design Services	FARHAD HOSSAIN
Director, Content Research & Alternative Programming	WILLIAM SCHMIDT

CREATORUP PRODUCTION TEAM

Producers	DAVE O'BRIEN JOSELITO SELDERA
Director	JOSELITO SELDERA
Casting	PERRIN DAVIS
Post-Production Supervisor	HARRISON KUNZ
Post-Production Coordinator	ASHLEY MCGRAW
Editor	CHRISTOPHER ORDAZ
Assistant Editor	MATTHEW HANISCH
Post-Production Sound Designer	NICK WARNER
Head of Client Solutions	CHRISTOPHER CUMMINGS
Cinematographer	ERIK SMITH
Second Camera Operator/Assistant Camera	MICHAEL GOMEZ
Hair and Makeup Artist	IEVA RADINA
Production Assistant	DELILA RIO
Head of Production & Creative Learning	MICHELLE KRAMER-PINO
Production & Platform Manager	DANNY BITTKER
Associate Producer	BRIANNA MCFADDEN
Production Associate	CHRISTINA ROSALES

GREAT COURSES PRODUCTION TEAM

Executive Producer	OCTAVIA VANNALL
Executive Content Developer	JAY TATE
Quality Control Technicians	MARGI WILHELM FRANCISCO VÉLIZ-ORTIZ
Creator of Opening Title Animation	BRIAN SCHUMACHER

GREAT COURSES EDITORIAL & DESIGN SERVICES

Sr. Managing Editor	BLAKELY SWAIN
Sr. Graphic Designer	KATHRYN DAGLEY
Proofreader	JESSICA MULLINS
Transcript Editor	WILLIAM DOMANSKI

Paulo Dichone

Java Programmer

Paulo Dichone is a software engineer, teacher, and consultant. Having earned his BS in Computer Science at Whitworth University, he now teaches Java, Android, and iOS mobile development to more than 100,000 students online in more than 175 countries. He is one of the top mobile development instructors on Udemy. His areas of expertise include Java, Kotlin, Dart, SwiftUI, Lua, and JavaScript. He also specializes in mobile game development and has built several mobile games for the Android and iOS platforms using Corona SDK, a 2D game engine now called Solar2D. As a consultant, he works with small start-ups in different industries, such as teledermatology and augmented reality, and designs and builds scalable business software.

Table of Contents

Introduction

Guides

Supplementary Material

Learning Java Programming

A computer program is a set of instructions that's explicitly written for computers to perform tasks. These instructions are often written in a high-level programming language, such as Java. To instruct computers to do specific tasks, you must learn a programming language's syntax, just like you would if you wanted to communicate well in a foreign language. This course takes you on a journey to learn how to write computer programs using Java—one of the world's most popular programming languages. The course also exposes you to how Java is used to build desktop graphical user interfaces (GUIs) and Android mobile applications.

In the first part of the course, you'll learn how to set up your machine for Java development by installing the necessary tools to start learning Java programming. You'll then learn how the Java compilation process works internally. You'll explore how the Java Development Kit and the Java Runtime Environment work together to process and compile Java code into a different code format, bytecode, which holds the instructions that computers or devices can run. You will learn the Java program's building blocks by looking at the Java code structure and Java basic syntax. Additionally, you'll discover how to store data using predefined Java variable types. Because Java programs are dynamic entities—they allow for data manipulation and decision-making—you'll learn how to control your programs' flow by using conditional statements. You'll also explore ways to execute code repeatedly.

In the second part of the course, you'll dive into a different realm of Java programming, object-oriented programming (OOP), a programming paradigm that looks at computer programs as a group of objects that interact with each other. You'll explore how OOP enables the creation of robust, more complex Java programs. OOP provides a way for breaking down complexity into smaller, manageable pieces of code. You'll write your first Java class—the blueprint of an object—and learn how to instantiate (create) an actual Java object from a Java class. You'll learn how to construct a bullet-proof class that only exposes its internal state (behaviors and properties) through appropriate,

safe mechanisms to maintain the class's integrity. Then, to further your OOP knowledge, you'll explore Java Inheritance, which is a way to leverage preexisting class attributes and behaviors in a treelike class hierarchy. Java Inheritance is one of the most valuable Java features that allows you to reuse code. In other words, the code that's already present in the parent class doesn't have to be rewritten in the child class; the child class can inherit the parent class's properties and behaviors, therefore reducing code redundancy. Understanding how Java OOP and Java Inheritance work and their benefits will give you a more holistic view of how software engineers build large software or break down complex problems into smaller pieces that are easy to solve.

In the third part of the course, you'll be introduced to Java Swing: a collection of Java classes used to build amazing GUIs. At this point in the course, you'll have learned enough about Java to start building interactive user interfaces, such as creating windows that pop up on the computer screen with buttons and fields the user can interact with. You'll learn how to leverage your Java knowledge along with Java Swing and Abstract Window Toolkit (AWT) to create virtually any GUI you'd like. You'll learn how to lay out your GUI components on the screen using Java Swing built-in classes. You'll finish this segment by building, from the ground up, a full-fledged Java GUI application that draws a circle in the middle of a window, and each time you click anywhere on the frame, the circle changes colors randomly.

The fourth part of the course introduces Android mobile app development using the Java skills you've acquired throughout the course. You'll start by learning what Android, as a mobile operating system, is. Next, you'll learn about setting up Android Studio, the integrated development environment used to build Android mobile apps. You'll then learn Android as a mobile development platform and use your Java skills to tap into the Android development platform.

The final segment's primary goal is to give you an overview of building Android apps with Java. You'll also have a full understanding of the main Android building blocks, such as creating views, buttons, and a few other

Android user interface widgets. Additionally, you'll learn how to run Android apps on a physical device. As a final project for this segment, you'll build an Android app called Quotes, which randomly picks a quote from a list and displays it on the screen when a button is tapped. By the end of this segment, you'll also have learned how Android projects are structured, how to design Android user interfaces, and how the user interface interacts with the Java code.

By the time you complete this course, you will have a solid understanding of Java as a programming language. You'll also know how to leverage Java to build stunning GUIs with Java Swing and AWT libraries as well as create Android mobile apps. This course takes you on a journey of learning Java programming and how versatile Java is—from writing simple Java programs, to building complex GUIs and software, to building mobile Android apps.

1

Welcome to Java!

As one of the most popular programming languages, Java is used for various purposes in many industries. Java can be used to build desktop applications, web applications, mobile apps, back-end servers, and enterprise applications, such as streaming services and e-commerce websites. Java is also used in many development frameworks and tools for data analysis as well as artificial intelligence, self-driving cars, and smart-house devices like security cameras and thermostats.

There are a few things to keep in mind as you go through this course:

1. If possible, type along with the presenter; programming is learned best by doing.
2. All code is available at this web address: www.TheGreatCourses.com/learningjava.

Java is a programming language that was first released in the 1990s. A programming language allows humans to give instructions to machines, such as computers and TVs. Machines only understand machine code (0s and 1s), so to interface with computers—in other words, to instruct devices to do something useful—you need a programming language, such as Java.

So you write a set of instructions using Java, and then those instructions are translated into machine code (which most people find hard to read). Finally, the machine or device would run those instructions and do something (hopefully) useful.

TIP

Java is a programming language. You can think of a programming language as a tool that enables programmers to solve computational problems and build software.

Here's how Java works: First, a programmer writes Java code in a source file. Next, the source goes through a compilation process through a compiler, which checks the validity of the source code (for things like errors and syntax). Then, the compiler creates a new document coded into Java bytecode, which contains all the instructions machines can interpret/translate into something it can run.

Here's where Java shines: The bytecode is platform-independent, which means it can (potentially) run on any device.

The only requirement is that the device must have a Java Virtual Machine (JVM), which is a piece of software that can be installed on devices. This is what reads the bytecode and runs those instructions on the device.

JVM comes included in the Java Development Kit (JDK)—a collection of tools that enables you to write and run Java programs—so you won't need to separately download JVM. You also won't even need to download the JDK, because you'll be downloading an integrated development environment (IDE) that includes the JDK packaged together with lots of other tools.

You can think of an IDE as a code editor that has all of the tools and features needed for programmers to write code. Android Studio is the official IDE for Android development—for building Android mobile apps—and you can use it to write plain Java programs, too.

There are other programming languages that can use JVM like Java does, such as Clojure, Scala, and Groovy.

The main idea is that the JVM, as a piece of software, can be implemented by device manufacturers so that their devices have it installed when they ship those devices to the end users. This is what makes Java portable. And portability is what has made Java a very successful programming language to this day.

TIP

It's easy to get lost when trying to choose the best IDE or text editor when you first start learning programming. You don't need an IDE or a fancy code editor to write Java programs. In fact, some may argue that it's best to have a simple, basic code editor while learning Java and move on to an elaborate IDE once you've mastered the language.

Resources

Learn Java, https://www.learnjavaonline.org/.

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

2

Choose an Integrated Development Environment

There are many code editors and IDEs you can use to write Java code. Here are a few IDEs that are recommended for Java programming:

1. IntelliJ IDEA is versatile in that you can use it to write programs in multiple languages. Android Studio was built on top of IntelliJ, so most features you'd find in Android Studio you'll also find in IntelliJ IDEA. However, in order to write Java programs using IntelliJ, you'll need to download and install the JDK and set up the `classpath` as an environment variable.

2. NetBeans comes bundled with the JDK, so you wouldn't need to do the extra setting up of the machine environment. NetBeans is a very powerful IDE because it simplifies the development of web, desktop, enterprise, and mobile applications that use Java and other web technologies. This is part of the reason why Oracle, the current sponsor of Java, highly recommends this IDE. If you want to develop web enterprise applications in the future with Java, NetBeans is a good choice. But you can also use it to learn to write simple Java programs.

3 Installing Android Studio for Mac

4 Installing Android Studio for Windows

Let's install Android Studio, which is the tool you'll be using to learn Java and later to build an Android app. The installation process for both Windows and Mac users is straightforward.

First, go to this link: https://developer.android.com/studio.

Scroll to the bottom of the page and make sure that your machine meets the system requirements.

Click on Download Android Studio.

Once the download is complete, move the file onto your desktop and then double-click it to start the installation process. An Android installation wizard will appear, and the installation will start.

After a few moments, the Android Studio wizard will prompt you with a Finish button. Click Finish.

For more detailed instructions, follow along with the video lesson.

5

Create Your First Java Program!

Start Android Studio.

First, you need to have an Android project and then create a Java lesson inside the Android project.

1. Click File > New lesson. Select Java Library and click Next.
2. Fill in the package name (`com.learnjavateachco.firstjava`) and click Finish.

You should now see a Java lesson inside your Android project.

Next, let's add some code. For now, you don't need to understand what this code does.

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

Now click Run and look at the bottom of the IDE to see `Hello World` printed out.

```
Hello World
```

Congratulations! You've created your first Java program.

TIP

You can use a shortcut to write `System.out.println` in Android Studio (and IntelliJ). Simply start typing `sout` and hit the Enter key.

Next, let's change the current code so that it prints out the text `Java programming is fun!`

To change the output, you simply replace the contents of `println("")` with `Java programming is fun!`

```
System.out.println("Java programming is fun!");
```

Run the program once again and you'll see:

```
Java programming is fun!
```

Resources

Learn Java, https://www.learnjavaonline.org/.
Schildt, *Java*.
Sierra and Bates, *Head First Java*.
Tutorials Point, https://www.tutorialspoint.com/java/.

6

Java Code Structure, Syntax, and `main` Method

Let's take a closer look at this program:

```
public static void main(String[] args) {
  System.out.println("Java programming is fun!");
}
```

At the top, you see `FirstJava.java`.

The extension `.java` tells the compiler that inside of this file, it should expect Java code.

Inside the source file is where you write Java code. The Java code must be written inside of a class, which represents a piece of your program. In this case, your class is called `FirstJava`.

```
public class FirstJava {
   public static void main(String[] args) {
       System.out.println("Java programming is fun!");
    }
}
```

Notice that at the very top of the file is a package keyword: `com.learningjava.firstjava`. That simply declares where the source file lives.

The class must go within a pair of curly braces.

A class may have one or more methods (yours only has one). Methods hold a set of statements.

Now remove the semicolon after the `System.out.println()` statement and try running the program.

```
error: ';' expected
        System.out.println("Java programming is fun!")
```

You should see an error telling you that something went wrong. First, you see a little warning in your code, and when you run the code, you receive a log in the console.

The compiler is telling you that it can't compile your source file because it found an error that needs to be fixed.

In Java, the syntax is essential: If you miss something, your code most likely won't run. Your code must follow Java basic syntax for it to compile.

Resources

Learn Java, https://www.learnjavaonline.org/.

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

7

Declaring Variable Types: `int` and `String`

The entry point to any Java program is the `main` method. Whatever you write inside of the `main` method runs first.

```
public static void main(String[] args) {
    System.out.println("Java programming is fun!");
}
```

In this code, what you want to be printed out is written inside of the `main` method.

Right now, this program is straightforward, but as you create more complex applications, you'll have many different pieces (classes) that do different things, but in the end, everything must go through the `main` method to execute.

Inside of the `main` method, you can

1. declare variables,
2. assign variables,
3. write statements,
4. use loops, and
5. use branching.

Almost all programming languages that inherit from C (another programming language) have a `main` method equivalent that serves as the main entry point to the program.

Exercise

Read the following Java statement and choose the correct answer.

```
String name = "Jeff";
```

a It's assigning the variable `name` to `"Jeff"`.

b It's comparing `name` to `"Jeff"`.

c `"Jeff"` is part of string theory.

TIP

Almost all programming languages (if not all) have variables, or at least a similar concept. This makes sense because programs need data to work with, and variables are the simplest way of storing data in memory so programs can use that information to compute and then output something.

TIP

The double equal sign (`==`) is a binary operator in Java that is used to compare primitives. It always returns a `boolean` (`true` or `false`).

Solution

a The Java statement is assigning a variable called `name` to hold `"Jeff"`.

The thing to remember from this exercise is that the equal sign (`=`) in Java means "assignment." Whenever you want to store something into a variable, you use `=`. Java expresses "equals" with the double equal sign (`==`).

In this statement, you've created a variable called `name` and assigned `"Jeff"` to it.

```
String name = "Jeff";
```

So now, whenever you want to access the text `"Jeff"`, all you need to do is invoke the `name` variable.

So now you can say something like this:

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

And run this code.

Notice how passing `name` inside of the parentheses gives you `"Jeff"` in the console. This tells you that a variable is simply a container—a holder for something (in this case, for `"Jeff"`).

Variables are essential because they make it much easier for programmers to hold data (pieces of information), work on the data (change the data), and then output a result that can be passed on to another method (or variable, for that matter) for further processing, and so on.

In this example, `name` was holding `"Jeff"` (a piece of data). Then, you called `name` inside `println()`, which printed the contents of the `name` variable.

What's great about variables is they can be changed to hold something else (hence the name *variable*).

You can change the variable contents by changing the string `"Jeff"` to `"Fred"`, like this:

```
String name = "Fred";
```

When you run this code, `"Fred"` is printed out in the console.

The compiler will always acknowledge the last-added contents to the variable. The assumption is that the last occurrence of that variable should be the most up-to-date data by the logic of what programs do. Java programs are interpreted from top to bottom, so the compiler always starts reading the source file from the top of the file to the bottom.

In the `"Jeff"` statement, the `String` keyword tells you what kind or type of variable you have. So this Java statement can be read as follows:

> *Create a variable called `name` that only stores a string `"Jeff"`.*

To declare a variable in Java, you must specify the type of the variable.

TIP

In Java, variables must have types associated with them.

When you declare a variable, essentially you are telling the computer to allocate space in memory to store something. Java must specify what kind of space needs to be allocated for that variable. In other words, what type of stuff will the variable be holding?

Types are fundamental in Java. Anytime you want to create a variable and assign it to something, you must include a type.

Types in Java are important because all operations are checked for type compatibility, so illegal operations, such as creating a variable without a type, will not compile in Java. This is why having a strong type-checking helps prevent errors, making the language reliable. This makes Java a strongly typed programming language.

Another common type in Java is `int`, which stands for *integer*, a whole number (1, 3, 19, 100, etc.). Here's how to declare an `int` type:

```
int age = 25;
System.out.println(age);
```

Run the code and you will see the number printed out:

```
25
```

Pay close attention to the syntax! You don't write `Int` or `integer` as the type. It must be the lowercase `int`.

