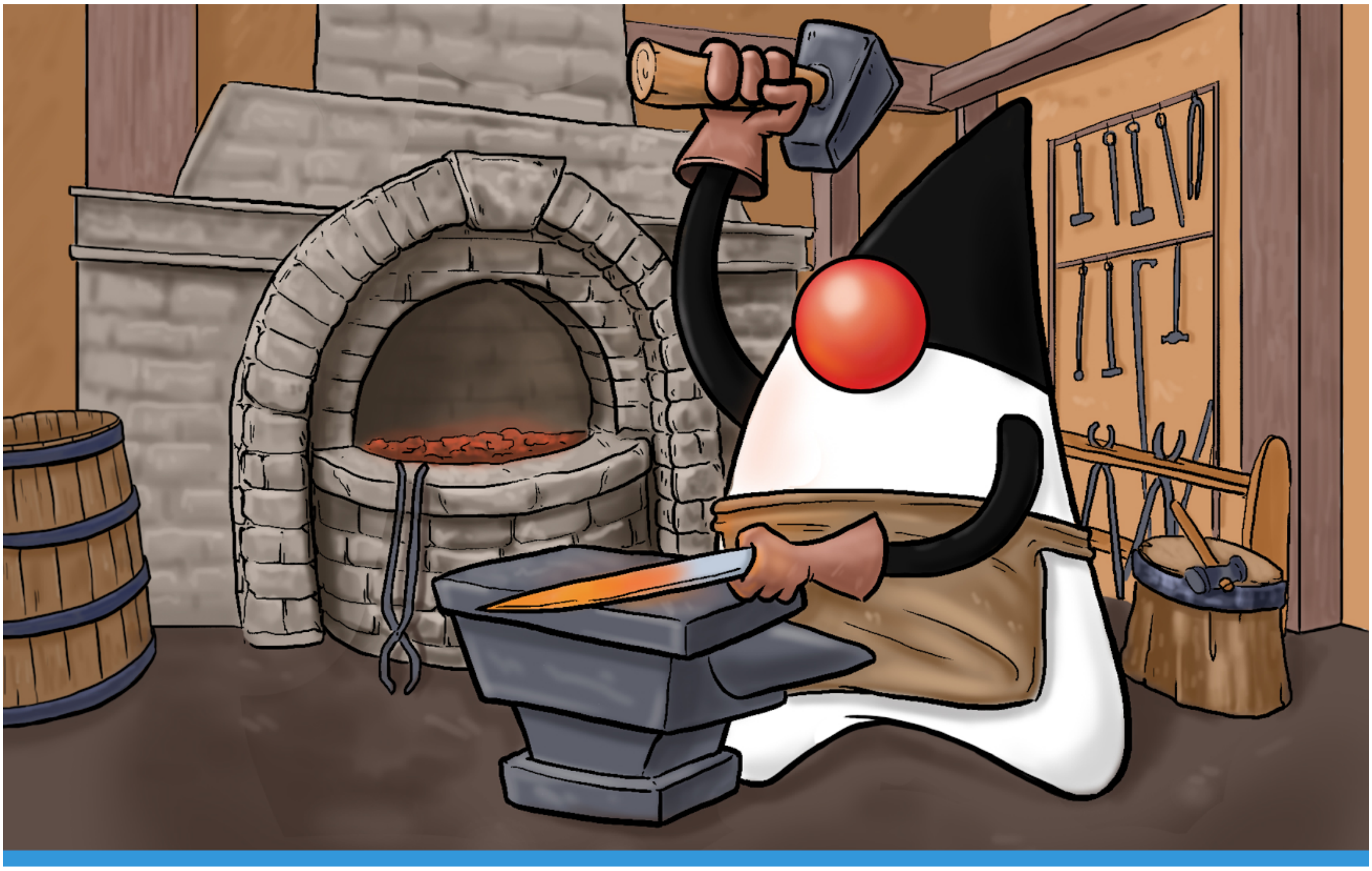


Java

for Passionate Developers

Marcus Biel

CLEAN CODE ACADEMY

Java for Passionate Developers

Marcus Biel

version 0.1

Table of Contents

Part 0: Introduction 5

A About The Author 6
B About This Book 8

Part 1: Getting Started 9

1 Setting up your local development environment 10
2 Basic Java Keywords 13
3 Writing Your First Java Program 23
4 Debriefing 31

Part 2: Basic Concepts 36

5 Instances and Constructors 37
6 Access Modifiers 47
7 Development Tools 55
8 Booleans and Conditional Statements 58
9 Basic Loops 66
10 For-Each Loops 71

11 Arrays 73
12 Enums 82
13 Switch Statements 89
14 Logging 95
15 The Java Main Method 105

Part 3: Intermediate Concepts **112**

16 Exceptions 113
17 Interfaces 124
18 Inheritance and Polymorphism 131
19 The Object Finalize Method 145
20 The Object Clone Method 150
21 The Java Object toString() Method 159
22 Static Imports, Data Types, and More! 164
23 Java Collections Framework 173
24 ArrayList 184
25 Linked List Data Structure 202
26 java.util.LinkedList 211
27 Object Identity vs Equality 228
28 The Java Comparable Interface 233

Part 4: Advanced Concepts **240**

29 Shallow vs Deep Copy 241
30 Immutable Classes 249

Part 0

Introduction

About The Author

Marcus Biel (@MarcusBiel) works as Director of Developer Experience for Red Hat. He is a well-known software craftsman, Java influencer and Clean Code Evangelist. He is also a regular speaker at Java conferences all over the world, such as JBCN Conf Barcelona, JPoint Moscow and JAX London. Besides this, he works as a technical reviewer for renowned Java books such as *Effective Java, Core Java SE 9 for the Impatient or Java by Comparison.*

In 2015, Marcus started a Java blog and YouTube channel that makes Java accessible to passionate developers. There are many advanced tutorials that you can find online, but tutorials with a solid background like this one are rare.

Marcus has become well-known in the Java community, with a total of 70 000 followers across various social media platforms. In 2017, the editorial team at jaxenter.com rated him #13 in their list of the world's top Java influencers.

Aside from this, Marcus is an individual member of the Java Community Process (JCP), as well as a member of the association of the German Java User Groups e.V. (iJUG) and the local Java and software craftsmanship communities in his hometown, Munich.

Marcus has worked on various Java-related projects since 2001, mainly in the financial and telecommunications industries. In 2007 he graduated with a degree in computer science from the Augsburg University of Applied Sciences in Germany. In 2008, Marcus successfully completed his Sun Certified Java Programmer certification (SCP 6.0), of which he is still very proud today.

For Marcus, programming is not just a job, but rather a meaningful, creative craft that he practices every day with joy and passion. Therefore he attaches great importance to the quality of his work. Marcus believes that while programmers can write 'quick and dirty' code, and deceive themselves that they are saving time and money, doing it properly is worth

the initial time and effort. The code will then work reliably and consistently, require less maintenance, and prove to be more economical in the long run.
When he takes a break from Java, Marcus likes hiking in the Alps as well as backpacking. He also likes dancing, a good beer or wine, and enjoying all that his hometown, Munich, has to offer. He lives with his wife and baby son, within walking distance of the Oktoberfest, and yes, he owns Lederhosen.

About This Book

Hi, welcome to my Java 8 book! My goal is to teach you Java in the clearest possible way, and to help you become a pragmatic perfectionist and clean code craftsman like I am!

This book was written in a very unconventional way. In 2015, I started creating a Java video course on YouTube. As the course became more and more popular, the calls for a text version of it became louder and louder. At first I simply published the typed transcripts of my video lessons. Over the years, I kept improving and refining these transcripts and eventually expanded their content beyond that of the original video course. It became a collection of tutorials that could be downloaded as PDFs from my website.

The next step was to turn this PDF collection into the present book. It should be mentioned though that this evolutionary stage is far from complete. There are still many chapters yet to be written, and I am constantly updating and improving the content to keep it current. In the same way that we never stop learning and our own journey is never finished, this book will probably also never be done.

I wish you a lot of fun in reading this book. If you find any mistakes or have any comments, please email me at marcus@cleancodeacademy.com.

Acknowledgment

Over the years I have had the support of hundreds of people without whom I would not have made it this far! I couldn’t list all of them here, but I would like to take the opportunity to thank some of them: Jacques Burns, Lyn and Ben Stewart, Michael Harvey and Ben Eisenberg.

Part 1

Getting Started

Chapter 1

Setting up your local development environment

If you are just starting out with programming, you'll need to set up your development environment to get started. This chapter will show you how do just that on 64-bit Windows. If you already have a Java environment that you are happy with, feel free to skip this chapter :)

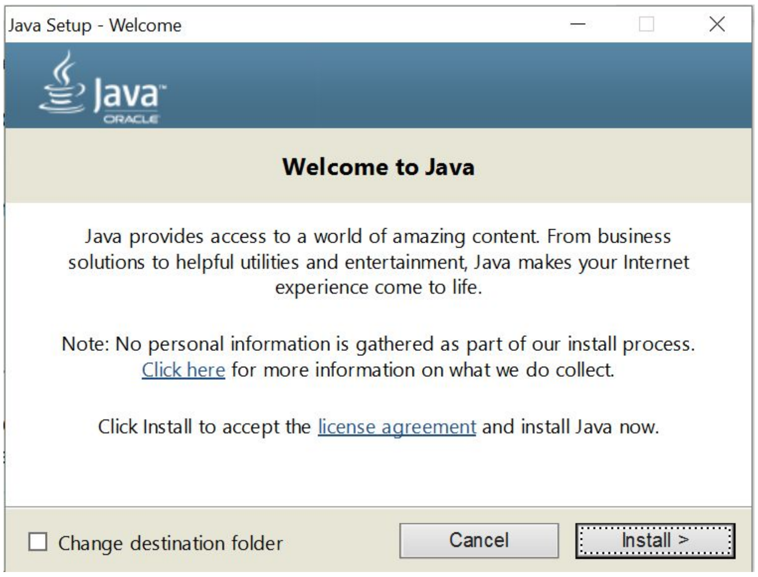
Downloading and Installing the JDK (Java Development Kit)

As you might have guessed, the first thing you need for your Java programming environment is **Java**.

To download the Java Development Kit, open this link. If there are multiple versions available, you can pick whichever one you want, but it's generally best to go with the latest version. Click the blue DOWNLOAD button on the version that you wish to download.

You should now see a bunch of download options. If there are multiple versions on this page, look at the latest version. Before you go any further, click the checkbox that says ○ Accept License Agreement. Now you can select the download for your own operating system. If you can choose between x86 and x64 versions, you should go for the x64 version. If you can choose between downloads with different file extensions, it's best to go with the `.exe` extension.

Click on the link of the installer you need and wait for the installer to download. When it's done, open the installer. You should get a welcome screen similar to this:


Java Setup - Welcome

Welcome to Java

Java provides access to a world of amazing content. From business solutions to helpful utilities and entertainment, Java makes your Internet experience come to life.

Note: No personal information is gathered as part of our install process. Click here for more information on what we do collect.

Click Install to accept the license agreement and install Java now.

If you want to choose where to install the JDK, check the "Change destination folder" option. Leave it blank if you want to install the JDK to the default location. Then click "Install" and follow any prompts you may receive. After everything has finished installing, you may need to restart your computer.

Downloading and Installing Your Text Editor

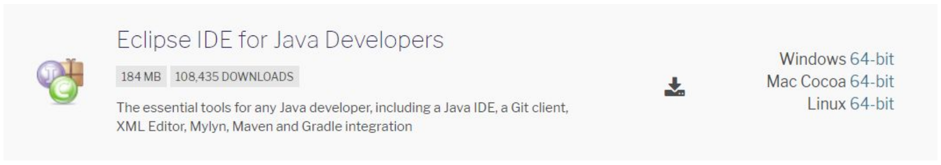
The text editor we will be using is TextPad. There are many others but this one is my favourite because of its simplicity and powerful features. It can be found here.

As with the JDK you should always select the latest version available. You should see a list of versions based on your language. Under your language, select the 64-bit version and wait for the file to download.

Open the downloaded file, and run the setup file inside it to install TextPad.

Downloading and Installing Your Integrated Development Environment (IDE)

Follow this link to view the various editions of Eclipse that are available. Look for the “Eclipse IDE for Java Developers” package shown below:



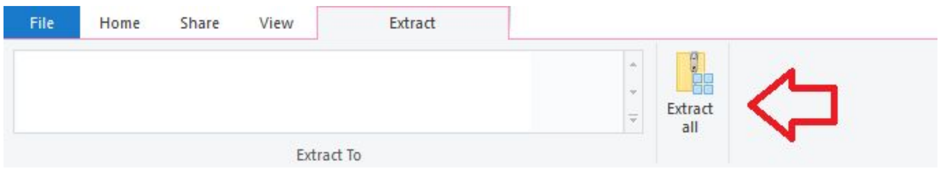
Eclipse IDE for Java Developers

184 MB | 108,435 DOWNLOADS

The essential tools for any Java developer, including a Java IDE, a Git client, XML Editor, Mylyn, Maven and Gradle integration

Select the installer for your operating system and wait for it to download.

Once you’ve downloaded the IDE, open the file and extract its contents to any location on your hard drive. To extract the file using Windows Explorer, simply click ‘Extract all’ at the top of the screen (see below) and choose an empty folder for Eclipse to be stored in.



Make sure that the location you choose is easy to access - you will need to navigate to your folder whenever you want to run Eclipse.

Once the files have been extracted, you can run the file named ‘eclipse.exe’ (the correct file has the Eclipse logo as the icon). It will take a while to load and may slow your computer down. Once Eclipse is open, it will run smoothly.

You will be prompted to choose a workspace - you can use the default one or choose another location for your first workspace. Workspaces are like desks - you could have many desks, which are all laid out differently, with different purposes for each desk. You don’t need to worry about multiple workspaces when you are starting out with Java programming.

Now you’re all set up, and ready to start programming in Java!

Chapter 2

Basic Java Keywords

Introduction

In this chapter I'm going to introduce you to fifteen Java keywords / concepts that are central to Java. To make the concepts more concrete, I'll illustrate them using the analogy of cars in a garage.

Please note: I really try not to use any term before I have officially introduced it to you. Getting a keyword explained using concepts I didn't know was something that always ticked me off in classes, so I aim never to do this to you. For this reason, I might intentionally not explain things one hundred percent correctly at first, but later, if necessary, I'll add the academically correct explanation.

package

Ok, let’s start off with our first keyword: **`package`**. A `package` is like the folder structure on your computer. It organizes the different files of your program. A `package` name must be unique since there are millions of other programs out there, and you don’t want to have the same program file name as someone else. Since Java code is heavily shared, the packages are used to prevent name clashes. If two different code files in one program have the same name, including their package name, only one of them will be loaded and the other code file will be completely ignored, which will usually result in errors.

To guarantee that a package is unique it is commonplace to include one's domain name as part of the package. Usually, the top or root folder of a package structure is the organization's domain name in reverse order. Using a package name is never required, but it is highly recommended, even in the most basic programs. When declaring a `package`, you must declare it in the first line of your program. Lowercase and singular nouns are commonly used for each part of your package name. In my example, I'm starting the package name with my top level domain, `com`, followed by my domain name, `marcusbiel`, then `javaBook` since that is the project we are working on. As I stated earlier, the sample code in this chapter is related to a garage, so we will end the package name with "garage". Our full package name will therefore be:

```java
package com.marcusbiel.javaBook.garage;
```
Example 1

Where possible, use terms that the client can understand and that relate to the business (garage in my case) rather than technical terms like "frontend" or "backend".

import

To simplify programs and reduce redundancy, many program files reuse existing code. To do so, you could use the code file's full name including its full package name. However, to make your code more readable, you would usually add the file to the list of imports, which will allow you to address the code by its simple file name, without having to constantly reference it by its full package location.

The import statement, or statements, is written directly after the package declaration. When you import code, you include the package name of the file. You can import specific files from the package, or the entire package, by using a star symbol.

Let’s begin our code by importing the first car into our garage, a `Bmw`.

```
package com.marcusbiel.javaBook.garage;

import com.marcusbiel.javaBook.car.Bmw;
```

Example 2

Class

Programs can easily consist of thousands, if not tens of thousands, of lines of code. When you have so much text in one place, it’s easy to get lost. To make code clearer, Java programmers classify their code into different *units*, called **classes**.

As a programmer, you are free to name your packages and your classes whatever you like, but your goal should always be clear. When you’re working with a business client, the client defines *what* he wants; it is your job to know *how* to express this. If you can create a common language to bridge the gap between the two, it will be easier for your client to understand the code without much coding knowledge. For this reason, programmers tend to name their classes using a noun that describes the class, starting with a capital letter.

For example we could create a `class Car`, which will later contain all the code related to a car (Example 3). At the beginning of a class, there is an opening curly brace, and at the end of our code there is a closing curly brace that shows the end of the class.

Classes should only be focused on one topic, and you should try to make them as small as possible for the sake of readability and maintainability.

```java
class Car {
    [...]
}
```
Example 3

Method

A class will consist of one or more methods. A method defines how a certain task will be completed using code. For example, a method `timesTwo` could double the amount of a given input. Methods are also sometimes called functions. While this term isn't totally academically correct, you may use it. By convention, a method is usually named after a verb that describes what actions it performs. Methods can operate on anything: numbers, colors, sounds - you name it.

```
timesTwo(2) => 4
add("ab", "c") => "abc"
print("Hello") => will print "Hello"
```
Example 4

Imagine your code as a book. A book has chapters (the packages), paragraphs (classes), and sentences (methods). Methods can also call other methods, creating a chain of methods. As a programmer, you want to structure your code by thinking of the current level of abstraction, just like writing a book. Your code should end up being readable like a book!

An example of a method calling other methods would be a "`prepareDinner`()" method internally calling a "`prepareAppetizer`()" method, followed by calling a method called "`prepareMainCourse`()", followed by a method called "`prepareDessert`()".

If our `prepareAppetizer`() method then has to call three more methods, "`washLettuce`()", "`addTomatoes`()" and "`tossSalad`()", we've created a readable and understandable hierarchy in our code. We could, of course, have `prepareDinner`() directly call all three of these methods, instead of `prepareAppetizer`(), but that would clutter our code and make it difficult to read.

Coming up with a clean structure and making your code “speak” is important. The harder it is to understand what your program does, the easier it will be to introduce an error. As a rule of thumb, try to keep your methods shorter than twenty lines. Personally, I aim for a method length of just 1-3 lines.