The general syntax for creating a variable is as follows:

```
Type name;
```

This creates a variable with a specific type, but nothing is assigned to it. So it would be like asking for the computer memory to allocate some space in memory that's empty.

Or you could have this:

```
Type name = something;
```

This creates a variable (with a specific type, of course) and assigns something to it.

To create a string type, you must add double quotes (`" "`) around the word or text. Anything inside double quotes is considered a `String` in Java.

Exercise

Declare three variables:

1. a `String` type called `firstName` (and assign `James` to it)
2. a `String` type called `lastName` (and assign `Bond` to it)
3. an `int` type called `age` (and assign `45` to it)

Then, print out the following text:

```
"Hi, my name is James Bond, and I am 45 years old."
```

Solution

The challenge here is that you'll need to pass the variables you've declared previously in the `println()` as variables, so the contents of the variables are extracted out when you run the program. Essentially, your printout should look like this:

```
System.out.println("Hi, my name is firstName
lastName, and I am age years old.")
```

(Hint: Google "Java string concatenation operator".)

Resources

Learn Java, https://www.learnjavaonline.org/.

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

8

Concatenating Variables in Java

When you put strings together—one after another—it's called concatenation in Java. It's like gluing pieces of strings into one long string.

Here you have two `String` types and the `int` type called `age`:

```
String firstName = "James";
String lastName = "Bond";
int age = 45;
```

Now print out the sentence along with your variables.

The obvious solution would be to write the following:

```
System.out.println("Hi, my name is firstName lastName
and I am age years old.");
```

But this doesn't work. You get a lot of red squiggly lines.

For you to pass variables along with text, you need to use a Java concatenation operator, the plus sign (+), like this:

```
System.out.println("Hi, my name is" + firstName +
lastName + " and I'm "+ age + " years old.");
```

When you run this code, you have a nice output, but there are a few cosmetic issues:

```
Hi, my name isJamesBond and I am 45 years old.
```

There are no spaces between *is* and *James* and *Bond*.

To solve this issue, you need to add some empty spaces, like this:

```
System.out.println("Hi, my name is " + firstName +
" " + lastName + " and I'm " + age + " years old.");
```

In other words, you can add spaces by adding empty strings!

Exercise

Try concatenating `String` and `int` variables:

1. Create two or three `String` variables and assign values to them.
2. Create one or two `int` variables and assign values to them.
3. Write a short story about yourself.

Here's an output example:

```
"Hi, I'm Paulo and I'm 100 years old. I grew up in
a small town, called HappyVille. I was a very happy
child in Happyville. I have 13 brothers and sisters."
```

Solution

```
String name = "Paulo";
String bio = "I grew up in a small town, called
HappyVille. I was a very happy child in HappyVille.";
int age = 100;
int siblings = 13;
System.out.println("Hi, I'm " + name + " and I'm " +
age + " years old. " + bio +" I have " +siblings +
"brothers and sisters.");
```

Resources

Learn Java, https://www.learnjavaonline.org/.

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

9

Primitive Variable Types: `boolean` and `char`

In Java, different types of variables have different sizes. Java divides its types into 2 main categories:

- primitive types
- reference types

Primitives are types that hold different sizes. There are eight primitives in Java. Each Java primitive has a set number (capacity) of bits it can hold. In computer jargon, a bit is the smallest unit of data or information. One bit essentially represents `1` or `0` (meaning "on" or "off," respectively).

The first primitive is a `boolean`, which can only hold either `1` or `0` (`true` or `false`). The size of a `boolean` type is 1 bit, and the value range is `true` or `false`.

Here's how to declare a `boolean` variable:

```
boolean isHappy = true;
boolean isDone = false;
```

Notice that `true` and `false` are not inside of double quotes—they are not `String` types.

Another primitive is `char`. A `char` represents one character only. A `char` can hold 16 bits, making it considerably larger than a `boolean` type.

Another important visual distinction for `char` is that you use single quotes (`' '`), not double quotes.

Resources

Learn Java, https://www.learnjavaonline.org/.

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

10

Primitive Variable Types: `byte`, `short`, and `long`

The most popular primitive you've learned about is the `int`, which represents a whole number. Generally, an `int` is the type you will be using the most, whenever you need a large enough type that can hold a bigger range of numbers.

An `int` can hold 32 bits, which means it can hold four times as much as a `char` can hold (which is 16 bits). But remember that although these two are both primitives, they are still different types.

But there are instances when you may need a small container to store data. That's when you would use a byte, which holds 8 bits and has a smaller value range than `int` (32 bits) that it can hold.

Here's how you create a `byte` variable and add a value to it:

```
byte aByte = 78;
```

Now let's test the value range for a `byte` type. Let's start by assigning `124`.

```
byte aByte = 124;
```

When you run the code, you see no errors.

But when you increase the number to `128`, the IDE tells you that this won't work.

```
byte aByte = 128;
```

The IDE even tries to help you by telling you to change the variable type to an `int` for this to work.

This error tells you the value range for a `byte`: It can only hold values from –128 to 127. Beyond that range, you will get an error.

In addition to `byte` and `int`, there's a midrange type called `short` that holds 16 bits. In cases when you need a container that's a little more than a `byte` (8 bits) and less than an `int` (32 bits), you use a `short`.

Here's how you declare a short:

```
short aShort = 30000;
```

The value range for a short is –32768 to 32767.

TIP

Don't try to memorize these value ranges and how many bits each primitive can hold. Just know that there are different types, each of which has a defined bit size and value range.

Exercise

Without writing this code in the IDE, do you think this will work? Why or why not?

```
int x = 13;
 byte a = x;
```

Solution

Without writing the code in the editor, you would think that it would work (13 is certainly small enough to fit into a `byte` type). However, the compiler will always look at `x` as an integer, so it will not allow an integer to be added into a smaller container. Hence, this won't work:

```
int x = 13;
 byte a = x;
```

If you write this piece of code in the code editor, you'll see that before you even attempt to run the program, errors will pop up.

The key point to remember here is that you can't forcibly mix types in Java.

The final numeric type for you to see is the `long` type, which you use when dealing with larger numbers.

Here's a `long` variable:

```
long aLong = 7876452;
```

A `long` is 64 bits and has a huge value range, too! So you'd use a `long` in cases where you know you'll need a very large container to hold your numbers.

Resources

Learn Java, https://www.learnjavaonline.org/.

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

11

Primitive Variable Types: `float` and `double`

So far, you've learned about ways to express whole numbers, from small to large. But what about decimal numbers?

You can express decimal numbers, such as pi (3.14) or $34.99, with floats and doubles.

A `float` is a type that holds any number that has decimal points. A `float` is 32 bits, and its value range varies.

Here's how to create a `float` and assign it 3.14:

```
float pi = 3.14f;
```

The `f` at the end of the number makes your variable a `float` in Java.

A `double` is 64 bits, and its range value also varies. A `double` is the best type to express monetary value:

```
double expenses = 234.89;
```

The reason to use one instead of the other depends on the level of precision you want. In Java, doubles provide more precision than floats. A `double` provides more than 10 decimal points, whereas a `float` can only give 6 to 7 decimal points.

As an example, let's say you're responsible for writing a program that calculates a precise location for a spaceship to land and that you need to divide the value of pi (3.14) by 212. Which should you use: a `float` or a `double`?

When you run this code, you get this result:

```
double dPie = 3.14/212;
float fPie = 3.14f/212;
System.out.println("Double location: " + dPie);
System.out.println("Float location: " + fPie);
Double location: 0.014811320754716981
Float location: 0.014811321
```

The location is more accurate when you use a `double` type—which has a precision of up to 18-digit decimal points—whereas `float` is less accurate.

And with these two types, `float` and `double`, your learning about primitive types concludes.

Resources

Learn Java, https://www.learnjavaonline.org/.

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

12

Java Operators and Operator Precedence

In Java, you can do number crunching quite easily using `+`, `-`, `*`, `/`, and `%`.

Exercise

1. Create two variables of type `double` called `a` and `b`.
2. Assign `10` to `a` and `4` to `b`.
3. Create another variable of type `double` and call it `sum`. Assign `sum` the summation of `a` and `b` (add `a` to `b`).
4. Use the `+` operator to add `a` and `b` and assign the result to a variable `sum`.
5. Then, print out the `sum` variable, along with this sentence:

```
The sum of 10 and 4 is 14.
```

Solution

```
  double a = 10;
  double b = 4;
 double sum = a + b;
System.out.println("The sum of " + a + " and " + b +
" is " + sum +".");
```

You can do the same if you want to subtract two numbers, except instead of using the `+` operator, you'd use the `-` operator:

```
double result = a - b;
System.out.println("Result is " + result);
```

A division expression uses `/`:

```
double divisionResult = a / b;
System.out.println("Division result " +
divisionResult);
```

You can multiply numbers by using `*`:

```
 a * b
```

You can also find the remainder from a division expression using the `%` operator, called the remainder operator or the modulo operator:

```
int remainder = a % b;
```

This should give a result of `2`.

You can employ, for example, multiplication expression as part of a more complicated statement:

```
int total = 100 + (a * 10);
```

You can also increment or decrement a variable by 1.

To decrement a variable by 1, you add a `--` suffix to the variable:

```
int age = 10;
 age--;
```

To increment a variable by 1, you add a `++` suffix to the variable:

```
int age = 10;
 age++;
```

Exercise

Without testing this code in the code editor, what's the final value of `y`?

```
int y = 10 * 4 + 5;
  y--;
```

a 45 **b** 44 **c** 98.5

Solution

To start, 10 * 4 + 5 is 45, and then you decrement the value of `y` by 1, so the correct answer is **b)** `44`.

Look at this expression alone:

```
int y = 10 * 4 + 5;
```

Run the code and you get this:

```
45
```

This tells you that multiplication takes precedence over addition.

But if you want to force addition to take precedence over multiplication, you can write it this way:

```
int y = 10 * (4 + 5);
```

Now the result is this:

```
90
```

What's inside parentheses now takes precedence, hence the new result.

Here's another expression:

```
int x = 3 * 2 + 4 * 10 / 2;
```

The result is this:

```
26
```

That's because this is the precedence order in this example:

1. multiplication, division
2. addition

Here's one of the possible ways the compiler evaluates this expression:

3*2 = 6
4*10 = 40
40/2 = 20
20 + 6 = 26

Understanding how operator precedence works in Java is crucial because you could end up with the wrong result even though you've plugged in the correct numbers.

Exercise

Calculate the weight of an object on several different planets, given the gravitational constants for a few planets and the formula you'll need to calculate the weight on the respective planets.

To calculate weight on all other planets, you need the following:

1. weight on Earth
2. gravitational constant for the planet
    a. Venus: 0.91
    b. Mars: 0.38
    c. Jupiter: 2.34
    d. Saturn: 0.93
3. weight on Planet *x* = weight on Earth * gravitational constant

How much would a 190-pound object on Earth weigh on Mars, Jupiter, Venus, and Saturn?

Here's basically how the output should look:

```
"Object's weight on Earth is 190."
"Object's weight on Venus is x."
"Object's weight on Mars is y."
```

...

Solution

Often in programming, there are many different ways to solve a problem, so this solution may look completely different from the way you chose to solve the problem. And that's fine—as long as you get the correct output.

So here's one way to solve this problem:

1. Create a variable that holds the Earth weight.
2. Output the base weight, which is the Earth weight.
3. Calculate the Venus weight by multiplying the Earth weight by the gravitational constant for Venus.
4. Output the result.
5. Repeat the same process for the other planets.

```
double weight = 190;
System.out.println("Object's weight on Earth is " +
weight);
double venusWeight = weight * 0.91;
System.out.println("Object's weight on Venus is " +
venusWeight);
double marsWeight = weight * 0.38;
System.out.println("Object's weight on Mars is " +
marsWeight);
double jupiterWeight = weight * 2.34;
System.out.println("Object's weight on Jupiter is " +
jupiterWeight);
double saturnWeight = weight * 0.93;
System.out.println("Object's weight on Saturn is " +
saturnWeight);
```

Remember, Java statements can use mathematical expressions by employing the operators `+`, `-`, `*`, `/`, and `%`. Keep in mind that operator precedence is very important.

If your expression has all 5 of these operators employed, then the following evaluation order is used when the compiler works out the expression:

1. Incrementing and decrementing come first.
2. Multiplication, division, and modulus division occur next.
3. Addition and subtraction follow.
4. Comparison comes next.
5. The equal sign (=) is used to set or assign the variable to a value.

Resources

Learn Java, https://www.learnjavaonline.org/.

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

Quiz

1. What is a compiler?

2. What is bytecode?

3. True or false? The following Java statement will compile without any errors:

```
    system.println(“Hello World”);
System.out.println(“Hello World”);
```

4. What is the purpose of the `main()` method in a Java program?

5. What is a variable?

6. What is a variable type?

7. Do the `float` and `double` primitive types have the same precision?

Quiz Answers

1 A compiler is a special program that translates code written in a high-level programming language, such as Java, into bytecode or machine code.

2 Bytecode is a compiled Java code format that's ready to be executed on a computer or device running the JVM, a program that interprets Java bytecode.

3 False. The statement has two syntax errors: `system` should be `System` and is missing the `out` static member. The correct statement should be this:

```
System.out.println(“Hello World”);
```

4 The `main()` method is the Java program's entry point. It's where you can declare and assign variables and write Java statements.

5 A container (inside the computer's memory) that can hold a piece of data that can be referenced.

6 A variable type is a keyword or identifier that specifies what kind or type of data the variable will hold.

7 No. A float is less precise than a double. A float is single-precision, meaning that it has less significant digits to the right of the decimal, whereas a double is double-precision.

13

The `while` Loop in Java

Programs can be made more autonomous by enabling them to make decisions based on the states and conditions of variables.

In essence, programs are supposed to be able to make decisions based on the flow of data—the input you give to them.

In real life, you could give your friend the following instructions:

> To get to the mall, you'll drive 10 miles north on Bella Street and hang a left on Martin road. If you see a big mango tree to your left, you're getting close. Keep going until you see a big clown waving at everybody. Then the mall is going to be on your left.

When these instructions are written out like this in English, they make sense. A person given these instructions would know what to do.

But how do you translate these instructions, with some conditions that must be met to get to the final destination successfully?

Loops and branching help programs do just that: repeat tasks, branching those tasks until the final condition is met.

Let's say you want a certain block of code (block of statements) to run until a specific condition is met. As an example, let's simplify the mall directions for learning. To get from A to B, your friend will need to keep driving until mile 10. This means that your friend must keep driving until a specific condition is met: Get to mile 10, and at that point, they no longer need to continue driving because they'll have arrived at the final destination.

To translate these instructions for the computer, you need to express the following in code:

> *While mile 10 has not been reached, keep driving until mile 10 is reached, then stop.*

This is called pseudocode. It's a mix of English and code. Essentially, it's a notation resembling a simplified programming language that's used in program design.

To translate this pseudocode into actual Java code, you use a `while` loop.

First, you need a few variables that will control the flow of the `while` loop:

```
int destination = 10;
int counter = 0;
```

The `destination` variable is the final destination—in this case, at mile `10`. The `counter` variable will help keep track of the progress as the friend keeps driving.

Then you have the actual `while` loop:

```
while(){
}
```

Inside the parentheses, you need to put the conditional test, which is a conditional statement, and then whatever is inside the curly braces will keep running until the conditional test is no longer true.

```
while(count < destination) {
   System.out.println("Keep driving");
}
```

You must keep track of the progress as you loop through. Otherwise, you'll end up with an infinite loop—meaning that the `while` loop will run forever, or at least until your computer runs out of memory.

To avoid this, all you need to do is update the counter by incrementing each time the loop runs by 1:

```
while(count < destination) {
   System.out.println("Keep driving");
count++;
}
```

Run this code and you will see this output 10 times:

```
Keep driving
Keep driving
Keep driving
Keep driving
Keep driving
Keep driving
Keep driving
Keep driving
Keep driving
Keep driving
```

As soon as the `while` loop is executed, the conditional test will be evaluated. The first time, it will check if `count` (which is `0` initially) is `< destination` (`10`). If it is, then it will run what's inside the curly braces (print out the text and increment count by 1). The counter will then be at 1, and the process continues until the test conditional is no longer true: when the counter is 10 (note that 10 is not less than 10).