Methods are defined similarly to classes. First you define the name of the method, usually a verb beginning with a lowercase letter. Methods are normally verbs because they do something. Classes are usually nouns because they are the objects that the methods are acting on/in. The name of the method is followed by a pair of parentheses. Inside these parentheses you can define anywhere from zero to an unlimited number of input fields. When you call a method, you need to send specific values, called arguments, to the specific fields, called method parameters. Arguments are the actual values you are sending to the method. In the context of the method, the general values it will receive are called method parameters. I recommend **an absolute maximum** of three to five method parameters, because more than that harms readability. The fewer parameters, the better. After your parameters, you begin your method definition with an opening curly brace and end it with a closing curly brace. In between, you put the code for your method. Let’s take another look at our `Car` class:

```
package com.marcusbiel.javaBook.garage;

import com.marcusbiel.javaBook.car.Bmw;

class Car {

    drive(speed) {
        [...]
    }
}

```
Example 5

Object

As I said before, a class is used to structure code in Java in the form of units of code. However, this is only part of the story. A class is like a blueprint for what you want to do when the program is running. The class `Car` defines how a car will behave. However, at runtime (when the program is running), there will be a number of cars, each with its own set of values.

So a class only acts as a template for the objects that are created in your program. Each object has the same set of behaviors, as defined by the class, but it also exists as its own set of values in the program that could change as the program runs. For example, you could have two Objects of class `Car` that both drive, but they could have different values for their respective speeds.

Variable

A variable, as the name implies, varies in value. It is a **placeholder** for a value that you can name and set. Variables can be used as the input and output for methods. Variables can be a variety of different data types. There are two categories of data types: primitive data types and object types. Primitive data types such as `int`, `char`, and `boolean` store things like numbers and single characters. Object types are used to store objects and are referenced by object reference variables. An example of a possible reference variable name for an object of type Car would be `myCar`, `myPorsche`, or `momsCar`.

Dot

In Java, a "`.`" is not used to indicate the end of a sentence. Instead, a "`.`" is used by a variable to *execute* or *call* a method. For example, we could have a variable `car` call the method `drive()`.

Semicolon

Since the "`.`" is used to indicate a method being called, the "`;`" is used to indicate the end of a command, statement, or declaration.

```
car.drive();
```

Example 6

Variable Declaration

Before we can use a variable, we need to define it. You declare a variable by writing the type of the variable, followed by its name, followed by a "`;`".

```
Car myPorsche;
```

Example 7

Object Allocation

Once you've declared a variable, you can allocate it to a specific object. You can also do both of these things in one line. First, you can create a variable of type Car, called `myPorsche` and then, using an equals sign, assign it to a new Car object, with a first value of 1 and a second value of 320.

```
Car myPorsche = new Car(1, 320);
```

Example 8

After we declare the variable and assign it to the object, whenever we use `myPorsche` we are referencing this object created in the memory. You might also notice that we put two values into the constructor of Car, but without actually looking at the constructor of Car we wouldn't know what their meanings were. This is one of the reasons to have as few fields as possible in methods and constructors.

public

`public` is an access modifier that defines that the class or method that is tagged as `public` can be used by any other class from any other package. Besides `public`, there are also other access modifiers such as `private`, default, and `protected`, which I will cover in more detail later.

void

For every method, it is required that you return a value, even when you don't want to. However, the tag `void` defines that the method will not return a value. If a method isn't `void`, it can return any kind of data type. A clean way to design your methods is to make each method fulfill one of two roles. A **query method** returns a value but doesn't alter the state of a class. A **command method** alters the state of a class, but doesn't return a value. `drive(speed)` is a command method. It performs an action ("driving"), but it doesn't return a value (and thus we define it as void).

```
package com.marcusbiel.javaBook.garage;

import com.marcusbiel.javaBook.car.Bmw;

public class Car {

    public void drive(speed) {
        [...]
    }
}

```

Example 9

@Test

The `@` symbol indicates an annotation. Annotations are a way for the programmer to express to the compiler certain background information about a method or a class that isn't directly part of the code. For now, I want to highlight one important annotation. Just know that any method annotated with @Test indicates that a method is a test.

camelCase

When a class, variable, or method consists of more than one word, you use an uppercase letter to indicate the new word. This is called camelCase, because the "humps" from the upper case letters make the words look like camel humps. In some other programming languages, multiple word names are separated with underscores, but in Java "camelCase" is used instead.

Object Oriented Language

Each programming language has a design in mind for its structure. Java is an object oriented language. That is, everything in Java is focused around objects. A clever Java programmer works towards fully understanding the problem and translating his theoretical model into objects and classes that help him achieve his goals.

The main purpose in writing a program is to solve a problem in the most concise way possible. The better job a programmer does, the easier the code will be to maintain, the faster the program will run, and the fewer errors the program will have.

Java’s power comes from the idea that a model takes names and concepts from the real world, allowing programmers to communicate easily with less code-savvy clients. While you might be thinking about the code behind your Car class, your client might be thinking of a real car, but both of you will understand each other.

Chapter 3

Writing Your First Java Program

Writing a Test

Whenever you write a program, you start by writing a test. Tests are extremely useful for a variety of reasons. A test allows the programmer to properly think through his idea, which will result in a cleaner final result. The test also serves as a formal definition of what needs to be done, and can be used for discussions with the client, before a single line of code for the program has been written. It gives a programmer the chance to ask the client: “Is that what you actually need?”

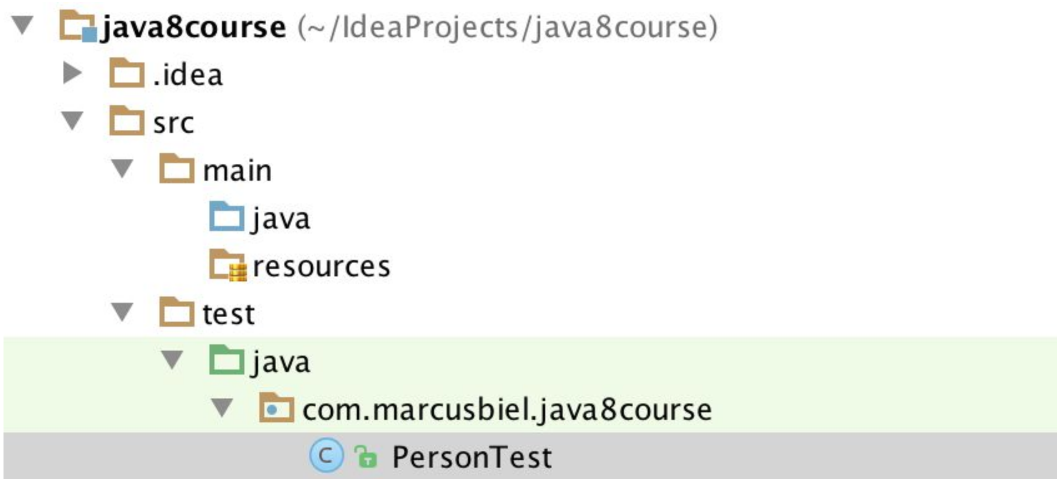
Another reason to write tests is that it will help in writing cleaner code. A test gives an external view on your program – if your code is difficult to use, you will experience the pain yourself, and this will make you refactor and improve it. When this happens to me, I usually say: “The test speaks to me”.

Last, but not least, for every refactoring or new feature added to the code, the test acts as a safety net. Ideally, whenever you change code an automated test should run and should fail if you introduced a bug. The test can show you the exact code line that “broke” the test, so you can instantly move in the right direction towards fixing the bug. A failing test is usually visualized as a red bar, and a passing test as a green bar, so we often speak of “green” and “red” tests instead of “passing” and “failing” tests respectively.

PersonTest

We write our code in our program editor “IDE” - the short form for “integrated development environment”. There are a variety of IDEs for Java and each has its own specific functions. Generally, they act as editors for your code, highlighting different keywords, pointing out errors, and assisting you in your coding. An IDE is extremely useful and makes every step of the programming process easier for the programmer. For example, they help programmers by completing lines of code, automatically creating package structures, and auto-importing packages when they are used. However, I think that an IDE is a crutch for those who are learning to code. It doesn’t allow you to completely understand what you’re doing and how you’re doing it and if at some point you can’t use an IDE, you wouldn’t be able to function without it. For that reason, I’d recommend you start coding by using a simple text editor and compiling from the console until you’ve completed the first twenty chapters of this book. This will allow you to focus on learning the key aspects of Java without the assistance of the IDE.

Now that I've explained why we should write a test, and where we write our code, let's start to actually code! In Example 1 below, I've navigated to a `PersonTest` class that I just created.


java8course (~/IdeaProjects/java8course)
.idea
src
main
java
resources
test
java
com.marcusbiel.java8course
PersonTest

Example 1

If you look back at Example 1 above, the test classes are all in a folder structure “src/test/java”, and the non-test code will be stored in “src/main/java”. A package name is basically identical to a folder structure, but Java gives the latter a special meaning. So our package name of “com.marcusbiel.javaBook” will end up in a folder structure of “com/marcusbiel/javaBook”. Remember, the folder structure must be identical to the package name in the Java class. Java will check that!

Now, back to our class. We have to start with our package declaration. For our package structure to match the folder structure explained above, we have to declare our package, `com.marcusbiel.javaBook` at the top of our class. To end the statement we have to put a semicolon. Here’s how it looks all together:

```java
package com.marcusbiel.javaBook;
```

Example 2

Next I define my class, which is called `PersonTest`. I do this by first typing public, followed by class, and finally the class name `PersonTest`. Then I add opening and closing curly braces. Between the curly braces is where you write your code.

```
package com.marcusbiel.javaBook;

public class PersonTest {
    [...]
}

```

Example 3

@Test and our test method

According to the JUnit 4 documentation, the annotation `@Test` tells JUnit that the “public void” method to which the annotation is attached can be run as a test case. If we run the test and the test condition is not met, the test will fail.

To use the JUnit `@Test` annotation we import it first, as you can see in Example 4 below:

```
import org.junit.Test;
```

Example 4

As stated above, a Junit 4 test method must be “public void”, and must not accept any parameters. Apart from that, we can freely choose any name for the method that we like. Generally, while testing, focus on what your program should do, not on how this is done. This can be encouraged by following a very simple rule: start the name of your test method with “should”, followed by what you expect the method to do, from a client’s perspective.

In our case we’ll call our test method `shouldReturnHelloWorld`. “`Hello World`” is a running gag in the world of programmers where the first program that you write in any language should always return “`Hello World`”. Needless to say, my book isn’t going to break that rule :).

Finally, above our test method we add the annotation `@Test` to tell JUnit that this method is a test case, as explained above. Here's what our code looks like at this point:

```java
package com.marcusbiel.javaBook;

import org.junit.Test;

public class PersonTest {

    @Test
    public void shouldReturnHelloWorld() {
        [...]
    }
}

```

Example 5

Variables

Ok, let's start writing the test! In this test we expect a `Person` object to return "`Hello World`" to us. In Java, plain text like "`Hello World`" is a specific type of data called a `String`. So in this test method, we're expecting to receive a String "`Hello World`". We haven't actually written any code for the `Person` object to do this, but we're starting in reverse by starting with a test, so you can assume that the `Person` class is done, even though your compiler will tell you that it's not.

For this `Person` we'll create a variable. We need a name for our variable. I'm going to call mine marcus. So to create the variable, we call the constructor, create the object of type `Person` in the memory, and assign it to the variable `marcus`, as you can see below:

```
 1  package com.marcusbiel.javaBook;
 2
 3  import org.junit.Test;
 4
 5  public class PersonTest {
 6
 7      @Test
 8      public void shouldReturnHelloWorld() {
 9          Person marcus = new Person();
10      }
11  }
12
```
Example 6

In Example 6 above you can see that the `Person` class on line 9 is underlined with red dots. This is because our `Person` class hasn't been created yet, but we'll take care of this later. Like I talked about above, this is a central part of writing a test. We're first going to create our test, in full, and then after that we'll create the classes and methods necessary to make it pass.

assertEquals

We can check that the method is returning "`Hello World`" with the help of the static JUnit `assertEquals` method. We import this method by adding the `assertEquals` method below our other import statement, as you can see in Example 7. This is a special kind of import called a static import, which I explain in this chapter.

```
import static org.junit.Assert.assertEquals;
```
Example 7

The `assertEquals` method expects two arguments. The first value is the *expected* value, the second value is the *actual* value. The expected and actual values must be equal, otherwise the assertEquals method will throw an error and the test will fail.

As the first argument, we put the String “Hello World”, as this is what we expect our helloWorld method to return. As the second argument, we directly put the helloWorld method call, as it will return the actual value. Here’s what this looks like all put together:

```java
package com.marcusbiel.javaBook;

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class PersonTest {

    @Test
    public void shouldReturnHelloWorld() {
        Person marcus = new Person();
        assertEquals("Hello World",
                                        marcus.helloWorld());
    }
}

```

Example 8

Currently, the code shown in Example 8 still won’t work because we haven’t implemented the class Person yet. We also still haven’t created a helloWorld method since we haven’t created the class. So now, let’s create this class and the method. Since this is a method of type String, it must return a String. So in this case, our Person class will return the String “Hello World”. In the previous chapter I mentioned the difference between command methods and query methods. This helloWorld method is the first query method we have written. It doesn’t change the state of the Person class, however, it returns something. Take a look at our Person class so far:

```java
package com.marcusbiel.javaBook;

public class Person {

    public String helloWorld() {
        return "Hello World";
    }
}

```

Example 9

Now if we execute our test, we get a green bar, so the test passed successfully! This proves that we have correctly implemented our first working code! Hooray :).

Chapter 4

Debriefing

For review purposes, let’s go back to our development environment and take a look at what we’ve learned already. Firstly, when we write our Java Class, the first line of our program is the package declaration. Usually it’s the domain name in reverse order. While it’s not mandatory to declare the package, I recommend that you always do it.

```java
package com.marcusbiel.javaBook;
```
Example 1

After the package declaration, we have our import statements. In this chapter, we’re going to create a new `Name` class inside the subpackage ‘`javaBook.attributes`’. Since this class is in a different package and we want to use it in our main class, we’ll import it using the statement shown below in Example 2. Please note, I’m creating the `Name` class in a separate package for the sole purpose of demonstrating import statements.

I’d like to take some time to discuss import statements in a bit more detail. We’ve already discussed that we need an import statement is when we are using an Object from a class. When we import the class, we import the full class name, which includes the package name, followed by the name of the class. Here’s an example:

```java
package com.marcusbiel.javaBook;

import com.marcusbiel.javaBook.attributes.Name;

public class Person {

    private Name personName;

    public String helloWorld() {
        return "Hello World";
    }
}

```

Example 2

There’s also another way that we can create instance variables of class `Name`. When we create our instance variable, we can include the full class name appended by the name of our variable. We wouldn’t need to import the class if we did this:

```java
package com.marcusbiel.javaBook;

public class Person {

    private com.marcusbiel.javaBook.attributes.Name
                                                    personName;

    public String helloWorld() {
        return "Hello World";
    }
}

```

Example 3

Firstly, from a readability standpoint, Example 3 is barely readable. On top of that, while you may not see any difference in this one line, having to repeatedly use the full class name to create instance variables is much longer to type out repeatedly.

The only time when you *have to* use the full class name rather than an import statement is when you have two different classes with the same short name, but the two classes are from different packages. For example if you had our class “com.marcusbiel.javaBook.**attributes**.Name”, but you also needed to use a class “com.marcusbiel.javaBook.Name”, you wouldn’t be able to import both classes. So you can import one and use the full class name for the other.

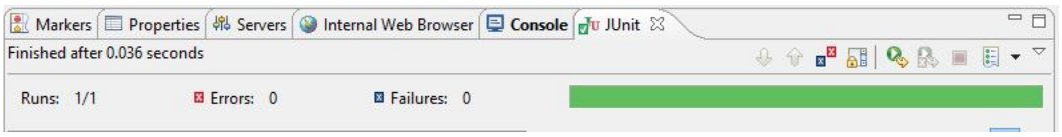
Next we have our class definition, `public class Person`, followed by opening and closing brackets. The opening bracket defines the beginning of the code for the class, and the closing one marks the end. The keyword `public` means that any class in the same package, or in any other package for that matter, can see this class. Just like this `Person` class can see the `Name` class, that `Name` class is able to see the `Person` class.

Now, inside the `Person` class, we’ll define a reference variable `personName` which is of type Name. To make this reference variable inaccessible to other classes, we define it as `private`. But since we want this field to be accessible to other classes, we’re also going to create a public method `name()` which is easily accessible and will return this reference variable `personName`. Any class that wants to access the `personName` variable will have to call the `name()` method.
There is no naming convention for this method, so we can name it whatever we want. The `name()` method’s implementation should contain a return statement that returns the `personName` variable.

```java
package com.marcusbiel.javaBook;

import com.marcusbiel.javaBook.attributes.Name;

public class Person {

    private Name personName;

    public String helloWorld() {
        return "Hello World";
    }

    public Name name() {
        return personName;
    }
}

```
Example 4

We also have the `helloWorld()` method from our last iteration of the `Person` class. If you remember, we already have a class called `PersonTest` from our last chapter, where we defined the `helloWorld()` method as one of the arguments in an `assertEquals()` method.

Defining this test method gives us an opportunity to think about a good design for our program. Once we execute the Test class and the code has been correctly implemented, we should get a green bar that means our test case executed successfully.



Finished after 0.036 seconds

Runs: 1/1   Errors: 0   Failures: 0

Example 5

While testing, we are always focused on the green and red bars. The green bar means that our test passed and the red bar means that our test failed. You should also note that in the `PersonTest` class, I used the `@Test` annotation and the static keyword in the import statement of the `assertEquals()` method from the JUnit library. I won't go into much detail about this now, but it will be covered in a later chapter. For your reference, here is the `PersonTest` class:

```java
package com.marcusbiel.javaBook;

import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class PersonTest {

    @Test
    public void shouldReturnHelloWorld() {
        Person marcus = new Person();
        assertEquals("Hello World",
                                         marcus.helloWorld());
    }
}

```
Example 6

Part 2

Basic Concepts

Chapter 5

Instances and Constructors

Instance Members vs. Static Members

As you can see below, I have created a class `Person`, and within that class we have an instance variable called `personName` and an instance method called `helloWorld()`. Our variables and methods are called the **instance members** of the class.

```java
package com.marcusbiel.javaBook;

import com.marcusbiel.javaBook.attributes.Name;

public class Person {

    private Name personName;

    public String helloWorld() {
        return "Hello World";
    }
}

```
Example 1

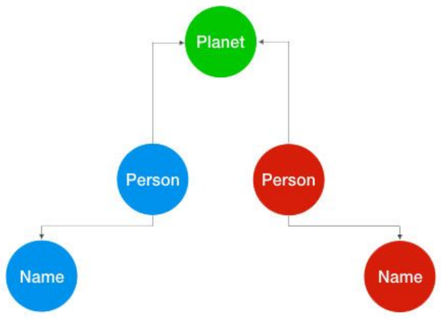
So what is an **instance**? An instance is a single occurrence of an object. While there is only *one* class, there can be *many* instances of the class: objects. For example, we can have hundreds and hundreds of different

Person objects. Each Person object has its own instance of the personName object, each with its own value and its own version of the helloWorld() method. Besides instance variables and methods, there can also be static ones. All instances of a class **share** the static variables and static methods of the class.