TIP

Always have a counter that's incremented inside the `while` loop to avoid infinite loops! Also, in case you end up having an infinite loop, just terminate your program by clicking on the Stop button.

When the conditional test evaluates to false, then the program breaks from the `while` loop and continues reading what comes next.

So you could add another printout to let your friend know that they have arrived!

```
System.out.println("You've arrived at mile " + count);
```

Make sure that inside the `while` loop you include a counter, which increments each time the `while` loop runs. Otherwise, you'll have an infinite loop. Your computer will end up slowing down and eventually use up all the memory and shut down.

Resources

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

14

Java Branching Statements: `if`, `if-else`, and `else-if`

The fun part of programming is that it allows you to create programs that do things depending on certain conditions. For example, you can express things like this:

> *If your bank account has less than 1000, you shouldn't go to Vegas.*

To express that in code, you use the `if` statement, like this:

```
int bankAcct = 900;
 if (bankAcct < 1000) {
            System.out.println("Nope. Just stay home.
Vegas is too expensive.");
        }
```

The `<` ("less than") operator, along with its opposite, the `>` ("greater than") operator, are called conditional operators.

Notice what happens when you change from `<` to `>`:

```
if (bankAcct > 1000) {
            System.out.println("Nope. Just stay home.
Vegas is too expensive.");
        }
```

When you run the code, it looks like nothing happened. But something did happen: The conditional test evaluated to false, hence whatever you have inside the curly braces didn't run, since 900 is not greater than 1000.

TIP

When dealing with `while` loops or `if` statements, the conditional test (what goes inside of the parentheses) always evaluates to either false or true. In other words, `while` loops and `if` statements depend on `boolean` variables to run tasks. Also, whenever you are comparing, you use `==`, not `=`.

What if you wanted to test if `bankAcct` is equal to 1000?

The obvious answer would be to just swap `>` for `=`:

```
if (bankAcct = 1000) {
            System.out.println("Nope. Just stay home.
Vegas is too expensive.");
        }
```

But that doesn't work. The IDE even gives you the correct option to be used, which is `==`:

```
if (bankAcct == 1000) {
            System.out.println("Nope. Just stay home.
Vegas is too expensive.");
        }
```

The `=` didn't work because `=` in Java is used to assign values to a variable, not to compare. When it comes to comparing two things, you always use `==`.

Run this code and you still will see nothing because the conditional test resolves to false: 900 is not equal to 1000.

In addition to *if*, there's another conditional keyword called the `if-else` statement, which gives you branching:

```
if (bankAcct > 1000) {
   System.out.println("Nope. Just stay home. Vegas is
expensive.");
}else {
System.out.println("You're doing alright. Have fun.");
}
```

Now, by adding `else`, you can say that you want to display/print out a different message if the `if` statement evaluates to false:

```
You're doing alright. Have fun.
```

Also, change `<` to `>` so you get a false—that way, the `else` branch runs. Here's the new output:

```
if (bankAcct > 1000) {
   System.out.println("Nope. Just stay home. Vegas is
expensive.");
}else {
System.out.println("You're doing alright. Have fun.");
}
You're doing alright. Have fun.
```

In addition to `==`, there's also the opposite of this operator: not equal. To express this new conditional operator, you use `!=`.

If you swap `>` with `!=`, the statement inside of `if` (the first printout) would show in the console because 900 is not equal to 1000.

```
!=
if (bankAcct != 1000) {
       System.out.println("Nope. Just stay home.
Vegas is expensive.");
}else {
      System.out.println("You're doing alright. Have
fun.");
  }
Nope. Just stay home. Vegas is expensive.
```

So you can branch your programs using `if` and `if-else` statements along with conditional operators (`==`, `<`, `>`, `!=`) to create fairly complex programs that can evaluate a conditional expression and decide what needs to run.

As simple as it may sound, this in essence is what programming is all about: manipulating data (evaluating conditional tests, assigning values, and calculating an output or doing something).

Next, write the following code:

- Create a `char` variable and assign `A`.
- Create an `if-else` statement.
- Write the printouts.

```
char grade = 'A';
if (grade == 'A') {
   System.out.println("A is for Awesome!");
}else {
   System.out.println("Did you get a B, a C, or
an F?");
}
```

Run this code and it prints out the following:

```
A is for Awesome!
```

Change `grade` from `A` to `B`:

```
grade == 'B'
```

Run this code and now it prints this:

```
Did you get a B, a C, or an F?
```

This is exactly what you would expect. But what if you wanted to account for a `B`, `C`, or `F` and output a meaningful message? How would you do that?

The `else` statement does not have a condition listed alongside it like the `if` statement does. The `else` statement always goes along with the `if` statement, like this:

```
if(condition){
//statements
}else {
//statements
}
```

You can use `else` to chain several `if` statements together, which gives more power for branching your code flow!

By adding an `else-if`, you can pass a conditional test that, once evaluated to a `boolean`, will either print the statement inside the code block or not. So if `grade` is equal to `B`, then `B is for Beautifully done!`.

```
else if (grade == 'B') {
   System.out.println("B is for Beautifully done!");
}
```

TIP

Avoid using long chained `else-if` statements when possible; it makes the code too hard to read.

You can keep going with the chaining by adding more `else-if` statements to run if the conditional tests evaluate to true. The complete code looks like this:

```
if (grade == 'A') {
   System.out.println("A is for Awesome!");
} else if (grade == 'B') {
   System.out.println("B is for Beautifully done!");
} else if (grade == 'C') {
   System.out.println("C is for Careless");
} else {
   System.out.println("F is for Future unknown");
}
```

Now, anything that is not an `A`, `B`, or `C` is going to fall under the `F` category.

If you change `grade` to `D` and run the code, you'll see this:

```
F is for Future unknown
```

So whenever you want to add a condition to the `else` statement, since it doesn't come with one, you should use the `else-if`, because then you'll have a way to add a condition and therefore be able to branch your statements even more.

`If` and `else` statements are good for situations with two possible conditions. It's true that you could keep chaining your `if-else` statements to account for several different conditions, but there's a more flexible and cleaner way to achieve the same goal: the `switch` statement.

Resources

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

15

Multiple Branches with the Java `switch` Statement

The `switch` statement can test for a variety of different conditions and respond accordingly. A `switch` statement is the best candidate for cases where you have more than two possible conditions.

Let's translate the final example from the previous lesson into a `switch` statement. (Note that you keep the `char` variable intact because you still need that.)

```
char grade = 'A';
if (grade == 'A') {
   System.out.println("A is for Awesome!");
}else {
   System.out.println("Did you get a B, a C, or
an F?");
}
```

You start by writing the keyword `switch` and finish the code block with curly braces:

```
switch(){
}
```

Inside of the parentheses is where you need to pass the condition—in this case, the variable that will be checked to see if a match was found. So you pass the `grade` variable in:

```
switch(grade){
}
```

Inside the code block is where you isolate all the possible cases and dictate how to respond accordingly. So you say that in cases where `grade` matches `A`, here's what you want to happen:

```
switch(grade){
   Case 'A' :
         //do stuff
     break;
}
```

The colon tells the compiler to run whatever is after the colon until it sees the next keyword, `break`, which indicates to get out of this particular case and go to the next one. And you continue to do the same thing for other cases:

```
case 'B':
   System.out.println("B is for Beautifully done!");
   break;
case 'C':
   System.out.println("C is for Careless");
   break;
```

You can't use the `else` statement as you did with `if` statements; instead, you use the `default` statement:

```
default:
   System.out.println("F is for Future unknown");
```

TIP

If you don't add the `break` keyword in a `switch` statement, you'll get an undesirable result. Also, the `default` statement is optional in `switch` statements.

Run the code, and the output should be the following, assuming the `grade` variable is still `D`:

```
F is for Future unknown
```

Change `grade` to `B` and you should see this:

```
B is for Beautifully done!
```

`Switch` statements make your code much cleaner and easier to maintain.

If you don't add the `break` keyword in a `switch` statement, you'll get an undesirable result, so always make sure to include it.

You can also add more than one statement inside the `switch` statement:

```
 case 'A':
   System.out.println("A is for Awesome!");
   System.out.println("Made mom proud.");
break;
```

Change `grade` to `A` and run the code and the output is this:

```
A is for Awesome!
Made mom proud.
```

Both printouts show in the console.

You can use as many statements as you want.

Optionally, you can even add a code-block body with curly braces, like this:

```
case 'A': {
   System.out.println("A is for Awesome!");
   System.out.println("Made mom proud.");
}
```

Run the code and you'll see the same result:

```
A is for Awesome!
Made mom proud.
```

Exercise

Create a program that will simulate an ATM transaction.

1. Create a `String` variable called `command` that can hold `Withdraw` or `Deposit` commands.
2. Create an `int` variable called `balance`, which holds `1000`.
3. Create another `int` variable called `amount`, which holds `100`.

If the `command` value is `Withdraw`, then subtract `amount` from `balance` and print out this:

```
Your balance used to be 1000 and now is 900.
```

If instead the `command` value is `Deposit`, then print out this:

```
Your balance used to be 1000 and now is 1100.
```

Use the `switch` statement for this exercise.

Solution

There are different ways to solve this, and the way you do it may be different from what you see here, but as long as you get the correct results, then anything is fair game.

You create all the variables and then use the `switch` command to determine which case needs to execute:

```
String command = "Deposit";
int balance = 1000;
int amount = 100;
switch (command) {
   case "Withdraw":
       balance = balance - amount; // or balance
-= amount
       break;
   case "Deposit":
       balance = balance + amount; // or balance
+= amount
       break;
}
System.out.println("Your balance used to be 1000 and
now is " + balance );
```

Notice that there are alternative ways to decrement (subtract) and increment (add) numbers. Each alternative is noted as a comment, which follows two forward slashes (`//`) and is thus ignored by the compiler.

Although the first way that's listed for `Withdraw` (`balance = balance - amount`) works, the second is a more succinct way of expressing the same thing: `balance -= amount`. Which to use is your choice, but most often you'll see the second way written.

When you run this code, you should see this result for the `Withdraw` case:

```
Your balance used to be 1000 and now is 900.
```

And here's the `Deposit` case:

```
Your balance used to be 1000 and now is 1100.
```

Resources

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

16

The `do-while` Loop and the `for` Loop in Java

In addition to the `while` loop, there is a type of loop in Java called the `do-while` loop. To create a `do-while` loop in code, `while ()` is written at the end:

```
do {
}while ();
```

Here's an example of the whole structure:

```
int limit = 10;
int counter = 1;
       do {
           System.out.println("We keep counting...");
           counter++;
       }while (counter < limit);
```

If you run the code, you get this output:

```
We keep counting...
We keep counting...
We keep counting...
...
```

With a `do-while` loop, the statements inside the loop are executed at least once, even if the loop condition is false.

To test this in the code, change < to >, which will make the condition false. When you run the code, the result is this:

```
We keep counting...
```

This tells you that even though the condition evaluated to false, the contents inside the loop ran once.

So if you want to have statements executed once even though the conditional test is false, then use a `do-while` loop. In any other case, a `while` loop will do.

Notice how pretty much anything else about the `do-while` is similar to a `while` loop: You still need to have some sort of a counter or else you end up with an infinity loop, which is bad news.

The `while` and `do-while` loops are great, but the `for` loop is more powerful and lets you do more in Java.

The `for` loop is Java's most complex loop statement. It allows you to repeat a section of a program a fixed number of times, and you get to control the flow.

Here's how to construct a `for` loop:

```
for (int i = 0; i < 10 ; i++) {
  //more code goes here
}
```

The `for` loop has three parts:

- the initialization part (`int i = 0;`), where you are initializing the `int i` variable and giving it `0`;
- the conditional section (`i < 10;`), where you have the conditional test, which resolves to either true or false; and
- the change section (`i++`), where in this case you are incrementing the variable `i` each time the loop runs.

Let's output the `i` value each time the loop runs:

```
for (int i = 0; i < 10 ; i++) {
   System.out.println("Counter is " + i);
}
```

Run the code and here's the output:

```
Counter is 0
Counter is 1
```

…

```
Counter is 0
Counter is 1
```

On the first run, `i` is `0`, which is less than `10`, and then the enclosed statement is run, so you see `"Counter is 0"` in the console. Next, `i` is incremented by `1`, so now `i` is `1`. The comparison continues: `1` is still less than `10`. This time you see `"Counter is 1"` in the console. Then, `i` is incremented by `1` again, so `i` is now `2`, and so on.

This continues until the conditional test is no longer true—e.g., when `i` is `10`, the `for` loop stops because `10` is not less than `10`. This is why the last output you see is `"Counter is 9"`.

The great thing with `for` loops is you can add other conditional statements inside of them to do more complex calculations/computations, like this:

```
for (int i = 0; i < 100 ; i++) {
   if (i % 2 == 0){
       System.out.println(i + " is a multiple of 2");
   }
   System.out.println("Counter is " + i);
}
```

TIP

The counter variable (`int i`) is also called an iterator. An iteration is a single trip through a loop.

You add an `if` statement to capture all of the numbers that are multiples of `2` and print those out. You are now leveraging the modulo, or remainder, operator you learned about in lesson 12.

You can increase `100` to `1000`, or `100000`, or an even higher number, and the code still works. The larger the number you use, the more time it may take (depending on your computer speed), but you'd get the expected results.

With `for` loops, the sky is the limit. And they make it easy for you to quickly do some complex calculations with conditional statements and get results.

Exercise

Use a `for` loop to count backward from 10 to 1. Here's what the output should be:

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1
```

(Hint: Decrement, not increment.)

Solution

```
for (int i = 10; i >= 1; i--) {
    System.out.println(i);
}
```

You must initialize `i` with `10` since that's your starting point. In the conditional section (`i >= 1`), you want to make sure that `i` is greater than or equal to `1` because `1` needs to be included in the countdown. If you remove `>=` and replace it with `>`, the countdown will omit the `1` and end with `2`, which is not what you want for this exercise. The change section (`i--`) is where the backward counting happens, since you decrement each time you loop through and update the iterator `i` by `1`.

Resources

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

17

Arrays in Java

Unlike variables, which can only hold one thing at a time, arrays store a collection of related variables that share the same type. So if you need to store a list of integers, for example, you would use an array.

You can think of an array as a container that holds smaller containers inside it. Each smaller container is called an element. You put values inside of the element. Each element has its associated index—an ID for that element. Array indexes start at `0`, not `1`, so the first element is index `0`, the second is index `1`, etc.

The great thing about arrays is that they have a length—how many elements are available in the array—so you can also specify the length of the array, which gives you a lot of flexibility.

If you wanted to create an array that holds integers, you'd start by writing this:

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

Notice the square brackets (`[]`) after the keyword `int`. This is how you tell the computer that you want to create an array in Java.

The `new` keyword in Java tells the computer to reserve a spot in memory for, in this case, an array.

Now you have an empty array with length `5` (same as having five smaller containers).

Next, you need to add integers inside of this array. To do that, you need to specify which smaller container—or slot—you will be adding the `int` into. This is when the index comes in handy. The index is the ID for each slot in your array, so you use the index to specify the slot to add the `int` to, like this:

```
ages[0] = 10;
```

The first slot (`0`) has a value of `10`, and you keep adding other integers to your array following the same syntax:

```
ages[0] = 10;
ages[1] = 9;
ages[2] = 1;
ages[3] = 8;
ages[4] = 2;
```

Now all of your five slots are filled with integers.

To print out the second slot's content, for example, you write this:

TIP

Array indexes start at `0`, not `1`.

```
System.out.println(ages[1]);
```

Run the code and it prints out this:

```
9
```

What happens if you run this code?

```
System.out.println(ages[5]);
```

When you run this code, you'll get an error like this:

```
Exception in thread "main" java.lang.
ArrayIndexOutOfBoundsException: 5
```

The compiler is telling you that it can't reach index 5 because there is no index 5. Your array only has up to index 4. If you try to reference an index that doesn't exist, you'll get an error.

You can also create an array and initialize it with values right from the start. Here's how you do it:

```
int[] ages = { 10, 9, 1, 8, 2 };
```

The difference here is that you didn't explicitly set the size of the array like in the previous example. Also note that you use curly braces instead of square brackets. Each item inside of the array is separated by a comma.

The way to access each element of the array is the same:

```
System.out.println(ages[1]);
```

This should still print out this:

```
9
```

You can use a `for` loop and loop through an array and retrieve all of its contents:

```
for (int i = 0; i < 5; i++) {
   System.out.println(ages[i]);
}
```

In the `for` loop, you need to make sure you pass the length of the array (number of slots) for the conditional test. In this case, you pass `5` because you know that the array you're looping through has five slots, or the length of the array is `5`.

The variable `i`, the iterator, is going to be updated each time the loop runs, and each time you pass it as the index so you can retrieve, or iterate, through your array.

Run this code and you'll see this:

```
10
9
1
8
2
```

On the first pass, `i` is `0`, and `i` is passed into the array as the index `age[0]`, so you get `10`, and so on.

Notice that you had to hard-code the length of the array in order to add it to the conditional test of the `for` loop.

In this case, it's fine since you know the length of the array, but if you updated the array, you would need to also update the length by hand in the `for` loop.

There's a better, cleaner way to do this. You can access the length of an array by simply writing this:

```
ages.length
```

The period here is called the dot operator, which means that you access a property of the array that will return the length of the array.

Now you can always be sure that you get the size of the array consistently without having to worry about changes or updates to the array, which would change the length of the array.

Run the code and you'll see the same results as before. But if you were to change the array, perhaps to something like the code that follows, you no longer have to update your conditional test in your loop since you are now directly accessing the length property of the array, so it will always pass the correct length.

```
int[] ages = { 10, 9, 1, 8, 2, 90, 67 };
```

You can do all sorts of things inside of your loop. For example, you can only print multiples of 2:

```
if (ages[i] % 2 == 0)
System.out.println(ages[i]);
```

Notice the lack of braces in the `if` statement. If there's only one statement inside the `if` statement, you can optionally remove the curly braces.

Run the code and you should see only multiples of 2 showing from your array:

```
10
8
2
90
```

With the power of arrays combined with loops and branching, you can pretty much do anything in Java!

Exercise

Let's say you have the following array of integers:

```
int[] ages = {20, 30, 100, 45, 10, 33};
```

Write a `for` loop that will do three things:

1. Add all of the age numbers in the array into one variable called `ageSum`
2. Calculate the average age by dividing `ageSum` by the length of the array and assign the result to a new variable called `averageAge`
3. Print out a message that reads "The average age is [pass the `averageAge` here]"

(Hint: Make sure to initialize `ageSum` and `averageAge` variables outside the `for` loop (`int ageSum = 0;` and `int averageAge = 0;`).)

Solution

```
int[] ages = {20, 30, 100, 45, 10, 33};
int ageSum = 0;
int averageAge = 0;
for (int i = 0; i < ages.length ; i++) {
   ageSum = ageSum + ages[i];
   //calculate average
   averageAge = ageSum / ages.length;
}
System.out.println("The average age is: " +
averageAge);
```

The bulk of the work happens inside the `for` loop when you write `ageSum = ageSum + ages[i]`.

Initially, `ageSum` is `0`. On the first pass through, `int i` is also `0`, which means you get the first element of the array, `20`, and add it to `ageSum`, which is still `0`, and 0 + 20 is 20. You then assign `20` to the `ageSum` variable.

On the second pass through, the same thing happens, except now `i` is `1`, which also means you pass it as the index to get the corresponding element (`ages[1]`), `30`, so 30 + 20 (the current value of `ageSum`) is `50`. Then you update the `ageSum` variable with the new result, `50`, and so on.

This computation continues until you've exhausted all numbers in the array: when `i < ages.length` no longer holds to be true.

Notice that there is also another calculation happening at the bottom of the loop:

```
averageAge = ageSum / ages.length;
```

That's where you calculate the average age. Note how the `averageSum` is calculated each time the loop runs, and on each pass the `averageSum` variable is updated accordingly.

Note that you can write the following statement in two different ways, and both are valid options. You will likely see the second option used more often in programs.

1. `ageSum = ageSum + ages[i];`
2. `ageSum += ages[i];`

Resources

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

18

Creating Objects in Java

In programming, objects are the building blocks that can be used to create something more complex. An object encapsulates tasks, or certain functions that can be plugged in to a bigger system.

If you think of a computer program as a group of objects that interact with each other, then you can design a program that's more reliable, easier to understand, and, most importantly, reusable in other projects or programs!

If two people have the same kind of car (make, model, year, etc.) and one car's engine blows up, a qualified mechanic could remove the other car's engine and put it into the first one and it would work. The first car would have reused the second car's engine (object). There's no need to make a new engine from scratch!

In order to create objects, you need a plan, or a blueprint, of how the object is going to behave and what kind of properties the object will have.

In Java, this blueprint is called a class—which is the representation of an object.

If you want a blueprint of a car, then you need to think about what a car has (properties or attributes) and what a car does (behaviors or functions).

Once you know the properties and behaviors of the object, then you can create the class (your blueprint).

TIP

The concept of objects in Java can be a bit difficult to conceptualize at first. Just remember that a Java object is like any physical object in the real world: They have properties and behaviors.

To create a class in Java, you use the `class` keyword, like this:

```
class Car {
}
```

Make sure you are outside of the `main` class.

You need to add attributes or properties of the car. A car is a very complex system, so let's keep it simple: A car has a color, a make, and a model year. So let's add those attributes to your `Car` class:

```
class Car {
   String color;
   String make;
   int modelYear;
}
```

To express behavior, you use methods to express and expose what a car does. Let's add a few methods or behaviors, such as starting, driving, and stopping:

```
public void start() {
}
public void drive() {
}
public void stop() {
}
```

There are a few new keywords here: `public` and `void`. These methods can be accessed by any other class (in other words, they're *public*) and they won't return a value, hence the *void* name.

Let's add printouts in each of the methods:

```
public void start() {
   System.out.println("Starting the car...");
}
public void drive() {
    System.out.println("Car moving now...");
}
public void stop() {
    System.out.println("Stopping for gas...");
}
```

So you have a solid blueprint for your `Car` object. Remember, this is a class, the blueprint of an object—not the actual object yet.

This is how you create, or instantiate, the actual object you can use in Java (using your class, or blueprint). You go up to `main` and write the following:

```
Car myCar = new Car();
```

So, just like you do with variables, you create a variable named `myCar` of type `Car` and assign it to `new Car();`.

All you are doing here is telling the computer to create a space in memory for a `Car` object to live in.

Now you have an actual `Car` type, or `Car` object!

You can use your `myCar` reference to access the behaviors and the attributes of the `Car` object as well as set those attributes!

Run the code and you'll see this:

```
myCar.start();
Starting the car...
```

You set attributes:

```
myCar.color = "red";
System.out.println("My car is " + myCar.color);
```

Run the code and it prints out this:

```
My car is red
```

Recall that there are two main data types in Java: primitive and reference types. You've already worked with primitives, and now you've created your first reference type! The `myCar` object is a reference type because it references the `Car` object.

Remember that a class is not an object. A class merely encapsulates the properties/attributes and behaviors of what the object will have once instantiated.

Think of a class as the blueprint for a house. A constructor will use the house blueprint to build a house. The house blueprint is not a house.

TIP

A class in Java is a blueprint.

Resources

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

19

Class Constructors in Java

If you print out `myCar` in the console, you get what looks like a long string of gibberish:

```
System.out.println(myCar);
....Car@6d06d69c
```

But it's not actually gibberish—it's the location of the object in memory! That's pretty cool, but it's not that helpful.

Let's take this one step further and try to access one of the attributes:

```
System.out.println(myCar.color);
```

Run this and you get a `null`, which simply means "value not set." Since the property `color` hasn't been set to anything, `null` is given to the reference attribute.

Although you could set that attribute/property to something yourself, wouldn't it be better if you could delegate that task to another method?

That's when constructors come into play. A constructor in Java is a method that has the same name as the class, and it constructs your object so that when you instantiate it, the object is not just an empty, useless object.

Let's take a look in code:

```
public Car(String color, String make, int modelYear) {
   this.color = color;
   this.make = make;
   this.modelYear = modelYear;
}
```

Notice that the name of the constructor has to be the same as the class name.

You are passing parameters into your constructor (method) that are used to set up your class attributes to whatever is passed as parameters.

The `this` keyword refers to "this class," and you access "the current" class attributes/properties or fields.

Notice also that the moment you add the constructor, your class instantiation no longer works. Why?

When you instantiate an object, you say this:

```
....= new Car();
```

And what's the name of your constructor? It's `Car`! So this tells you that whenever you instantiate an object, you are actually calling the class's constructor! Why? You're calling it to construct the object.

So why didn't you have any issues previously with just instantiating your `Car` object by just writing the following?

```
Car myCar = new Car();
```

That's because every class you create in Java implicitly has an empty constructor that looks like this:

```
public Car(){
}
```

However, the moment you explicitly create a constructor and pass parameters in, then Java will ignore the empty constructor and use the new one you created. And that's why you have that little complaint showing now in `main`.

To fix it, you just have to pass the arguments into the constructor:

```
Car myCar = new Car("Red", "Toyota", 1989);
```

Run the code again and you should see this:

```
Red
```

This is because the color property of the `Car` object is set to `Red`.

You can extract other properties/attributes as well:

```
System.out.println(myCar.color + " " + myCar.make +
" " + myCar.modelYear);
```

Run the code and this is the output:

```
Red Toyota 1989
```

Constructors are very helpful because they force your objects to be instantiated with some values, which prevents having empty objects. Although you could have classes without constructors, it's best to have constructors when you create classes.

Resources

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

20

Methods: Passing Arguments, Returning Values

Essentially, a method is a code block where you can have statements and write your logic in.

Methods are usually called to execute their content (statements).

For instance, the `main` method is called by the compiler to execute what's inside of it—hence, the `main` method is the entry point to a Java program.

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

The `main` method has a few keywords: `public`, `static`, and `void`.

The `void` keyword in Java means that the method returns nothing, or it doesn't have a `return` type.

Does this mean that methods can have types, too? Sort of. This tells you that you can have methods that return a result to you.

For instance, you can create a method that calculates a sum of two numbers like this:

```
public void sum() {
   System.out.println(10 + 10);
}
```

There are two ways you can call this function in `main`. This is the first way:

```
new FirstJava().sum();
```

You instantiate an object (remember that `main` is actually inside of a class!). This is a simplified way of instantiating an object, especially if you don't need a reference. And then you call the `sum()` method.

Run the code and you'll see this:

```
20
```

Even though this way is valid, wouldn't it be much easier if you called `sum()` inside of `main`?

Yes, it would be much easier and quicker; however, to be able to do that, you need to change a few things.

First note that if you try calling `sum` directly in the `main` method, you get an error:

```
error: non-static method sum() cannot be referenced
from a static context
        sum();
```

As the error tries to explain, `main` is a static method, which means it can only call other static methods inside of it.

In a nutshell, the `main` method is static so that the compiler can call it without the creation of an object or before the creation of an object of the class.

But creating objects is what makes Java an awesome object-oriented programming language! Isn't creating objects a good thing?

Yes, objects are great, but they do take up memory, and you should avoid having to create objects unnecessarily whenever possible, especially if it's for things that don't really need object instantiation, such as to run the `main` method.

To be able to call `sum()` in `main`, `sum` must also be static.

```
public static void sum() {
   System.out.println(10 + 10);
}
```

Comment out the object instantiation:

```
public static void main(String[] args) {
//new FirstJava().sum();
   sum();
}
```

And `sum()` can now be called inside of `static main`! Run the code and it will output `20`.

Let's improve `sum` by adding a few arguments:

```
public static void sum(int a, int b) {
   System.out.println(a + b);
}
sum(10, 10);
```

Next, since you changed `sum` to have arguments, those are now expected when you call `sum` in `main`.

Run this and you'll see the same `20` printed out.

Even though you still get the same results, it's important to note how flexible `sum` is now. It will always take in whatever is passed to it as arguments and calculate the sum and print out the result!

But `sum` is still a void, meaning that it doesn't return a value you could use to do other things with.

For instance, if you wanted to capture the sum and put it into a variable and later multiply by another number coming from another method, it's virtually impossible to do that the way things are now.

To make `sum` even more flexible, you attach a `return` type to it—to signal the compiler that you are expecting a value of a certain type to be returned.

If you want `sum` to return an integer, then you do the following:

```
public static int sum(int a, int b) {
  // System.out.println(a + b);
   return a + b;
}
```

The `return` keyword must be added so that the function actually returns a value: an `int` type because you changed from `void` to `int`.

Now you can pass the `sum`-returned value to a variable:

```
int total = sum(10, 10);
System.out.println("The total is " + total*10);
```

You can even do further calculations and then print out a text, like this:

```
The total is 200
```

Exercise

Look at the following code:

```
   String greet = "Hello, friend! How are you
doing?";
if (greet.contains("Hello ")) {
   System.out.println("Yep!");
}else {
   System.out.println("Nope!");
}
```

Without running this code, what will be printed out in the console: `Yep!` or `Nope!`? Why?

Solution

`Nope!` will be printed out in the console.

Even though `greet` does contain the word `Hello`, remember that in Java a space is also counted as a character! So when you write `Hello ` , that's not the same as `Hello,` or `Hello`.

Resources

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

21

Java Getters and Setters

Returning to the `Car` class from previous lessons, let's add one more field/property/attribute or instance variable:

```
int weight;
public Car(String color, String make, int modelYear,
int weight)
this.weight = weight;
```

Then, let's add this to the constructor:

```
Car myCar = new Car("Red", "Toyota", 1989, 100);
```

And then fix the `Car` object instantiation:

```
System.out.println(myCar.color + " " + myCar.make +
" " + myCar.modelYear + " " + myCar.weight);
```

Run the code and you should see this:

```
Red Toyota 1989 100
```

Even though you are using the constructor to construct your object, you still can directly access the attributes by using the dot operator (`.`) to set the `weight` attribute to `8998`:

```
myCar.weight = 8998;
```

Run the code and the printout will be this:

```
Red Toyota 1989 8998
```

Let's now set the weight to `-10`.

```
myCar.weight = -10;
```

Run the code and the result is still fine:

```
Red Toyota 1989 -10
```

So now you have a red 1989 Toyota that weighs –10. Sure, the code works fine, but there's a problem.

Imagine you are building out a very complex class that is part of a bigger system at a car factory. And then you ship your class off to another programmer, who also has another class that does other cool stuff for the car assembly line. And then another programmer uses your class and directly invokes the `weight` attribute and by mistake sets it to `-10`.

You can see that you'd have a big problem here. How can a car weigh –10?

Even though you trust that other programmers are intelligent people who would never intentionally set `weight` to `-10`, you must protect your classes from such catastrophic mistakes.

TIP

Always protect the attributes of your classes by making them private.

You should make your class-attribute variables private so that no one can directly access them and set them to undesired values. So let's do that:

```
private String color;
private String make;
private int modelYear;
private int weight;
```

Now you have other problems: You can't access any of these from `main`. While this is a problem in that your code now won't compile, it's actually a good thing because you're now forced to set up your class properties in a more secure way by using getters and setters!

```
public String getColor() {
   return color;
}
public void setColor(String color) { this.color =
color; }
```

A setter method sets up a class attribute to either have an initial value or check for invalid values (such as setting `weight` to `-10`). A getter method exposes those set attributes to other classes so that those other classes can only access attributes through getters. It's a mechanism of protecting a class's internal state and attributes.

Now the only way you can access and tamper with the attributes is through getters and setters.

You set the weight and the color attributes using setters:

```
myCar.setWeight(8998);
myCar.setColor("Blue");
```

Instead of accessing the `weight` attribute directly, which you can no longer do because the attribute is now private, you use a getter method, `getWeight()`, to retrieve the `weight` value:

```
myCar.getWeight();
```

Run the code:

```
System.out.println(myCar.getColor() + " " + myCar.
getMake() + " " + myCar.getModelYear() +
" " + myCar.getWeight());
```

And the result is this:

```
Blue Toyota 1989 8998
```

You really haven't solved the problem of setting undesirable values to your attributes, but this is one step forward. Now you can easily add an `if` statement to check the values before you set them up. So if you detect that `weight` is less than or equal to `0`, set the weight attributed to a default value of `1000`:

```
public void setWeight(int weight) {
   int defWeight = 1000;
   if (weight <= 0) {
        this.weight = defWeight;
   }else {
       this.weight = weight;
   }
}
```

In object-oriented programming, hiding your data—in this case, your class attributes—so they are protected from being directly modified by other objects is called encapsulation.

Notice, too, that while your attributes must be private, your getters and setters are public, because it's through getters and setters that any other object can interact with the class attributes.

The bottom line is to mark attributes/properties/fields or instance variables `private` and getters and setters `public` so that your classes are secure and protected.

Resources

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

22

Using the `String` Class as a Reference Type

In lesson 9, you learned that Java has two major category types: primitive and reference types.

Java classes fall under reference types because unlike primitives, they are not predefined data types—meaning that they hold a reference to a dynamically created object in the memory.

A string is a reference type, which means that it's a class that you can instantiate into an object!