Non-local variables and methods are also called “**members”**. A member that belongs to the instance, is called **instance member**; a member that belongs to the class (a static variable or static method) is called **class member**.

```java
package com.marcusbiel.javaBook;

import com.marcusbiel.javaBook.attributes.Name;

public class Person {

    private Name personName;
    private static Planet homePlanet;

    public static String helloWorld() {
        return "Hello World";
    }
}

```
Example 2

If we change the homePlanet value for one instance of Person, all of the other instances will have their values changed as well. This is an advanced topic that will be discussed later in more detail. For now, just know that without the static keyword, our methods and variables are instance variables and methods.



Example 3

Constructor

The next thing we will do in our program is write a **constructor** for our `Person` object. I've mentioned constructors before, but in this chapter I will go into more detail.

A constructor is used to initialize objects. It looks and acts very similarly to a method. However, a constructor does not "return" a value in the normal sense – once called, it "constructs" an instance of the class it belongs to. Hence, a constructor does not define a return value, not even void. Also, there is only one valid name for a constructor – the short name of the class. It is followed by parentheses that, just like a method, may contain a set of parameters. You can have as many constructors as you want in a class, but each constructor needs a unique set of parameters. The constructor body (the space between the opening and closing curly braces) may be used to set up all the object's attributes into a valid state.

As I explain in this chapter, each attribute has a default value. For objects, that is `null`, basically a "reference to nowhere". Further, you can also statically initialize attributes directly in the same line where you declare them. One way or the other, you must ensure that, once the

constructor is executed, your object is in meaningful state, ready to be used. If you fail to do so, evil things may happen.
For example, consider a car object without an engine set up. Once you aim to start the engine, your entire program will crash. This advice is contrary to a lot of what you will read in other tutorials – often you will read about providing setters to set up your object after it has been created. That has a lot of negative consequences. For instance, this is not object oriented programming – it makes you think in data, rather than behaviour. However, it is a more advanced topic and beyond the scope of this chapter. You can read more about why getters and setters are evil on my blog.

If you don't write a constructor for your class, the compiler will implicitly include an empty constructor without parameters. Note that the parameterless, empty constructor added ***implicitly*** by the compiler is called the **default constructor.** Strangely enough, a parameterless, empty constructor ***explicitly*** written in code, is **not** called "default constructor". Relying on the implicitly added default constructor is a bit problematic. If you do (at a later time) add a constructor that does contain one or more parameters, the default constructor will not be added by the compiler, and existing code that relies on it will break. I refer to such automatic compiler actions as "*magic*", as they are not always clear and easily lead to confusion. Therefore, if you need a parameterless, empty constructor, I recommend that you always write it out explicitly. Never rely on the compiler to add it for you. Unfortunately, this may lead to further confusion about why you added a seemingly unnecessary constructor, so you should also leave a comment documenting *why* you explicitly added the parameterless, empty constructor. To give a concrete example: empty constructors are often needed when certain frameworks are in use, like Hibernate, for instance.

For our Person class, I'm going to create a constructor that sets the value of `personName` given a value. If you look at the example below, I've created two different variables called `personName`. One of them is the instance variable in the class, and one of them is an argument in our constructor. To differentiate them, we have to add “`this.`” to the instance variable `personName`. This way `this.personName` is set to the `personName` received as an argument in the constructor.

```java
package com.marcusbiel.javaBook;

import com.marcusbiel.javaBook.attributes.Name;

public class Person {

    private Name personName;
    private static Planet homePlanet;

    public Person(Name personName) {
        this.personName = personName;
    }

    public static String helloWorld() {
        return "Hello World";
    }
}

```

Example 4

Since we’ve created a constructor, the compiler won’t automatically use a default constructor. Now whenever someone calls the constructor to create an object of type person, they have to include the `personName` since that is the only constructor available to them.

If you remember from our test class from the last chapter, we constructed our object without any arguments. If we try to execute this test again, it will fail.

Test Case Creation

As I’ve talked about before, a test method is annotated with `@Test`. This annotation invokes the Test class from the JUnit framework and sets up a test environment in our Java Virtual Machine (JVM). Let's create a second method in our test class so that we can see a little more in-depth how our test class works. In our test class, I’m going to create a second method, `shouldReturnHelloMarcus()`.

```java
package com.marcusbiel.javaBook;

import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class PersonTest {

    @Test
    public void shouldReturnHelloWorld() {
        Person person = new Person();
        assertEquals("Hello World",
                                    person.helloWorld());
    }

    @Test
    public void shouldReturnHelloMarcus() {
        Person marcus = new Person();
        assertEquals("Hello Marcus",
                                 marcus.hello("Marcus"));
    }
}

```

Example 5

Once I’ve created this method, I’m going to create an object of type `Person` with an instance variable name of `marcus`. If we try to run this, we will get a compilation error as I noted earlier, because our `shouldReturnHelloWorld()` method tries to create a Person object with a parameterless constructor, which is no longer being generated since we added the Person constructor that accepts a `Name` instance. As you can see in Example 6 below, to get rid of this error, we'll simply add a parameterless constructor explicitly.

```
 1  package com.marcusbiel.javaBook;
 2
 3  import com.marcusbiel.javaBook.attributes.Name;
 4
 5  public class Person {
 6
 7      private Name personName;
 8      private static String homePlanet;
 9
10      public Person(Name personName) {
11          this.personName = personName;
12      }
13
14      public Person() {
15
16      }
17  }
18
```
Example 6

Comments

As I said before, there might be confusion later as to why you created this parameterless constructor (lines 14-16 in Example 6 above). You may even forget the reason yourself, further down the line. Adding comments to the code will prevent the confusion. They're ignored by the compiler so besides denoting the comment itself, little syntax is required. Do remember to use comments sparingly though, and never comment on the obvious. A variable `person` doesn't need a comment that says "`this is a person`"!

Instead, you should always aim to express your intent in the code itself. For example, a method that adds two numbers that is called `add()` clearly portrays what it does and probably doesn't require a comment to describe the method. There are a few exceptions to this rule, but generally, a comment is used as a fix for bad code that doesn't properly express its intent. Comments can never be as precise as well-written code. Also, while code is always up to date, a comment will usually end up being out of sync with the code. Obsolete comments are really dangerous, and they are often the cause for severe confusion. Therefore, I usually say, "Comments lie" and I never completely trust a comment. Code can't lie. `1` will always be `1` and `0` will always be `0`. Code always does exactly what it claims to do.

There are two types of comments you can define in Java: multi-line comments and single-line comments. Multi-line comments start with `/*` and end with `*/`. Between these two symbols, you can write as many lines of comments as you'd like.

A single-line comment starts with `//` and lasts for the remainder of the line in the code. In general, I recommend that you always use multi-line comments, because if a single-line comment is too long, it will be broken up when the code gets auto formatted, which might happen quite often in a development team.

To pick up from Example 6 again – assuming we would need an empty, parameterless constructor because we are using the Hibernate framework, we could add a comment to the constructor as you can see in Example 7 below:

```java
public Person() {
    /*
     * parameterless constructor, required by Hibernate
     */
}
```
Example 7

Concatenating Strings

Next, I am going to create the `hello()` method that we called in our test class. It receives one argument, which is the name of the person we are "saying" hello to.

In our `hello()` method we are returning a `String`. However, the `String` we return will not always be the same. It will always start with "`Hello`", followed by the name of the person. We have to turn these two separate `String` instances into one `String` that we return. This process of "adding Strings together" is called **Concatenation**. To do this, we put a '`+`' between the two Strings, creating one larger `String`. Now our method will return "`Hello`" followed by the person's name, as you can see in Example 8 below. On purpose, I've left a bug in the `hello` method that will cause our test to fail. See if you can find it!

```java
public static String hello(String name) {
    return "Hello" + name;
}
```
Example 8

Testing our Code and Fixing the Bug

Now let's execute our test. It failed as expected. The reason is that there is a mismatch between our **expected value** which was "`Hello Marcus`" and the **actual value**, “`HelloMarcus`”. When we concatenated our two Strings we forgot to include a space between them!

This demonstrates the great value of tests quite clearly. Programs written in Java often have 100k to 100 million lines of code – some even more than that! Unlike computers, humans get tired easily, and we do make mistakes. Errors like these can have disastrous consequences, from billions of dollars lost when a trade system goes berserk, to human lives lost because of a space shuttle miscalculation. Therefore, you must always write tests. Tests act as the formal specification of the program. A program without unit tests is a program without a formal specification, and that is worthless in the best case and harmful in the worst.

Okay, let’s go back to our failing test. To make it pass, we have to add a space after the word “`Hello`”, as demonstrated in Example 9 below.

```java
public static String hello(String name) {
    return "Hello " + name;
}
```
Example 9

With that done, when we execute our test again, it’s green and we have successfully applied String concatenation for the first time!

Chapter 6

Access Modifiers

There are four access modifiers that exist in Java, three of which I will cover here: `public`, `private`, and *default*. The fourth modifier is `protected`, which is related to a more advanced topic (inheritance), so I will skip it for now. There are also many non-access modifiers. For now, I will only be focusing on the `static` modifier.

The Public Access Modifier

The public modifier signifies that a method, variable or class is accessible from ***any*** other class. For example, the `Person` class that we've used in previous examples can access and use the `Name` class because the `Name` class is `public`. Please note that I've placed the `Name` class in a separate `package` for demonstration purposes only.

```java
package com.marcusbiel.javaBook;

import com.marcusbiel.javaBook.attributes.Name;

public class Person {
    private Name personName;
}

```

Example 1

The Private Access Modifier

The `private` modifier signifies that the method or variable is only accessible from the class where it was declared. Make all variables and methods `private` until you absolutely need to make them `public`. This is very important, especially for variables. You don't want other classes to poke around in your `Person` class and have them changing your `Person` object's name whenever they feel like it. Generally, if you want to expose some variables to outside classes, you should not make the variables `public`. This allows outside classes to not only see them, but also modify them without any restrictions.

A Data Centric Approach

A commonly used alternative to making all your attributes `public`, is to provide so-called `public` "getter and setter" methods, that allow other objects to directly access and change your `private` attributes. This approach is taught as "object oriented programming" by many Java books and courses. To me, this always felt like having a locked door, with a key and a note saying, "Please don't open this door" stuck on it. It took me many years to realize that I wasn't the only one who was confused, but that actually all those "smart books" and teachers were wrong! **Don't let them fool you!** It is fallacy to believe that you can effectively encapsulate a class while still providing `public` methods that allow one to ***directly*** operate on its internals. (For more details, read my blog post about why getters and setters are evil). Avoid this data-centric approach.

An Object Oriented Approach

Instead, use an object-oriented approach. Focus on providing functionality from a business point of view, independent of the internal details of a class.

When designing your class, put yourself in the shoes of someone who will have to use your class. Make it as simple as possible for the caller to use the class's methods. *To achieve this, focus on what the class should do, and not on how this will be achieved.*

After careful consideration, offer only a small set of well defined public methods, independent of the internal details of a class. **Generally, less is more.** The less the client "knows", the more flexible your code stays - every method that is not `public` can easily be changed, without affecting other code.

As an analogy, a house also has a well defined number of doors, and they are usually closed. The house owner decides **if**, **when,** and **how** he wants to open them. For example, he's not going to open the safe door when a delivery person comes to the door, but he might open the front door for the person, so they can carry his package inside.

See every `public` method as an open safe door; as a potential **threat** to your class.

Finally, you should also always validate incoming arguments. If the package the delivery person brought was supposed to be a new book, but it was ticking, the house owner probably wouldn't let it come inside. The same goes for your `public` methods. As a Software Craftsman, you must make sure that each class doesn't cause harm to the system, even when it's used beyond its intended purpose.

I will continue to talk about these object-oriented principles throughout the book as they are very important to good code design.

The Package-Private Modifier

The third modifier I'm going to discuss is the "package-private" modifier. It is also called the "default" modifier, *because this modifier is never* *declared*. Instead, it is used as a *fallback* when no other visibility modifiers are declared.

A class, method or variable with this visibility is accessible from the `package` in which it is declared, **but from nowhere else**. In other words, for classes that reside in the same `package` as a package-private class, it's as though the class is `public`; however, for classes belonging to other packages, a package-private class acts as though it was a `private` class.

```java
package com.marcusbiel.javaBook.attributes;

class Name {
    /*
     * Package-Private Modifier for Class Name
     */
}

```
Example 2

This modifier is only sparsely used by Java developers, *without good reason*. Generally speaking, use the default level modifier whenever you need classes of the same `package` to use a method, but you don't want classes outside of this `package` to use the method. For example, imagine a class `Car` has a method `diagnose` that you want a class `Mechanic` of the same `package` to be able to use. But the sales-oriented car company you are coding for doesn't want class `Customer` to fiddle around with this method, because that would hurt its earnings.

In my opinion there is a flaw in the package-private modifier: Since there is no keyword, it is unclear whether a missing modifier is an error on the part of the programmer, or a *planned* package-private modifier. If you forget to put a modifier in, you are going to cause issues for your program down the road, but you won't know. There will be no warning from the compiler that there's a "missing modifier", since it is legal coding practice to leave it out. If you had wanted your class or members to be public, when you try to access them outside of the package you can't. Even more dangerous, is when you've forgotten to set a `private` modifier and months later, your method or variable is used somewhere else, without your noticing.

On the other hand, if you intended to use the package-private modifier, you're intentionally leaving the visibility modifier out. Another programmer might not realize this and try to "fix" your code by adding in a modifier that they assume you wanted. That's why I recommend that if you are *on purpose* using the package-private modifier (which in some cases is **very useful**), then leave a comment denoting your intent.

Coding Example

Now that you know about visibility modifiers, let's apply them to a coding example. First, we are going to create a new `@Test` method in our `PersonTest` class. This method will be called `shouldReturnNumberOfPersons` and will contain three objects of type Person named "person1", "person2", and "person3".

```java
package com.marcusbiel.javaBook;

import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class PersonTest {

    @Test
    public void shouldReturnNumberOfPersons {
        Person person1 = new Person();
        Person person2 = new Person();
        Person myPerson = new Person();
        assertEquals(3, myPerson.numberOfPersons());
    }
}

```
Example 3

Next we're going to use an `assertEquals()` method to check if the number of `Person` objects created is equal to 3.

Let's begin to write the code to make this method work in our `Person` class. In our `Person` class, I've created an instance variable of type int called `personCounter`. Then, in the default constructor, we will add `1` to `personCounter` each time this constructor is called. Logically, every time a `Person` is created this constructor is going to be called, so if we add `1` to `personCounter` each time the constructor is called, we count the number of `Person` objects we have created. We never initialized `personCounter`, but this should still work because the default value for an `int` is `0`. (If you'd like to learn more about default values, you can take a look at this chapter).

```java
public class Person {
    private int personCounter;

    public Person() {
        personCounter = personCounter + 1;
    }
}
```

Example 4

As an added note, there are actually three ways to add 1 to `personCounter`. The first is the way we did above. The second is:

```java
personCounter += 1;
```

Example 5

which can increment `personCounter` by any value we wish. The third option is the shortest, but only works if you want to increment by `1`:

```java
personCounter++;
```

Example 6

All three of these options take the value of `personCounter`, increase it by `1`, and then store that new value in a new version of `personCounter`. Now let's write our `numberOfPersons()` method to return `personCounter`:

```java
public static int numberOfPersons() {
    return personCounter;
}
```

Example 7

If we execute this code, our test fails because our `numberOfPersons()` method returned 1. Can you guess why?

Each time we created a new `Person` object, we stored the object and all its values into a separate `person` variable. Therefore, each time we create a new object, all of its instance variables are reset by the constructor and stored as part of a new object. So for each of our `Person` objects, the value of `personCounter` got initialized to `0`, and then incremented by `1`.

The Static Modifier

This brings us to our solution, the `static` modifier. As you might remember from the last chapter, the `static` modifier associates the method or variable with the class as a whole instead of with each individual object. Normally if you create a hundred `Person` objects, each will have its own `personCounter` variable, but with this modifier, all one hundred objects will share one common `personCounter` variable. This way, our `personCounter` will retain the same value, no matter how many `Person` objects we create.

First, we add this modifier to our `personCounter` variable and we're also going to add it to our `numberOfPersons()` method, as we should never have an instance method return a `static` variable and vice versa.

```
package com.marcusbiel.javaBook;

public class Person {
    private Name personName;
    private static int personCounter;

    [...]

    public static int numberOfPersons() {
        return personCounter;
    }
}

```
Example 8

By making the method and variable `static`, we can now accurately count and return the number of `Person` objects created. Our variable is associated with the class, and since it can't be accessed due to the `private` tag, we have the `public` method `numberOfPersons()` which allows outside code to access, but not modify, the value of `personCounter`.

Chapter 7
Development Tools

Java IDEs

Let’s start with something that I’ve discussed before, our Integrated Development Environment or IDE. The three most popular IDEs are Eclipse, IntelliJ IDEA, and Netbeans, in that order. Eclipse is free and open source, so it is very popular, especially among companies. I have been using Eclipse since 2002, but it’s starting to fall behind other major IDEs. I’m using this course as an opportunity to show off IntelliJ IDEA, which is becoming more and more popular. From what I’ve seen, IntelliJ IDEA is much more powerful compared to Eclipse, and unlike Eclipse, doesn’t require additional plugins to allow you to start using it. Also, Eclipse has many bugs that are frustrating to deal with and is infrequently maintained.