```
String name = "Lucy";
System.out.println(name);
```

You expect to see `Lucy` in the console when you run this code.

You can write the same statement this way:

```
String name = new String("Lucy");
System.out.println(name);
```

Run the code and you get the same result: `Lucy` is printed.

This tells you that, in fact, `String` is a class that you can instantiate into an object!

Notice that `String` starts with an uppercase *S*; classes in Java always start with an uppercase letter!

Why is it, then, that you don't see the second syntax used often when creating string types?

The Java Virtual Machine (JVM) does this instantiation for you, so you don't have to use the `new` keyword to instantiate a string. Remember, each time you instantiate an object, you are using up memory. The JVM knows this and does the instantiation in the background for you whenever you need to create a string type.

If `String` is a class, that means it must have methods (behaviors) and instance variables (fields/properties/attributes).

Type `name.` and you see all the different methods you can use!

Let's choose this one:

```
System.out.println(name.toUpperCase());
```

Run the code and you see this in the console:

```
LUCY
```

Let's test the `contains()` method. You can tell that it requires a parameter sequence to search for and that it returns a `boolean` type (`true` or `false`).

Because you know that the function returns a `boolean`, you can use that in an `if` statement to search for `y` in the string. If it's found, then you print that message:

```
if (name.contains("y"))
System.out.println("Yes, there's a y in Lucy");
```

Run the code and you'll see this message:

```
Yes, there's a y in Lucy
```

You can even check the length of the string, which is how many characters are in the string.

You can write this:

```
System.out.println(name.length());
```

Run the code and you'll see this:

```
4
```

Now add a space after `y`:

```
String name = "Lucy ";
```

And run the code again:

```
5
```

Length returns `5` now. This is because an empty space in Java is also read as a character.

You could also use the concatenate method, which appends text in front of `Lucy`:

```
System.out.println(name.concat("'s birthday"));
```

Run the code and you should see this:

```
Lucy's birthday
```

You could even use the `charAt()` method, which allows you to get a character at a certain index in the string, like this:

```
name.charAt(0);
```

This prints out `L` because that's the first letter in `Lucy`.

You can also print out all of the letters separately in a `for` loop, like this:

```
for (int i = 0; i < name.length(); i++) {
   System.out.println(name.charAt(i));
}
```

Here's the output:

```
L
u
c
y
```

As you can see, you can do all sorts of things with a string object.

`String` is just one of the thousands of classes that come prepacked in Java for you to use in your programs, which makes Java a very powerful programming language.

Exercise

What does the `super` keyword refer to in Java inheritance?

a the subclass object **b** the object class **c** the parent class

Solution

C the parent class

Whenever you want to reference the parent class from a subclass, you use the `super` keyword. By using `super`, you can access the parent/superclass methods and any other public attributes.

In this case, answer B is wrong. When it comes to inheritance in Java, you usually have a class hierarchy where the direct, immediate parent class can be any class other than the object class.

The super object class is the parent class of all Java classes by default. In the Java inheritance hierarchy, the `super` keyword will always refer to the immediate parent class, not the object class itself.

Resources

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

23

Java Inheritance: Overriding Parent Methods

TABLE OF CONTENTS

In Java, you can reuse code that's already been written so that you can have classes inheriting from other classes.

Returning to the `Car` class from previous lessons, you can remove, or abstract out, the attributes and behaviors into another class called `Vehicle`, for example, like this:

```
public class Vehicle {
    private String color;
    private String make;
    private int modelYear;
    private int weight;
    public Vehicle(String color, String make, int
modelYear, int weight) {
        this.color = color;
        this.make = make;
        this.modelYear = modelYear;
        this.weight = weight;
    }
    public void start() {
        System.out.println("Starting the car...");
    }
    public void drive() {
        System.out.println("Car moving...");
    }
    public void stop() {
        System.out.println("Stopping for gas...");
    }
    public String getColor() {
        return color;
    }
```

```
    public void setColor(String color) {
        this.color = color;
    }
    public String getMake() {
        return make;
    }
    public void setMake(String make) {
        this.make = make;
    }
    public int getModelYear() {
        return modelYear;
    }
    public void setModelYear(int modelYear) {
        this.modelYear = modelYear;
    }
    public int getWeight() {
        return weight;
    }
    public void setWeight(int weight) {
        int defWeight = 1000;
        if (weight <= 0){
            this.weight = defWeight;
        }else {
            this.weight = weight;
        }
    }
}
```

Then, you can create a new `Car` class that will inherit from the `Vehicle` class. To express inheritance in Java, you use the `extends` keyword, like this:

```
public class Car extends Vehicle {
}
```

Now `Car` inherits from the `Vehicle` class. So `Car` is now a subclass and `Vehicle` is the superclass—the parent class.

Let's go to the `main` method and test this out.

Run the code and you'll see this:

```
Car myCar = new Car();
myCar.setMake("Toyota");
myCar.setModelYear(1978);
myCar.setWeight(8998);
myCar.setColor("Blue");
Blue Toyota 1978 8998
```

You can see that inheritance is working here. Notice that the `Car` class has nothing written inside of its body, yet you are able to set the attributes to, and call on, the object's methods!

Now you could simply create any implementation of a vehicle without having to copy and paste code. For instance, if you wanted to create a `Truck` class, you can now do so by inheriting `Truck` from the `Vehicle` superclass, like this:

```
public class Truck extends Vehicle {
}
```

Back in `main`, you instantiate a `Truck`:

```
Truck myTruck = new Truck();
myTruck.setMake("Ford");
myTruck.setModelYear(2019);
myTruck.setWeight(998);
myTruck.setColor("Red");
System.out.println(myTruck.getColor() + " " + myTruck.
getMake() + " " + myTruck.getModelYear() +
" " + myTruck.getWeight());
```

Run the code to get this:

```
Red Ford 2019 998
```

You can potentially create any other kind of vehicles you want by inheriting from the superclass vehicle! As you can see, inheritance is a very powerful thing: It gives you more flexibility and code reusability.

Although each subclass inherits from the superclass, each is its own entity—meaning that each can have its own attributes and behaviors to make it unique.

This can be achieved by simply adding attributes and behaviors to each subclass. Let's say that the way you'd start a truck is unique to a truck only. You can override the `start()` method from the superclass, like this:

```
@Override
public void start() {
   System.out.println("Vrooom Vroom like a Truck");
}
```

Notice how the IDE added the `@Override` annotation there. This just tells the compiler that instead of using the superclass `start` method, now, for `Truck`, always use this start method to start a truck—in other words, override the behavior!

Let's test it out. Back in `main` is this:

```
myTruck.start();
```

Run the code and you see:

```
Vrooom Vroom like a Truck
```

What if you wanted to add a unique attribute to a truck, such as horsepower? You can add this new attribute to the `Truck` class along with the getters and setters:

```
private int horsePower;
public int getHorsePower() {
   return horsePower;
}
public void setHorsePower(int horsePower) {
   this.horsePower = horsePower;
}
```

Now, back in `main`, you can use the getters and setters to set up your new attribute.

Run the code and here's the output:

```
Red Ford 2019 998 HP: 12000
```

Inheritance in Java is a method of code reuse, which means that you can create a hierarchy of classes that inherit behaviors and attributes of other classes, making it possible to create complex programs that can be reused and modified as needed.

To design an inheritance tree, follow these steps:

1. Look for objects that have common attributes and behaviors.
2. Design a class that represents the common state and behavior.
3. Decide if a subclass needs behaviors (method implementations) that are specific to that particular subclass type.

Resources

Schildt, *Java*, section 7.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

24

Java Inheritance: Invoking Parent Methods

In the previous lesson, you saw how to override behaviors of the parent class (superclass) so that the subclass implements its own version of that behavior. There may be times when you'd still want to invoke the superclass's method inside of a subclass.

This can be done by using the `super` keyword:

```
public void showSuperBehavior() {
   super.drive();
}
```

So `super.drive` is going to call the `Vehicle` (superclass) `drive` method inside of the `Truck` class.

Now when you call `showSuperBehavior()`, you'll see this:

```
Moving now...
```

Let's look at the superclass. You'll see that the constructor had been temporarily commented out. Let's uncomment it out and see what happens.

Notice that you now have an issue in your `Truck` class. The compiler wants you to create a constructor that matches the super's constructor. When you abide, here's what you get:

```
public Truck(String color, String make, int modelYear,
int weight) {
   super(color, make, modelYear, weight);
}
```

You now have a `Truck` constructor that invokes, first, the super (parent) constructor. This means that constructors are not inherited in Java. In reality, whenever you inherit a superclass with a non-argument constructor, the JVM will still call `super` and call the non-argument constructor.

Since `Truck` has a field that may need to be included in the "construction" process of a truck, you can also set up the `horsepower` attribute after the `super()` constructor is called:

```
public Truck(String color, String make, int modelYear,
int weight, int horsePower) {
   super(color, make, modelYear, weight);
   this.horsePower = horsePower;
}
```

And back in `main`, where you instantiated a `Truck` object, you can do this:

```
Truck myTruck = new Truck("Black", "Nissan", 1999,
8900, 9999);
```

If you run this code as it is, you'll see different results since you reset all of those properties at the bottom. To test things out, comment out the resetting of your truck object so you can only rely on the constructor.

But before you do that, you need to pass the super constructor to the `Car` class as well, even though you are not using it. This will prevent you from getting errors.

Now you are ready to run your code.

```
Black Nissan 1999 8900 HP: 9999
```

And it works!

Exercise

Create a method called `showFullName()` that takes in two parameters: `String firstName, String lastName`. In particular, create a method that will return a string that says this:

```
"Here's my full name [firstName] [lastName]."
```

When the function is called in `main`, it must look like this:

```
System.out.println( showFullName("Gina",
"Alexander" ) );
```

And here's the output:

```
"Here's my full name Gina Alexander."
```

Solution

```
public static String showFullName(String firstName,
String lastName) {
    return "Here's my full name " + firstName + " " +
lastName;
}
```

Notice the `String` type right before the function name. The `String` before `showFullName` tells the compiler that the methods must return something—in this case, a string type.

Also notice the `return` keyword inside the function. Since you say that the function expects to return something, you also must use the `return` keyword and return a string. Additionally, the function also has a few parameters, of type `String`.

Resources

Schildt, *Java*, section 7.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

Quiz

1 What's the problem with the `while` loop in the following piece of code? How do you fix the issue?

```
 int upperLimit = 10;
int counter = 0;
while (counter < upperLimit) {
    System.out.println("Not there yet…");
}
```

2 What's the difference between primitive and reference types in Java?

3 Why are constructors important?

4 True or false? If the class name is `Person`, then the constructor (or constructors) must also have the same name as the class.

5 What's wrong with the following function? Why won't it work?

```
  public void showName(String name) {
    System.out.println("Name is "+name);
   return name;
}
```

6 What is the purpose of an access modifier, such as `private`?

Quiz Answers

1. The `while` loop will continue to run for a long time—potentially forever, or until you terminate the program. It's an infinite loop. To fix the problem, you need to increment the counter (`counter++`, or `counter = counter + 1`) right after the `print` statement.

2. Primitive types are not dynamic types, meaning that they are predefined by the language and are named by a reserved keyword, such as `char`, `boolean`, `int`, or `double`. These are the basic types of data. Reference types are any instantiable class, such as `String`, or any custom object you create.

3. Constructors, as the name implies, construct the object when you instantiate it so that the object's instance variables have some initial values.

4. True. Constructors in Java must have the same name as the class name.

5. The `showName` function is a void function, which means that it returns nothing. Therefore, having the `return name;` statement will prompt a compilation error since void methods are not allowed to return anything.

6. In Java, it's recommended that classes protect their internal state (instance variables and behaviors) at all costs. The idea is that no other class should directly access another class's internal state. Classes can interact with each other via getters and setters. The class member variables (instance variables) should be set to private.

25

The Java Class Library

In the Java Class Library documentation, you can see all of the Java classes available to you as a programmer. Keep in mind the same concepts you've learned about classes and inheritance; they were also used in the creation of the Java Class Library.

Head over to this address:

https://docs.oracle.com/javase/8/docs/api/

Related classes are organized in packages. There are hundreds, if not thousands, of packages. There's virtually every class you'd need to get the job done, so you don't have to write it yourself. That's the beauty of Java: You have a library of usable classes you can plug into your own classes and have them work for you!

The `java.lang` package is the package with all of the fundamental Java classes that even the most simplistic Java program will use.

Let's find something familiar first: the `String` class. When you click on `String`, you'll see more information about the `String` class, including constructors and methods.

Note that `String` inherits from the `Object` class. When you click on the `Object` class, you see that it doesn't have a superclass from which it inherits. The description of this class reveals that it's the ultimate Java class, meaning that everything in Java inherits from `Object`!

Everything in Java, one way or another, inherits from the superclass `Object`!

Spend some time exploring the Java Class Library—there's a lot to explore. Keep in mind that you don't have to memorize any of the classes. That's not the point. The point is for you to realize how easy it is to leverage classes that already come with Java.

The Java Class Library is extensive and is the best resource to learn more about what you can do with Java. You can look up classes you can use in your programs so that you don't have to reinvent the wheel. This is an important concept in programming: If there's already a tool that will solve a problem or help you solve a problem, then use that tool. Don't reinvent the wheel when you don't need to.

Resources

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

26

Java `ArrayList` and Object-Oriented Pros and Cons

In programming, you constantly store, retrieve, and manipulate data and then store it again for further processing.

So far, you've learned three useful places to keep information:

1. basic data types (`int` and `char`)
2. arrays
3. string objects

There are far more places to store information, because any Java class can hold data. The most used is `ArrayList`, a data structure that holds objects of the same class.

The difference between arrays and the `ArrayList` is that the `ArrayList` can grow or shrink in size at any time.

Let's return to the Java Class Library to learn more about the `ArrayList`:

https://docs.oracle.com/javase/8/docs/api/

The `ArrayList` belongs to the `java.util` package, which is also one of the most useful in the Java Class Library.

To use the `ArrayList` class in your program, the first thing you'll need to do is import the `java.util` package, but specifically the `ArrayList` class.

Since you are using a very sophisticated IDE, you probably won't need to manually import the package. All you have to do is start typing `ArrayList` and then hit Enter.

QUESTIONS ABOUT IMPORTS

1. Why do you need to explicitly import java.util and not java.lang?

You don't have to explicitly import **java.lang** because the **.lang** package has all the basic classes that are automatically needed for the most basic Java program to be run. Hence, it's already included, so you don't need to do it yourself.

2. Why aren't all Java packages automatically included when you create a program so that you don't have to import any packages?

It's always better and more efficient to only have what you need. If you are writing a program that doesn't use the **ArrayList**, then it's redundant to have the **.util** and hundreds of other packages, each of which may have thousands of classes.

Then, you'll see at the top, outside of the class, this import:

```
import java.util.
ArrayList;
```

This means that you are ready to start using the `ArrayList` class.

The syntax may take some getting used to:

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

What you're saying here is this:

> *Create an `ArrayList` called `names` that will hold strings.*

The type that the `ArrayList` will hold is always inside of the angular brackets (`<>`).

How would you instantiate an `ArrayList` to hold integers?

It turns out that, for each primitive type, there's an equivalent class type you can use in situations like this.

So, the primitive `int` has its wrapper class `Integer` that you can wrap your `int` primitive into and add to an `ArrayList`, for example, like this:

```
ArrayList<Integer> ages = new ArrayList<>();
```

Now you can use either one of the two `ArrayList` instances you created to insert things into them.

Because an `ArrayList` is an object, you can use the `add` method to insert data, like this:

```
names.add(0, "Carla");
names.add("Lucia");
```

In the first statement, you specify the index where `Carla` is inserted, and in the next statement, you just inserted `Lucia`, and the `ArrayList` knows how to automatically assign the appropriate indexes for each string you add.

Add a printout and run the code:

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

And you'll see this:

```
[Carla, Lucia]
```

This tells you that data stored in `ArrayList` is actually internally stored in a simple array!

Now how do you get a specific string from the `ArrayList`?

You use the `get()` method and pass the index `0` for the first item in the `ArrayList`:

```
System.out.println(names.get(0));
```

Run the code and you'll see this:

```
Carla
```

You can also loop through the `ArrayList` using a `for` loop. To get the size of the `ArrayList`, you just invoke the `size()` method:

```
for (int i = 0; i < names.size(); i++) {
System.out.println(names.get(i));
}
```

Run the code and you'll see this:

```
Carla
Lucia
```

You can also remove an element from the `ArrayList` by passing the index or, in the following case, the actual object:

```
names.remove("Carla");
```

Run the code and you'll see only `Lucia` printed out.

You can also remove by passing the index:

```
names.remove(0);
```

This code gives the same result.

In general, object-oriented programming (OOP) has many advantages, such as these:

- code reusability
- code extensibility
- code reliability

These advantages are part of the reason why OOP is one of the most useful ways to build software because it forces programmers to think about software as a collection of different pieces interacting harmoniously with each other.

There are, however, some disadvantages associated with OOP:

- steep learning curve
- larger program sizes
- not suitable for all types of programs

Ultimately, OOP is a tool that helps you organize your code and build complex software by breaking down the complexity into smaller pieces (objects).

Resources

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

Tutorials Point, https://www.tutorialspoint.com/java/.

27

Java Swing: Create a Simple User Interface

Java is a cross-platform language, which means that it can run on many different operating systems, including Mac and Windows. Its graphical user software must be flexible so that, for example, if you write a program that displays a window on the screen, that window will look native to each particular operating system.

In Java, to create programs with user interfaces, you use Java Swing and a set of classes called the Abstract Window Toolkit, which enables you to create a user interface that can receive input from the user!

It's very easy to create user interfaces in Java. All you do is create objects, set their variables, and call their methods. So you'll be using exactly the same techniques you've learned throughout this course with OOP.

To create any type of user interface in Java, you'll need these two objects:

- components
- containers

A component is anything you put in a container. So it could be a button, a text field (where users can enter text), etc.

A container is the canvas, where you can paint anything you want. It's what you use to add other components into.

Let's create a simple window that has `Hello World Swing!` for a title.

Create a separate class called `HelloFrame` that extends `JFrame`, which is a container that can hold components.

First, you set up the frame. You pass a string to your super constructor; this string is the title of the frame. You then set up the size of your frame by passing the width and height values. Next, you make sure that when the user clicks on the Exit button, the program exits. Finally, you must set the frame to be visible:

```
public class HelloFrame extends JFrame {
   public HelloFrame() {
       super("Hello World Swing!"); // or setTitle
       setSize(500, 300);
      // setTitle("Hello World");
       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       setVisible(true);
   }
}
```

Now that you have created your frame class, it's time to instantiate it in `main`, like this:

```
public class HelloSwing {
   public static void main(String[] args) {
       new HelloFrame();
   }
}
```

Since you don't need a reference to your frame for now, all you do is instantiate by writing the new keyword and the name of the class, `HelloFrame`.

Run the code and you should see a window with a title that reads `Hello World Swing!`.

It's easy to add a label to your frame. You create a label by invoking the `JLabel` class, and back in your `HelloFrame` class, you instantiate the label:

```
JLabel label = new JLabel("What's your name?", JLabel.
CENTER);
```

You passed the text you want to show in your label, as well as an optional alignment for the label, `JLabel.CENTER`, which aligns the label's contents to be shown in the middle of the frame.

Now you need to add it to your canvas (frame). Since you are inside of a `JFrame` subclass, you have direct access to all methods from the `JFrame` class, such as the `add()` method, which allows you to pass the label into the frame:

```
add(label);
```

Run the code and you'll see a window pop up with a label that says `What's your name?`.

Let's say you want to have a label and text field side by side. To do that, you can use one of many predefined layouts called the `GridLayout` class, which is a layout manager that lays out a container's components in a rectangular grid. The container is divided into equal-sized rectangles, and one component is placed in each rectangle.

First, you'll need to instantiate the layout and pass `0` for rows and `1` for columns.

```
GridLayout gridLayout = new GridLayout(0, 1);
setLayout(gridLayout);
```

Next, you have to set this layout on top of your frame by calling the `setLayout()` method (from the `JFrame` class) and pass the `gridLayout` you just instantiated:

```
setLayout(gridLayout);
```

Visually, you have the canvas (`JFrame` container) and then a layout (`GridLayout`), which just helps you lay out your components better. Then, you add any components inside of this layout (`GridLayout`), which lays its contents in a row.

Next, you create the components to be inserted into the `GridLayout`. First, you create a label and add a string to it. Also, you want it to be in the middle of the frame, so you pass `JLabel.CENTER`. Then, you create a text field by instantiating a `JTextField` object:

```
JLabel label = new JLabel("What's your name?", JLabel.
CENTER);
JTextField nameField = new JTextField();
```

And you add those onto your layout:

```
add(label);
add(nameField);
```

Run the code to test your layout. It should work and look great.

How would you instantiate a new button with text that says `Show!` and then add it to the layout?

To add a button is also a very simple process.

Instantiate a `JButton` object and pass the title of the button through the `JButton` constructor:

```
JButton showButton = new JButton("Show!");
```

Then, add it to the layout:

```
add(showButton);
```

Notice that the order in which you add items to layouts is very important. If you move `add(showButton)` at the top of the first call to `add`, then the button will show to the left, which is not what you want.

Run your code and your button should show up.

Resources

Haase and Guy, *Filthy Rich Clients*.

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

28

Adding Buttons and Event Listeners

The button you created in the previous lesson does nothing right now when clicked. Let's add an `ActionListener` to the button so that it can listen for events like when users click on the button.

In Java Swing, whenever you want to have components react to a click event, for example, you need to attach those components to an `ActionListener` and then pass the `ActionListener` object. The `ActionListener` object has a method called `actionPerformed`, which is called the moment an event happens to the component that's listening for events. It's inside of the `actionPerformed` method you write the logic to be executed when an action is performed—for example, the click of a button. In this case, you simply print out a string.

```
showButton.addActionListener(new ActionListener() {
   @Override
   public void actionPerformed(ActionEvent
actionEvent) {
       System.out.println("Show!");
   }
});
```

When you run this code and then click the button, this prints out:

```
Show!
```

What you really want to do is retrieve what's in the text field and show that. To accomplish this, you can call the `getText()` method, which returns the text that's inside the text field:

```
String nameText = nameField.getText();
```

You put the contents into a `String` variable for easy handling.

Let's print out the contents of the `Text` field when you click the button.

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

If you type in `Lucia` and click the button, you should see `Lucia` printed out!

This is great, but you only see the results as text in the console. Let's add another component so you can actually see the result in the user interface.

First, you create a new label to hold the results. Notice how the label has an empty string. This is intentional because you want the label to dynamically show the results when the button is clicked.

```
JLabel result = new JLabel("", JLabel.CENTER);
add(result);
```

Next, inside of the `ActionListener`, pass the contents of the text field onto your result label, like this:

```
result.setText(nameText);
```

Run and test! The code should work.

You can also clear the text field when you click the button by adding this code inside the `actionPerformed` method:

```
nameField.setText("");
```

Run and give it a try. Now when you click the button, you also clear the field.

Resources

Haase and Guy, *Filthy Rich Clients*.
Schildt, *Java*.
Sierra and Bates, *Head First Java*.

29

Java Swing: `BorderLayout`

When creating user interfaces (UI) in Java Swing, laying out what you want your users to see is very important. This is why Java Swing provides a few layout-manager classes to manage and give you a UI design framework to work with.

As you learned, `GridLayout` lays out a container's components in a rectangular grid and organizes everything into rows and columns.

Depending on how you want your components to show on the screen, you might want to be able to, for example, lay out all of your components in different regions of the frame. For example, you might want to have a label for a title at the top of the frame, on the left a button, to the right another component, in the middle maybe a button, and at the bottom something else. For such cases, you'd want to use the `BorderLayout` layout-manager class.

To use the `BorderLayout` class, first you create a new, separate class within your project and call it `HelloBorderLayout` and inherit from the `JFrame` class. Add the corresponding constructor as well:

```
public class HelloBorderLayout extends JFrame {
  public HelloBorderLayout() {
       }
}
```

Go back to the `main` method and make sure you are now instantiating this new class—that way, when you run the code, this class is called.

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

Back to the `HelloBorderLayout` class, you set up the frame inside of the constructor and give it a size and make sure it's visible.

You first set the title by calling the `setTitle()` method and passing a string. You could have called `super` and then passed the string as well, but this is another way of doing the same thing. Then, you set the size of the frame: width `500` and height `300`.

```
setTitle("BorderLayout");
setSize(500, 300);
```

Next, you make sure that when users click on the frame's Exit button, the application exits:

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Next, you make the frame visible by passing the `boolean` `true` to the `setVisible()` method:

```
setVisible(true); //important - otherwise you won't
see the frame
```

Run this and you should see a frame with the specified title pop up on the screen.

The frame (represented by the `JFrame` class) is a high-level container, which means that other containers you might want to add are going to be on top of `JFrame`.

`JFrame` always comes with another container (subcontainer) called `ContentPane`. You must always add any components into the content pane, as opposed to adding directly to the `JFrame`, especially if you have more than one component.

In order to get to the content pane, you use the `getContentPane()` method on the root frame (`JFrame`), and you can put this container (which is returned by the `getContentPane()` method) into a reference variable of type `Container` for easy handling:

```
ContainergetContentPane();
Container container = getContentPane();
```

Now you can start adding components, such as buttons. Let's add a few of them:

```
container.add(new Button("Save"));
container.add(new Button("Delete"));
```

Run the code and you'll see that the only button showing is the `Delete` button. Where did the other one go?

Both of them are inside of your container. But since you didn't specify where these buttons should be located, they all default to the center of the screen, and they are all piled on top of each other. Since the `Delete` button is the one on top, it's the only visible one.

To solve this issue, you must pass a layout manager that can specify where each button will be located on the screen. Let's pass the `BorderLayout` manager:

```
container.add(BorderLayout.NORTH, new Button("Save"));
container.add(BorderLayout.SOUTH, new
Button("Delete"));
```

You pass `BorderLayout.NORTH` to specify the region you want the `Save` button to be located on the screen and `BorderLayout.SOUTH` for the `Delete` button.

Notice how the order with which you pass the border layout calls; it must be the first thing you pass to the `container.add()` method.

The `BorderLayout` layout manager defines five regions: east, west, north, south, and center.

Let's fill the frame with more buttons:

```
container.add(BorderLayout.NORTH, new Button("Save"));
container.add(BorderLayout.SOUTH, new
Button("Delete"));
container.add(BorderLayout.EAST, new Button("Print"));
container.add(BorderLayout.WEST, new Button("Edit"));
container.add(BorderLayout.CENTER, new
Button("Copy"));
```

Run the code once again and you'll see all of the regions filled with buttons.

One thing to notice here is that components in the center get whatever space is left over based on the frame dimensions. Components in the east and west get their preferred width; components in the north and south get their preferred height.

Resources

Haase and Guy, *Filthy Rich Clients*.

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

30

Java Swing: `FlowLayout`

The `FlowLayout` layout manager cares about the flow of the components it holds. This means that as you add components into this layout, the components will be aligned horizontally (in a row) if the frame length is small, and then `FlowLayout` will create more rows to accommodate the placement of the components. Let's take a look in code.

First, you are going to separate `FlowLayout` into a new class. So create a new class and name it `HelloFlowLayout`. Make sure to set up the boilerplate code so that the frame is visible and ready for you to start working with the `FlowLayout` layout manager.

```
public class HelloFlowLayout extends JFrame {
   public HelloFlowLayout() {
        setTitle("FlowLayout");
        setSize(500, 300);
        //Write FlowLayout code here
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true); //important - otherwise you
won't see the frame
   }
}
setLayout(new FlowLayout());
```

Since you are inside of a `JFrame` (root frame), you can directly invoke the `setLayout` method and set it to a `FlowLayout` layout manager. You can invoke the `add()` method to start adding components to your `FlowLayout`, and not to `JFrame` directly since you just inserted a new layout on top of `JFrame` when you called `setLayout(new FlowLayout())`.

Make sure that you are now calling `HelloFlowLayout` in `main()`; that way, when you run this project, you get the contents you are setting up in the `HelloFlowLayout` `JFrame` class.

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

Back to the `HelloFlowLayout` class, let's add a few buttons onto the `FlowLayout`:

```
add(new Button("Save"));
add(new Button("Delete"));
add(new Button("Print"));
add(new Button("Edit"));
add(new Button("Copy"));
```

When you run the code, you see a row of buttons anchored at the center of the screen. Next, resize the window frame. When you do that, notice that everything is rearranged into more rows to accommodate the changing size of the frame. This is the way a `FlowLayout` works: It's a fluid and dynamic layout that accommodates its component's preferred size but also rearranges the layout as the size of the root frame changes.

The other property of `FlowLayout` is you can pass an argument to the constructor that will anchor where the row is positioned on the frame—center, leading, trailing, left, right. Let's change the `FlowLayout` constructor:

```
setLayout(new FlowLayout(FlowLayout.LEFT));
```

Pass the left property of the `FlowLayout` to force the row to be anchored on the left of the frame.

Run the code and you'll see that all contents have shifted to the left.

Resources

Haase and Guy, *Filthy Rich Clients*.

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

31

Java Swing: BoxLayout

Let's create another class and name it `HelloBoxLayout` so you can see how the `BoxLayout` manager works:

```
public class HelloBoxLayout extends JFrame {
   public HelloBoxLayout() {
        setTitle("BoxLayout");
       setSize(500, 300);
```

Make sure you set the title to `"BoxLayout"` and set the size; also make sure you have the `setVisible` function receiving a `boolean` of `true`:

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setVisible(true); //important - otherwise you won't
see the frame
   }
}
```

Don't forget to instantiate this class (`HelloBoxLayout`) in `main` so you see the contents of this class:

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

Run to make sure you can see a blank window entitled `BoxLayout`.

A `BoxLayout` is a layout manager that either stacks components on top of each other or places them in a row. Let's create a `BoxLayout` so you can see how all of this works.

First, you are going to create a container to add to the `BoxLayout`. This container is called the `JPanel`. A `JPanel` is what's called another lightweight container, which means that you can use it to quickly create a simple container when you need it.

First, instantiate a `JPanel`:

```
JPanel panel = new JPanel();
```

Right after the instantiation, you are going to add this to your `JFrame` via the content pane by invoking the `getContentPane()`, method which returns a container to which you can pass the panel you've just created by invoking the `add` method:

```
getContentPane().add(panel);
```

As you can see, containers can hold other containers. That's the flexibility you have when you use Java Swing: You get to construct and design your own user interfaces with preexisting containers and components!

Next, you use the panel, since this is the container you are now using to add anything else to it, and invoke the `setLayout` method so that you can set a layout on top of the container:

```
panel.setLayout(new BoxLayout(panel, BoxLayout.Y_
AXIS));
```

Inside of the `setLayout` method, you pass the `BoxLayout` object. The `BoxLayout` constructor needs a few arguments: the component it's laying out (in this case, your panel) and which axis to use (you use `Y_AXIS` for vertical stack).

Now that you have containers set up properly, you can start adding components to your panel:

```
panel.add(new Button("Save"));
panel.add(new Button("Delete"));
panel.add(new Button("Print"));
panel.add(new Button("Edit"));
panel.add(new Button("Copy"));
```

Run the code and you'll see vertically stacked buttons.

Then, change the axis to `X_AXIS`:

```
panel.setLayout(new BoxLayout(panel, BoxLayout.X_AXIS));
```

Run it again and you'll see horizontally stacked buttons.

Resources

Haase and Guy, *Filthy Rich Clients*.
Schildt, *Java*.
Sierra and Bates, *Head First Java*.

32

Java Swing: Build a Fun Graphical User Interface

You are now going to build a fun Java Swing graphical user interface (GUI) application. You'll draw a circle in the middle of a frame, and each time you click anywhere on the frame, the circle will change color randomly.