Finally, NetBeans is another free IDE that is also popular in corporate software development. It has great autocomplete and autoimport features. These three and many others are all valid choices, and I always feel that each individual should pick the IDE they are comfortable with. In my opinion, a seasoned developer should be at least somewhat familiar with *all* three. Here are the download links for the three IDES:

Eclipse
https://eclipse.org/downloads/

IntelliJ IDEA
https://www.jetbrains.com/idea/download/

NetBeans
https://netbeans.org/downloads/

Testing Frameworks

Next, let's talk about the different options for creating Tests. In all the tests I've used so far, I've used JUnit, which is a testing framework. Frameworks are a set of classes bundled together. JUnit is open source and free, making testing really easy. Most IDEs also have it preinstalled which makes it the most easily available choice.

Another Testing Framework is TestNG, which is very similar to JUnit. IntelliJ IDEA and Eclipse actually support both testing frameworks, so you can easily try both, but I'm not going to go into more detail about using TestNG, purely out of preference.

Maven

Another program I'd like to talk about is Maven. Maven is a build management tool; as the name implies it manages your build. Basically, a build is the process of turning your program into one file. Imagine that your program consists of hundreds of classes that you need to release or send to a client. You don't want to send each file separately, so you gather them in a package and turn that package into a single file. Before the original build management tool Unix's make, this was a manual task requiring various scripts to compile software, but now we have tools like Maven to do it for us! There are other build tools, but for this book I'm using Maven.

I highly recommend that you also read through the documentation on the website to understand how Maven works.

Text Editors

The final thing I'll talk about in this chapter is text editors. I've mentioned before in the book that you should start learning Java with a text editor. An IDE is great for a seasoned developer; it provides many features that can greatly assist programmers, such as warnings and error highlighting in your code, automatic code completion and tools to help with code refactoring. However, it also acts as a crutch and causes a beginner to avoid actually learning many things that they should at least be aware of while coding. Using a text editor, you can write code, save it, and build your project in the console, which is really all you need if you're learning Java for the first time.

For Windows, I recommend that you use Textpad. It is very simple to use, yet very powerful. For all other operating systems, use Sublime. Both offer features like code highlighting and advanced search and replace features. Again, I recommend that you use one of these text editors throughout this book as it'll help you retain more Java and focus on writing high quality code.

Chapter 8

Booleans and Conditional Statements

Booleans

A boolean is a primitive data type that stores one of two values, `true` or `false`. The name comes from the inventor, George Boole who discussed the idea of a `boolean` in great detail. In Java, we can use booleans to create conditions and execute blocks of code based on those conditions. To illustrate this idea, imagine a situation that most of us experience every day. When your alarm goes off in the morning, whether or not you went to sleep early could cause you to decide between getting up or pressing the snooze button. Whether you went to sleep early could be considered a `boolean` value, but for the code to decide which of these two actions you should respond with, you need conditional statements.

Conditional Statements

Conditional statements define conditions that are `true` or `false` and then execute based on whether or not the condition is `true`. Basically, conditions say, “If x is `true`, then execute y”. This logic is called an “if-statement”. Throughout all programming languages this if-statement is the most powerful and important statement, because it allows a program to execute differently every time. For the sake of demonstration, if we created a `boolean isMonday` and a `boolean  isRaining`, and set them both to `true`, we could then have an if-statement that checks this and then calls `drinkBeer()`, if both of them are `true`. After all, what else would you do on a rainy Monday? ;-)

```
@Test
public void demonstrateBoolean() {
    boolean isMonday = true;
    boolean isRaining = true;

    if (isMonday && isRaining) {
        drinkBeer();
    }
}
```

Example 1

Checking if both conditions are `true` is done using the “&&” symbol. If both conditions are true, then the `drinkBeer()` method will execute. We could also check if only one of the conditions are `true`:

```
@Test
public void demonstrateBoolean() {
    boolean isMonday = false;
    boolean isRaining = true;

    if (isMonday || isRaining) {
        drinkBeer();
    }
}
```

Example 2

The if-statement in Example 2 says, “If it’s Monday or it’s raining, then drink beer”. The `||`, called a pipe operator, defines an OR operator. Now, if it is raining or it is Monday, the `drinkBeer()` method will be executed.

Short Circuiting

One interesting aspect of compound if-statements is the idea of short circuiting. As we discussed previously, in an AND operator, if both conditions are `true`, the `drinkBeer()` method will execute. However, if the first condition is false, the if-statement will "short circuit" and will not execute the code without checking the second boolean. If the `boolean isMonday` was `true` and the `boolean isRaining` was `false`, you would excitedly note that it's Monday, but since it wasn't raining you still couldn't drink beer.

The same is true for a OR operator. If the first condition is `true`, then checking the second condition is unnecessary, since the code inside the conditional will execute whether or not the second condition is `true`.

Complex If-Statements

Our "if-statements" can also be made much more complex by compounding various conditions. The logic works by evaluating conditions in multiple levels of parentheses and then evaluating conditions in only one set of parentheses. The logic also checks conditions from left to right. Before you read on, see if you can figure out if the `drinkBeer()` method will execute in Example 3.

```java
@Test
public void demonstrateBoolean() {
    boolean isMonday = false;
    boolean isRaining = true;
    boolean isTuesday = true;
    boolean isSunny = false;

    if ((isMonday && isRaining) || (isTuesday &&
                                        isSunny)) {
        drinkBeer();
    }
}
```
Example 3

Ok, let's look at the first condition, "`isMonday && isRaining`" - that's `false`. After that you can see that we have a OR operator in between the two sets of conditions, so the if-statement must check the second condition. So let's do that: "`isTuesday && isSunny`". This is also `false`, because it is Tuesday, but it isn't sunny. Since neither condition is `true`, the entire statement is `false` and we can't drink a beer ;-)

Until you fully understand "boolean algebra" and have mastered using conditionals, continue using parentheses to enforce the order of execution you need safely. In short, a conditional is interpreted as follows:

1. Any conditionals inside parentheses
2. Any AND symbols
3. Any OR symbols

Unless you feel very comfortable with conditionals, you should surround all of your conditions in parentheses just to be safe.

The Else Statement

Now I'll introduce you to the counterpart of the "if-statement": the "else statement". Let's say it's not Monday, so we can't drink beer, but we still need to stay hydrated. We could say, "If it's Monday, drink beer; otherwise, drink milk."

```
@Test
public void demonstrateBoolean() {
    boolean isMonday = false;

    if (isMonday) {
        drinkBeer();
    } else {
        drinkMilk();
    }
}
```

Example 4

You might notice that the "else statement" doesn't have a condition. This is because the "else" executes in all cases where the "if" case doesn't apply.

The Else-If-Statement

If I were you, I'd get bored with drinking milk six days a week. But at the same time, I don't want to drink beer more than once a week. This is where the final conditional statement comes into play: the "else-if" statement. The `else if` evaluates a condition if the if-statement is `false`. You can also have multiple "else ifs" that execute if all previous statements are false. At the end of all these statements, you can have your "else" statement that still executes in all other cases, meaning that all of the other statements were `false`. Let's take a look at an example where on Fridays we drink water:

```java
@Test
public void demonstrateBoolean() {
    boolean isMonday = false;
    boolean isFriday = true;

    if (isMonday) {
        drinkBeer();
    } else if (isFriday) {
        drinkWater();
    } else {
        drinkMilk();
    }
}
```

Example 5

Using Conditionals with Other Primitive Data Types

Not only can we use conditional statements to check if a boolean variable is `true` or `false`, but we can also create a boolean using a condition, and evaluate that. For example, we could have two ints, i and j with the values 4 and 3 respectively. We can compare them using the following symbols:

Symbol	Meaning
>	Greater Than
<	Less Than
>=	Greater Than or Equal
<=	Less Than or Equal
==	Equal
!=	Not Equal

Example 6

You may notice that the operator for equals is a double '=' operator, rather than a single '='. This is because '=' already has a use: it is used for the assignment of values to primitive data types and for assigning Objects to reference variables. Therefore, to avoid confusion for both the programmer and the compiler, == is used to **compare equality.**

'!=' means 'not equal'. Generally, '!' in front of any boolean value will negate its value. So it follows that we'll read, '!true' as 'not true.', which is equivalent to false. We read '!false' as 'not false', therefore it will be equivalent to true.

If you take a look at the example below, you can see different ways that conditionals can be used to compare values. Obviously, since we know the values assigned to `i` and `j`, this isn't very helpful, but if these values were dynamically given as a method parameter, then these conditionals would be useful.

```java
@Test
public void demonstrateBoolean() {
    int i = 4;
    int j = 3;
    boolean areEqual = (i == j);
    if (i > j) {
        /* i is greater than j */
    } else if (!(i >= j)) {
        /* i is not greater than or equal to j */
    } else {
        /* i is equal to j */
    }

    if (areEqual) {
        /* i is equal to j */
    } else {
        /* i is not equal to j */
    }
}
```
Example 7

Applying Conditionals

You may not have the skills to create more complex conditional statements yet, but you can still apply conditionals to some useful examples. Let’s say we have our values of `i` and `j`, but now we want to increase the value of `j` if it is Monday. We won’t be incrementing in every case; we only do this if our condition is met. We can do other things too, all of which might be useful under certain conditions.

```java
@Test
public void demonstrateBoolean() {
    int i = 4;
    int j = 3;
    boolean isMonday = true;
    boolean areEqual = (i == j);
    if (areEqual) {
        i = 8;
    } else if (j > i) {
        j = i - 3;
    }

    if (isMonday) {
        j++;
    }
}
```
Example 8

Conditionals provide Java code with the means to respond differently depending on different outside conditions. They are extremely flexible and powerful tools that you will continue to use as you learn more and more Java.