First, you'll create a new class called `FunColor` that extends the `JFrame` class. Add a constructor and set up the boilerplate code:

```
public class FunColor extends JFrame {
   public FunColor() {
       setTitle("FunColor");
       setSize(500, 300);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setVisible(true); //important - otherwise you won't
see the frame
        }
}
```

Next, you'll create another class called `DrawCircle` in the same file where `FunColor` was created. Make sure this new class is outside the last brace at the end of the file. Also, this class extends `JPanel` (not `JFrame`).

```
class DrawCircle extends JPanel {
@Override
protected void paintComponent(Graphics graphics) {
          //we draw here
     }
}
```

`DrawCircle` is a subclass of `JPanel` and not `JFrame` because you are going to be drawing something (a circle), and `JPanel` has a `paintComponent`, which makes it possible for you to actually draw ovals, rectangles, and many other shapes.

Next, you override the `paintComponent` method:

```
class DrawCircle extends JPanel {
@Override
protected void paintComponent(Graphics graphics) {
          //we draw here
         graphics.setColor(Color.pink);
        graphics.fillOval(70, 70, 200, 200);
```

Notice how the `paintComponent` method has the `Graphics` reference as a parameter. The `Graphics` class enables you to draw shapes. It also enables you to color those shapes. It's a very handy class that's readily available for you to use!

To draw something using the `Graphics` reference, you use the `graphics` reference being passed and set the color you want to use to paint with by invoking the `setColor` method. To get Java predefined colors, you invoke the `Color` class and then access `pink`, in this case. (You could pick any other colors available, but let's go with pink.)

Next, you invoke the `fillOval()` method from `graphics`. Here you are saying that you want the `graphics` to draw you an oval shape that starts from `70` pixels from the left (the first value passed as an argument) and `70` from the top and make it `200` pixels wide and `100` pixels tall. Since the `graphics` reference already knows what color will be used to draw and fill the oval shape, you should be set.

But now how do you then show this circle on-screen in your frame?

Remember, `DrawCircle` is just a `JPanel`—a component like any other, except this one happens to draw something. You can still treat it like any other component you've dealt with.

You need to instantiate an instance of this class (`DrawCircle`) inside of the `FunColor` constructor:

```
DrawCircle drawCircle = new DrawCircle();
```

Next, you'll need to get the frame's content pane so you can add `drawCircle` to it:

```
Container frame = getContentPane();
```

Now that you have access to your content pane as a frame variable, you can just add your circle to your frame:

```
frame.add(drawCircle);
```

To position your circle on the frame, you can also pass `BorderLayout` and invoke the `CENTER` attribute, which will make sure that the circle stays in the center region of the entire frame:

```
frame.add(BorderLayout.CENTER, drawCircle);
```

Next, add a label that says `"Click anywhere..."` so the users know what to do. You want this label to be in the `SOUTH` region of the `BorderLayout`:

```
frame.add(BorderLayout.SOUTH, new Label("Click
anywhere..."));
```

Finally, remember to instantiate `FunColor` in `main`; otherwise, this won't work.

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

Run the code and you'll see a pink circle and the label at the bottom (`SOUTH` region) of the screen.

So with just a few lines of code, you're able to draw a circle on the screen!

Let's now go back to the `DrawCircle` class and create a random color for the circle.

To create colors, you need to mix three main colors: red, green, and blue (RGB values). Each one of these colors ranges from 0 to 255. You can instantiate a `color` object in Java Swing, but you need to feed the RGB values in the constructor:

```
Color color = new Color(0, 0, 200);
```

This will generate a blue color.

But you want all of this to be generated randomly, meaning that this RGB combination needs to randomly generate values in the range between 0 and 255 and then pass those values into the `Color` constructor so you get the random color. Here's how you do it.

Create three variables that will hold random values between 0 and 255 for each RGB variable:

```
int red = (int) (Math.random() * 255);
int green = (int) (Math.random() * 255);
int blue = (int) (Math.random() * 255);
```

Here you are leveraging the Java Class Library by using the `Math` class and invoking the `random` method. You take that random number and multiply it by `255`. So, each time `Math.random` is called, you get a random double value between 0.0 and 1.0. When you get that double, you multiply it by 255. But you actually need to convert this double into an `int` since the `Color` constructor only takes `ints`, not doubles. You do this conversion by casting the result you get to an `int` with this syntax:

```
int red = (int) (Math.random() * 255);
```

Then, you pass the new `int` value to the `red` variable and do the same with the other two (`green` and `blue`) variables:

```
int green = (int) (Math.random() * 255);
int blue = (int) (Math.random() * 255);
```

Next, you instantiate a `color` object and pass those three random values in the constructor:

```
Color randomColor = new Color(red, green, blue);
```

Now you'll have a random `color` object every time the `paintComponent()` method is called. But you still need to pass this randomly generated color to the `setColor()` method of the `graphics` reference so that it knows which color to use when painting and drawing the shape:

```
graphics.setColor(randomColor);
```

Run the code and you'll see a circle filled with a randomly generated color. If you exit the frame and run the code again, you should see a different color each time.

Even though you are getting a circle with different, random colors each time you run the code, notice how the circle is not centered. Ideally, it would be great to have the circle in the middle of the screen.

To make sure the circle is in the middle of the frame, you need to do some math inside of the `fillOval()` function.

You know that the first argument you pass controls the horizontal displacement of the circle. Let's test this by padding `-90` for the first argument:

```
graphics.fillOval( -90, 70, 200, 200);
```

Run the code and you'll see that the circle is now to the far left. But what you really want is to make sure the circle is always in the middle of the screen in the *x*- and *y*-axis (horizontal and vertical). Let's start with the *x*-axis.

First, you need to get the width of the entire panel, which you can get by invoking the `getWidth()` method inside `DrawCircle`:

```
graphics.fillOval( this.getWidth(), 70, 200, 200);
```

You use the `this` keyword to say you are invoking the `getWidth()` method of this current class, `DrawCircle`. You could leave out the `this` keyword and just write `getWidth()` and it would still work, but it makes it easier to read and understand where the method is coming from if you use the `this` keyword. Now that you have the total width of your screen, you must subtract `200` from that, which is how wide the circle is. Next, you divide the difference by `2` to get the middle of the screen. Notice the surrounding parentheses before you divide by `2`. This is important because you want to get the difference first and then divide by 2:

```
graphics.fillOval( (this.getWidth()-200)/2
, 70, 200, 200);
```

So in this case, the `getWidth()` method will return `500` (because you set your frame width to be 500). Because 500 – 200 = 300 and 300/2 = 150, now your circle will be 150 pixels from the left of the screen.

Run the code to see the results.

If you resize the screen, the circle will always appear to be in the middle horizontally, in the *x*-axis, because the width of the screen is being calculated dynamically.

Now do the same to the *y*-axis:

```
graphics.fillOval( (this.getWidth()-200)/2, (this.
getHeight()-200)/2, 200, 200);
```

Run the code and you should see that the circle always shows in the middle of the screen!

It's always better to extract some of these values you are passing to variables; that way, it's much easier to change them if you need to. Let's make the values for the width and height of the circle (`200`) variables, right above the RGB variables in the `paintComponent()` method:

```
int circleWidth = 200;
int circleHeight = 200;
```

Now you can just use these variables instead of hard-coding the values:

```
graphics.fillOval((this.getWidth() - circleWidth) / 2,
(this.getHeight()
       - circleHeight) / 2, circleWidth,
circleHeight);
```

Whenever you want to change the values for the width or height of the circle, you can just change them in one place—in the variables you've just declared.

Run the code and you'll see that the circle is still in the center of the screen. No matter how wide or tall you make the screen (by resizing), the circle will always be in the middle!

The final thing you need to do to finish your fun program is to make it so that when users click anywhere on the screen, the circle changes colors randomly. This means you need to attach an event listener that would listen for mouse events and then to something when the click event is detected.

Go back to the `FunColor` class because this is where you are instantiating the whole frame and its contents: the `drawCircle` object.

Since `drawCircle` is a `JPanel`, you can attach to it an event listener and listen for events. There are many different types of events you could listen to, but you are interested in the `MouseListener` so that you can listen to the mouse-click event. Let's add that one:

```
drawCircle.addMouseListener(new MouseListener() {
   @Override
   public void mouseClicked(MouseEvent mouseEvent) {
       // put code to run when the mouse click is
detected on the screen
   }
```

…

Then, you pass the `MouseListener` object, which has a few override methods you can use to respond to events. You are interested in the `mouseClicked()` method. Inside of this function is where you want to have the code that would repaint the screen, which means that the `paintComponent()` function of the `DrawCircle` class is called again: Redraw the circle with a new random color!

To repaint or call the `paintComponent()` method, you call the `repaint()` method from your frame:

```
drawCircle.addMouseListener(new MouseListener() {
   @Override
   public void mouseClicked(MouseEvent mouseEvent) {
       frame.repaint(); // this redraws the circle!
   }

…
```

You can ignore the other methods.

Run the code once again and click anywhere on the frame window. You should see your circle change color each time you click the frame!

One thing you'll notice is when you resize the window, the circle also changes color. This is because the `paintComponent()` method is automatically called whenever the window is resized—meaning that the frame is repainted.

Java Swing and the Abstract Window Toolkit come with many other classes that are readily available for you to use to create stunning GUIs.

Using the skills you've learned about Swing, you can build even more complex Java user interfaces. Just remember that the Java Class Library is your friend: use it to learn more about other classes you can use to build your next Java Swing application!

Resources

Haase and Guy, *Filthy Rich Clients*.

Schildt, *Java*.

Sierra and Bates, *Head First Java*.

33

Android Studio: Setup, Emulator, and First App

You should already have Android Studio installed on your machine. Open Android Studio and create a new Android project by going to File → New Project → Select a Project Template.

There are different project templates to choose from. Also, notice how many different kinds of devices run on Android that you can create apps for (Phone and Tablet, Wear OS, TV, Automotive, Android of Things).

You're interested in the Phone and Tablet tab, and you'll select an Empty Activity for your template. Then click Next.

Now you need to configure your Android Project. You need to give your project a name. Let's call it `HelloAndroid`. Notice that the package name changes as you type the name for your app. The package name is the unique identifier for this app, so make sure you have a unique set of strings to identify this project. It's customary to use the reverse domain name of your website address or company name, such as com.google.myfirstandroidapp. It just needs to be unique.

Then, you save the project in a familiar location on your computer. You can choose where to save the project.

For the language, you'll use Java, of course.

There's a new programming language that's used to create Android apps called Kotlin, hence the option of Java or Kotlin.

The minimum SDK is the version of Android that you want this app to target. Android is an evolving platform, which means that there are different versions of Android. By choosing a minimum SDK, you are essentially choosing what version of Android you want your app to run on. Different devices run different versions of the Android system, such as Android 4.0 or 8.0. Each successive version often adds a new API (set of libraries and functionalities) that's not available in the previous version. To indicate which set of APIs are available, each version specifies an API level. Each API level also has a name associated with it, such as Nougat, Pie, or Marshmallow.

The lower the API level you choose as the minimum SDK, the more devices you are going to target, and the higher you go, the fewer devices you'll target. This makes sense because the higher the API is, the newer the Android version is, and fewer devices are out there running on that version of the Android system.

Traditionally, all Android versions were named after a dessert, but that changed with Android 10. And Google will continue using numbers for versions instead of dessert names.

In this case, choose API level 21: Android 5.0 (Lollipop). This will make the app run on approximately 90% of devices.

You can look up this information by changing the API levels, and at the bottom you'll see the percentage of devices running that particular Android version.

Now click Finish.

It will take a few seconds for the IDE to create your project, and then you'll see your Android project ready to be worked on.

Before you start exploring your project, you need to set up an emulator so that you can test your apps while you are developing them. An emulator is a virtual device that emulates a physical Android device so that you can run your apps on it and test them to make sure they're working properly.

To create an emulator, click on the AVD Manager icon (top right) and you'll see a window where you can create a virtual device (emulator). In the bottom-left corner, click the Create Virtual Device button.

Then, you'll need to choose the Hardware (device definition). Since Android runs on a plethora of devices, you get to choose what kind of emulator you want to create. Let's pick the Phone category. You'll see a few options. Let's pick Nexus 6. Click Next.

Now you need to pick the version of the Android system that will run on the emulator. Let's pick Oreo, API level 26.

Click Next.

Now you get to give your emulator a name. (You can leave what the IDE recommends.)

Click Finish.

Wait a few seconds and you should see the Your Virtual Devices window showing the newly created emulator.

Click out of this window to get back to the main area of Android Studio.

Now you can drop down the menu in the menu bar close to the Play button to see the new device that was added.

Click the Play button to start the emulator and run this Android project. (It may take a little while for the emulator to start and run the project.)

A few moments later, you should see your app running on the emulator that has a `Hello World!` text in the middle of the screen.

You've now run your first Android app on the emulator!

Resources

Android Developer Fundamentals, https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/index.html.

Phillips, Steward, Hardy, and Marsicano, *Android Programming*.

33

Android Project Structure

An Android app project is structured in a way so as to separate the views (user interface, or UI) from the logic (Java code). This structure makes the development of Android apps much easier for the developer: Design the app (how the app will look, where buttons and any other UI widgets are going to be on the screen, etc.) and then design the logic side of the equation (use Java to write the logic—for example, what needs to happen when a button is clicked).

This connection between the UI and the logic (Java code) is facilitated by a very important resource class called `R.class`. It's named `R` because it stands for `Resource`, a class that references resources in Android. This class is automatically created by the Android framework and keeps track of every button and view that you add to an Android project so that you can reference anything through `R`.

Go to Android Studio and look at the Android project and you'll see many different folders/packages. Again, an Android project is primarily divided into two main categories: UI and Java code. The UI category includes any resource files that facilitate the creation or design of the screen. All of the UI assets can be found inside of the `res` folder, which includes four packages/folders: `drawable`, `layout`, `mipmap`, and `values`.

Let's focus on the `values` and `layout` folders.

All of the UI resources and other resources are written in XML, which is a markup language that makes it easy to represent views and data.

The `layout` folder is where you have the layouts of the UI: what users will see on the emulator. When you open the `layout` folder, notice `activity_main.xml`.

Double-click on this file and you'll see a new window with a lot of things on it. This is the Design view. Basically, you can design your screen from here. You can add buttons, `TextView` labels, and `EditText` (like `TextField` in Java Swing).

Look at the `Hello World!` string. This is exactly what you see when you run this project on the emulator. When you click on this widget, an Attributes panel shows up to the right. You can change the attributes, or properties, of this `TextView` (such as the size of the text, the color, and so on) from here.

Let's change the size of the text. You can search for the attribute you want to change, too! Change `textSize` to `24sp`. The moment you do that, you see the change in the Design view right away! This is a great visual tool to design your views/UIs.

sp, meaning "scale-independent pixels," is the unit used for texts in Android.

Save and run this project again in order to see the change in size of the `TextView` (`Hello World!`). Now the text `Hello World!` is larger.

Next, let's look at the Java side of things. Go to the `Java` folder and open the `MainActivity` Java class.

Notice that each UI file comes with its accompanying Java file (e.g., `activity_main.xml` and `MainActivity.java`).

Open the file `MainActivity.java` and you'll notice a few very familiar things. First, `MainActivity` extends (inherits) `AppCompatActivity`, and then you have an `onCreate` method, which resembles the `main` method you've seen before.

```
public class MainActivity extends AppCompatActivity {
   @Override
   protected void onCreate(Bundle
savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);
   }
}
```

An `Activity` in Android represents a screen: what users will see and interact with. Think of it like the Java Swing `JFrame`.

The `onCreate` method calls another method called `setContentView()`, which gets the `activity_main.xml` file through the `R` resource class, which invokes the `layout` folder and gets the `activity_main.xml` file. Android inflates the XML file into a Java object and creates a view object that's put on-screen for users to interact with. Thus, you see the `Hello World!` text showing up on the emulator screen when you run the project.

So anytime you add something—for example, a button—in your XML file (`activity_main.xml`), the `onCreate` method, through `setContentView`, will inflate the XML file and show the contents of it.

Let's add a button. Drag and drop a button onto the design screen. Let's add constraints so that the button knows where to position itself on-screen and in relation to the `TextView`.

Let's also give the button a text: `Show`.

Run the project and you'll see a button underneath the `TextView`.

How do you connect this button to your Java code in `MainActivity` so that you can, for instance, attach an event listener to it?

It turns out that for every widget (UI component) you have in your XML, there's the equivalent of that in Java. So there's a class `Button` that represents a button.

Go back to `MainActivity`. First, you create an instance variable in the `MainActivity` class for your button and call it `showButton`.

```
public class MainActivity extends AppCompatActivity {
  private Button showButton;
   @Override
   protected void onCreate(Bundle
savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);
   }
}
```

Inside `onCreate`, you need to access the button you created in the UI XML file (`activty_main.xml`) in order to connect to it. For that, you need to give that button an ID so that you are able to reference it and create a Java object from it.

Go back to the XML file and give the button an ID. The IDE already gave a default ID of `button`, which is fine for you to use for now.

Now go back to Java code and write this:

```
showButton = findViewById(R.id.button);
```

This code connects your `Button` object to the button you created in the XML file by using the `findViewById` function and passing the button ID through the `R` class again.

Now you have a fully functioning button object that you can work with.

To add a click-event listener to the button, you call the `setOnClickListener` method and then pass an `OnClickListener` object, which then overrides the `onClick` method:

```
showButton.setOnClickListener(new View.
OnClickListener() {
   @Override
   public void onClick(View view) {
   }
});
```

Notice how similar this code is to the code you wrote in Java Swing when you added an event listener to a `JButton`. First, you attach the button to a listener, and then you pass the actual listener object, which enables you to override a method, where you write what you want to happen when the click event is detected!

For now, add a printout inside of the `onClick` method so that you can see that the button is actually tracking and listening to click events.

Run the project and then click on the button.

To see the printout, you'll have to open the Run tab at the bottom of the IDE. You should then see this:

```
I/System.out: Hello button!
```

The `System.out.println()` is purely Java, but it still works in Android. This goes to show that you can still call pure Java classes in Android and they work.

Although `System.out.println()` works for logging messages to the console, Android has its own logging class you can use.

To use it to log messages in the console, you write `Log.d` and then pass two strings as arguments. The first one is a tag, which can be anything; it'll work as an ID for you to be able to know where the log is coming from. Usually, you add the name of the class the log is being called from. The second argument can be anything—in this case, you can leave it as is.

```
Log.d("MainActivity", "onClick: ");
```

Run the project again and click the button.

To see what's logged, open the Logcat tab at the bottom of the IDE and you'll see this:

```
...helloandroid D/MainActivity: onClick:
```

You may need to look for it, since there's a lot that's being logged by the Android system. You can also search for the tag name in the search bar.

Resources

Android Developer Fundamentals, https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/index.html.

Phillips, Steward, Hardy, and Marsicano, *Android Programming*.

35

Android `EditText` and the `strings.xml` File

Mobile apps are interactive. Users want to be able to add data and have the app do what it needs to do and then output some results. Let's see how to enable users to add input to the app, and then, when a button is clicked, the app will show what they have entered.

Android has another UI widget called `EditText`, which enables users to enter information.

Let's add an `EditText` into your screen.

Go back to your design screen by opening `activity_main.xml`.

Under the Palette tab and Text, you'll see there are all sorts of text-input widgets. Even though they all have different names, they all inherit from the `EditText` widget.

Let's use the Plain Text one. Drag and drop this onto the screen to the right. Drop it above the `Hello World!` `TextView`. Add the left, right, and bottom constraints.

To the right side of the IDE, you have the Attributes panel. While the `EditText` is still highlighted, give it an ID: `enterName`. Then, hit the Enter key.

`EditText` also has another attribute called `hint`, which puts a text inside of the `EditText` that goes away as soon as the cursor appears inside of the `EditText`. This text is for informing users of what is expected for them to enter.

Let's add `enterName`. For this to work, you'll need to remove the `text` attribute, which has `Name` text. Once removed, you can see `enterName` showing in the design screen.

Notice that there are a few warnings attached to `button` and `enterName`, which are the widgets you added to the screen.

When you hover over and click on the yellow warning, it warns you that the texts you've added to these widgets were hard-coded and they should actually be declared as a string resource.

If you scroll down, there's a button to fix this issue. Click the Fix button, which will extract the string you hard-coded into the string resource file.

What's this string resource file? Go back to your project files on the left panel. Go to the `res` folder, then the `values` folder, and then `strings.xml`.

Double-click on `strings.xml` to open it. You can see that you have an XML file with string resources defined. Each string starts with an open tag, and inside you have a `name` property and the associated text value. You see the two you have just added: `show` and `enterName` string values.

This is how Android wants you to create and store string resources. The reason it wants all of the string values put into this file is so that when you want to make your app available in a different language, all you have to do is translate these string values in this one place only, as opposed to having to change the values individually on every widget of your app.

Run the project and you should see your `EditText` on the screen.

Next, let's wire `EditText` in `MainActivity`.

First, you create an instance variable of type `EditText` and name it `enterName`:

```
private EditText enterName;
```

Then, inside of `onCreate`, right after `setContentView()`, you instantiate and connect your `EditText` object to the UI widget you created:

```
enterName = findViewById(R.id.enterName);
```

While you're here, pull in the `TextView` that says `Hello World!` so that you can change the text dynamically when you click the button.

Go back to `activity_main.xml` and change the `TextView` ID to `helloId`.

Go back to `MainActivity` and create another instance variable for your `TextView`.

```
private TextView helloTextView;
```

Then, in `onCreate`, connect your `TextView` right above `enterName`.

```
helloTextView = findViewById(R.id.helloId);
```

Now that you have all the widgets wired, you can start working inside of the `onClick` method.

What you want to do is retrieve the contents of the `EditText`, and when the button is clicked, you want to dynamically populate the `TextView` with whatever users enter in the `EditText`.

So, inside of `onClick`, first you create a string that will store the contents of the `EditText` called `contents`. You get the contents of the `EditText` by invoking the `getText()` method. You then convert this content into a string by invoking the `toString()` method. Also, you invoke the `trim()` method, which gets rid of all of the unnecessary spaces users may add when entering data.

```
String contents = enterName.getText().toString().trim();
```

Next, you set this string to your `TextView` by invoking the `setText()` method and pass the `contents` string.

```
helloTextView.setText(contents);
```

Let's add a check to make sure users enter something:

```
if (!TextUtils.isEmpty(contents)) {
   helloTextView.setText(contents);
}else {
    helloTextView.setText("Enter something");
}
```

Here you utilize the `TextUtils` class from Android to check if `contents` is empty, and you negate the whole conditional test by prefixing with an exclamation point (`!`) to say that if the `contents` string is not empty, then you set your `TextView` to show the text. Otherwise, you print a message to tell users they need to enter something.

Run your project to test it. If you don't enter anything, you get the message `Enter something`. If you enter `Lucy` and click the button, then you'll see `Lucy` on the screen.

The app is working!

Resources

Android Developer Fundamentals, https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/index.html.

Phillips, Steward, Hardy, and Marsicano, *Android Programming*.

36

Build an Inspiring Android App

The app you're going to build will randomly show a quote on-screen when a button is clicked. The great thing about this app is that you can improve upon it by adding more features if you'd like. It's a solid starting point if you choose to pursue Android development.

First, you create a new Android project (choose the Empty Activity template and name it Quotes).

Open `MainActivity.java` (it should be already open, as Android Studio opens it automatically, as well as the accompanying XML file).

First, create an array that will hold a few `String` `quotes`, as an instance variable:

```
private String[] quotesList = new String[] {
       "A house divided against itself cannot
stand.",
       "Important principles may, and must, be
inflexible."
...
}
```

Next, create the user interface. Open `activity_main.xml`.

Remove the `Hello World!` `TextView`.

Add a new `TextView` that will hold a quote.

Add an ID in the Attributes panel on the right. Give `quotesTextView` for an ID.

Next, add a button underneath the `TextView`. Give `randomQuoteButton` for the ID. For the button text attribute, use `Inspire Me!`.

Extract this text to the `string.xml` resource by clicking on the yellow warning icon and then clicking Fix and Ok.

Make the `TextView` text larger by clicking on `TextView` on the Attributes panel. Search for `textSize` and change it to `18sp`. Next, make the text bold. Search for `bold`, and you should see `textStyle`. Tick the `bold` field so that it changes to `true`, which means the text will now be bold—and you can instantly see this in the Design panel.

Run this project so that you can see what you have so far.

Next, head over to the `Activity` code.

Pull in your widgets as instance variables:

```
private TextView quotesTextView;
private Button showQuoteButton;
```

Wire these instance variables with your widgets in the XML file (right below `setContentView()`):

```
quotesTextView = findViewById(R.id.quotesTextView);
showQuoteButton = findViewById(R.
id.randomQuoteButton);
```

Then, set up your button so that it can listen to click events:

```
showQuoteButton.setOnClickListener(new View.
OnClickListener() {
   @Override
   public void onClick(View view) {
       //code goes here!
 }
});
```

Now you have all the pieces ready to go. All that's left is to work on the logic to randomly select a quote from your array and display it in the `TextView`.

Let's create a method that will return a random quote. Notice that this method must return a string since the return type of the method is `String`.

Inside the function is where you are going to generate a random number and then pass it into your list of quotes (array) so that whenever this function is called, you get a random quote pulled from the array.

First, add a `String` type variable that will hold the actual quote:

```
String getRandomQuote() {
String quote;
```

Then, you need to know the length of the array holding all the quotes. You can get that by invoking the `length` property of the array:

```
int arrLength = quotesList.length;
```

Next, you generate a random number—but you need to make sure that you generate these numbers within an appropriate range. You don't want to generate, for example, the random number 11 because your array holding all of the quotes doesn't have an 11th index. This would generate an `OutOfBoundsException` since you would be trying to access an index that doesn't exist.

To simplify it all, you will use this line of code to guarantee that the generated random numbers will always fall within the range of the length of the array:

```
int randomNum = ThreadLocalRandom.current().
nextInt(arrLength);
```

You invoke the `ThreadLocalRandom` class and call the `current()` method, which returns the current thread, and then you call `nextInt` and pass the bound value—in this case, the length of the array. Essentially, you're making sure that the randomly generated numbers don't end up falling out of range.

Now that you have your `randomNum` generated, you can use it to pass into your array of quotes as an index. If the random number that's generated is 4, then you'll get the 4th index quote and return it.

Then, use the `quote` string variable and assign it to that random quote you've just retrieved:

```
quote = quotesList[randomNum];
```

Now that you have an actual random quote, you must return it:

```
return quote;
```

This is how the `getRandomQuote()` method looks now:

```
String getRandomQuote() {
   String quote;
   int arrLength = quotesList.length;
   int randomNum = ThreadLocalRandom.current().
nextInt(arrLength);
   quote = quotesList[randomNum];
   return quote;
}
```

Next, inside of `onClick`, all you have to do is set the `TextView` to the random quote so that it shows on the screen when the button is clicked:

```
quotesTextView.setText(getRandomQuote());
```

Notice that all you have to do is invoke `setText()` and then pass the `getRandomQuote()` method inside. Since `getRandomQuote()` returns a string, that's fine to do.

Run the project to test it out.

Click the `Inspire Me!` button and you'll see different quotes showing each time you click!

Let's work on the cosmetics so that the quotes showing have some breathing room around them—in other words, add some text padding.

Open `activity_main.xml` and click on `TextView`. In the Attributes panel, search for `padding`. Let's give a general padding of `10dp`.

dp, meaning "density-independent pixels," is the unit used for padding in Android

Save and run the project again to see if that helps.

It looks good, but it needs a bit more padding. Let's change to `30dp`.

Run and test.

It looks much better. You can play around with padding by changing the numbers.

Notice that when you first run the app, `TextView` text shows, not a quote. Instead, to make a quote show on-screen the first time the app runs, go back to `Activity` and, underneath the `showQuoteButton` instantiation, write this:

```
quotesTextView.setText(quotesList[0]);
```

You are setting the `TextView` to show the first quote in the array of quotes.

Save and run the project once again and you'll see a quote on the screen even before you click the button.

Clicking the button, you should continue seeing random quotes as usual.

Congratulations! You've built a simple yet fun Android application from scratch. You can take this app to a new level by adding more quotes, and you can even improve upon it.

Android is a great first framework, but Java is just as useful on many other frameworks. For example, the Spring Framework is a very popular Java framework that's used for building web applications as well as enterprise software that major video streaming companies and e-commerce platforms use. You can learn more about the Spring Framework here:

https://spring.io/projects/spring-framework

If you would like to learn more about building enterprise software with Java, check out this link:

https://netbeans.org/kb/docs/javaee/javaee-gettingstarted.html

Resources

Android Developer Fundamentals, https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/index.html.

Phillips, Steward, Hardy, and Marsicano, *Android Programming*.

Quiz

1 True or false? The `setVisible()` method must be set to true (`setVisible(true)`) for the JVM to show the window.

2 What is the purpose of a layout manager?

3 Can you explicitly specify the location of a GUI element in a window?

4 True or false? In Android, user interfaces and layouts are created with XML, but it's also possible to do so with Java.

5 Which of the following statements is true about the `R.class` file in Android?

   a The `R.class` is responsible for getting the `MainActivity` primary responsibilities.

   b The `R.class` is the connector between the user interface and the Java code in an Android project. Additionally, the `R.class` enables you to access any other resource (strings, raw files, images) in an Android project.

   c The `R.class` makes it easy to regulate the life cycle of an Android app.

6 True or false? Each user interfaces widget (`Button`, `EditText`) must have a unique ID.

7 True or false? Google officially adopted Kotlin as the preferred language for Android development. However, you can still use Java to develop Android apps, and you can also use both Kotlin and Android interoperably since both languages use the JVM.

Quiz Answers

1 True. If you don't set `setVisible()` to true, the window you are trying to display to the user won't show.

2 A layout manager maps areas of the container where you can position GUI elements.

3 No. Instead, you specify a relative position for each GUI element within a layout manager (`GridLayout`, `FlowLayout`, etc.). This is because the JVM needs the flexibility to adapt your GUI layout to various computers without rewriting your program.

4 True. Although XML is the recommended and most used way for creating user interfaces, you can also use Java to do the same (only if necessary).

5 **b**.

6 True. You can reference a user interface widget in your `Activity` through the widget's ID; therefore, user interface widgets must have unique IDs.

7 True. Kotlin is a modern programming language that resembles Java in many ways. Kotlin syntax is less verbose than Java, which is very appealing. However, it doesn't mean that Java is no longer used. Java is still vastly used in Android development.

Bibliography

Haase, Chet, and Romain Guy. *Filthy Rich Clients: Developing Animated and Graphical Effects for Desktop Java Applications.* San Francisco: Addison-Wesley, 2007.

This book is an excellent resource for learning more about creating rich desktop GUIs with Java Swing. The authors of the book do a great job at dissecting each Java Swing topic thoroughly.

Phillips, Bill, Chris Steward, Brian Hardy, and Kristin Marsicano. *Android Programming: The Big Nerd Ranch Guide*. 2nd ed. Atlanta: Pearson Technology Group, 2015.

Android development tools and technologies are continually evolving, but this book does a great job of extracting and presenting all the fundamentals of Android development that remain the same.

Schildt, Herbert. *Java: A Beginner's Guide: Create, Compile, and Run Java Programs Today*. 6th ed. New York: McGraw-Hill Education, 2014.

This is a must-read Java book for beginners. The author, who has written extensively about programming for nearly three decades, incorporates a step-by-step approach complete with examples, self-tests, and projects. The book assumes no previous programming experience.

Sierra, Kathy, and Bert Bates. *Head First Java*. 2nd ed. Sebastopol, CA: O'Reilly Media, 2005.

This book takes you from beginning topics about Java to intermediate and advanced topics. The authors casually introduce each topic with many analogies, quizzes, and thorough explanations that a beginner would appreciate. It's a well-rounded book for anyone getting started in Java.

Website Recommendations

Android Developer Fundamentals. https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/index.html.

Developed by the Google Developers Training Team, this is an Android training website where learners are taught fundamental (and intermediate) Android development concepts. It's an invaluable resource for beginners.

Learn Java. https://www.learnjavaonline.org/.

This is a great resource to learn Java online. This website teaches Java basics and advanced Java topics as well. The great thing about this site is that it has a built-in Java code editor where you can write Java code and run it to see the results live!

Tutorials Point. https://www.tutorialspoint.com/java/.

This is another great online resource to learn Java programming for beginners. The website also has intermediate to advanced Java topics.

4840 Westfields Boulevard
Suite 500
Chantilly, VA 20151-2299
USA
1-800-832-2412
www.thegreatcourses.com