Chapter 9

Basic Loops

Introduction

In this chapter I will discuss the use of loops in Java. Loops allow the program to execute repetitive tasks or to iterate over vast amounts of data quickly.

Background

Imagine that I want to send out an email to every subscriber of my newsletter. Now, if I had to manually send an email to every single subscriber, I would be kept busy for days. Instead, I want to write a program that helps me do this. So I write a program like this:

```
sendEmail(Tom);
sendEmail(Ben);
sendEmail(Christian);
[...]
sendEmail(Julia);
```

Example 1

For every new subscriber, I will have to extend my code and add a new "`sendEmail`" call. This isn’t much better than my manual approach before. To fix this, I want to group all my subscribers into a list and then tell my computer to uniformly send an email to each subscriber on my list.

That’s where “loops” come into play. They are a way to express, “I want to execute this code 999 times”, for instance.

For-Loop

The first loop I will discuss is the “for-loop”. It is called a for-loop because it tells the program, “Execute this loop FOR a certain number of times”. Our for-loop has three sections. The first section assigns and defines a variable, such as “`int i = 0`”. It uses this variable to iterate.

Note:
Generally, you can use any variable name, but one-letter variable names are commonly used in this context. This, at first, might seem counterintuitive and contradict the recommendation of using descriptive variable names. “`i`” in this context is okay, because one letter variables like ‘`i`’ or ‘`j`’ are commonly used for counter variables in for-loops. The letter ‘`i`’ is specifically used because it stands for the word “index”. ‘`j`’ is used in cases when you need two different indexes - just because ‘`j`’ is the next letter in the alphabet after ‘`i`’. Finally, the length of a variable name should be related to the length of the block of code where it is visible. When you have a large block of code, and you can’t directly see where this variable was defined, it should have a descriptive name. For-loops, however, are preferably short - in the best case just spanning over 3 lines of code - so “`i`” as THE index variable should actually be very descriptive in this case, and any other name than “`i`” or “`j`” for a simple index variable of a for-loop could actually even confuse other developers.

The second section defines how many times we want to execute the code inside the for-loop. In Example 2, we used ‘`i < 4`’. It is important to remember that it is commonplace in programming to start index values with 0, not 1. Since we want our code to iterate four times, we say ‘`i < 4`’ meaning that the code will execute when our ‘`i`’ value is 0, 1, 2, or 3.

The last section defines the incrementation of our variable. It can either increase the value or decrease it. In our example, we are increasing it by 1 using ‘`i++`’ which increases our ‘`i`’ value by 1. This occurs after the code block inside the for-loop is finished. If we were to print out our ‘`i`’ values inside the code of our for-loop, it would print ‘`0,1,2,3`’, not 4. ‘i’ would increment to 4 after our code runs for the fourth time, but since that does not satisfy the condition in the middle section of our for-loop, the code inside the loop stops executing.

```
public void shouldReturnNumberOfPersonsInLoop() {
    Person person1;

    for (int i = 0; i < 4; i++) {
        person1 = new Person();
    }
    assertEquals(4, Person.numberOfPersons());
}
```

Example 2

To restate the logic of the for-loop, initially ‘`i`’ is assigned a value of 0. After we execute our code inside the loop, ‘`i`’ is now incremented by 1, so now it has a value of 1. The condition we wrote in the middle section stipulates that ‘`i < 4`’ for our code to execute. Since that is evaluated to be true, the code executes again, increments ‘`i`’ by 1 and keeps repeating. The first time the condition evaluates to false is when ‘`i`’ has a value of 4. At this point the loop is exited and we will have executed our code 4 times.

You can also increment by numbers larger than 1. Say we incremented the previous example using “`i = i + 2`”, you would only create two person objects. This is because we are now only going through the loop twice, when `i = 0` and `i = 2`.

One of the things you should watch out for when creating loops is the possibility of an infinite loop. For example if our condition in Example 2 is "`i < 1`" and you continually decrease the value of '`i`' using "`i--`", the loop will go on forever, create lots of objects, and eventually will probably crash your PC. Each of the three sections of the for-loop is optional. Technically, you could even leave all three sections blank and provide only two semicolons. Be careful, though, because that way you would create a loop that would never terminate (an "infinite loop").

While-Loop

Our second loop is the while-loop. A while-loop allows the code to repeatedly execute WHILE a boolean condition is met. The basic syntax for a while-loop is shown below:

```java
while (condition) {
    /* Code that executes if the condition is true */
}
```

Example 3

The while-loop is useful when you have a condition that is being dynamically calculated. We could create the person object inside a while-loop, by dynamically changing the value of a variable '`i`'.

```java
@Test
public void shouldReturnNumberOfPersonsInLoop() {
    Person person1;
    int i = 0;

    while (i < 4) {
        person1 = new Person();
        i++;
    }
    assertEquals(4, Person.numberOfPersons());
}
```

Example 4

In Example 4, we again create four person objects. Again, we start off with ‘i’ valued at 0 and increment it by 1 while ‘i<4’. When that condition becomes false and i’s value is 4, the loop is exited.

Do-while Loop

The third type of loop we’ll look at in this chapter is the do-while loop. A do-while loop executes the code at least once and then repeatedly executes the block based on a boolean condition. It functions like a while-loop after the first iteration, which happens automatically. In Example 6, we create the four Person objects using a do-while loop.

```java
@Test
public void shouldReturnNumberOfPersonsInLoop() {
    Person person1;
    int i = 0;
    do {
        person1 = new Person();
        i++;
    } while (i < 4);

    assertEquals(4, Person.numberOfPersons());
}
```
Example 5

There is also a fourth type of loop, the for-each loop. The loop you should choose depends on the logic you need. All four of the loops are equivalent in terms of functionality, but each loop expresses the logic in a different style. You should always try to use the loop that is easiest for someone reading your code to understand and that makes the most sense in context.

Chapter 10

For-Each Loops

In this chapter I will be discussing the for-each loop. The for-each loop is a simplified loop that allows you to iterate on a group of objects like arrays.

Array

The array is an extremely powerful tool that allows you to store multiple objects or primitive data types in one place. You can read more about it in the chapter dedicated to arrays. For now I'll just give you enough to understand the for-each loop. As an example of an array, imagine an array "persons" that internally holds an entire group of persons. Let's see how we can statically iterate over a `persons` array that can hold ten persons, using a for-loop that I introduced you to in the last chapter:

```
for (int i = 0; i < 10; i++) {
    persons[i].helloWorld();
}
```

Example 1

Each iteration of the loop will increment the variable integer primitive i, that will successively be used to access each element in the persons array.

As you can see, the loop is not easy to read or even to understand. Also, we have to put the size of the array in the loop upfront. As we will see later, there is a better, dynamic way to get the array's size, but that won't make it easier to read.

The For-Each loop

A for-each loop essentially works like a simplified for-loop. A for-each loop uses a simpler, more readable syntax. It does all the dirty work behind the scenes.

```
for (Person person : persons) {
    person.helloWorld();
}
```

Example 2

As you can see in Example 2, we don't need to put in size of the `persons` array. The for-each loop will conveniently retrieve it for us. Also, we don't need to index and fiddle around with array cells. For each iteration over the array, it will retrieve the current person object for us, and we can conveniently call the `helloWorld()` method on each `Person` object.

However, keep in mind that the simplicity of the for-each loop comes at a price. There is no such thing as a free lunch! It is not as dynamic as a for-loop, as it always indexes through every spot in the array. With the counter variable "i" missing, you can't look at every other cell, or every third cell, or the first half of the cells. Of course - you *could* add an additional counter variable - but that would render the benefit of the simplified for-each loop useless. Don't even think about it!

The for-each loop is **the perfect loop** to use when you simply need to go through every element of a container. Actually, this is the most common use case for a loop, which makes the for-each loop the most commonly used loop in Java!

Chapter 11
Arrays

An array is a special type of object in Java. Imagine it like a container that can hold a number of primitive data types or objects. It stores each one in its own ‘compartment’ and allows you to access them by providing the location of the ‘compartment’, called an index.

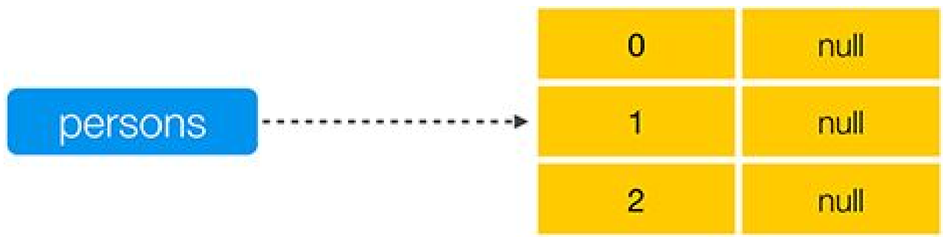
Let’s say we wanted to create an array of Person objects. You would do it like this:

```
Person[] persons = new Person[4];
```
Example 1

The first part `Person[] persons` defines a Person-array reference variable named `persons`. The second part, `= new Person[4]` creates a Person-array object and assigns it to our reference variable `persons`. `[4]` indicates that the array will be able refer to a maximum number of four Person objects. You can't change the size of an array after you've created it. So once you've initialized the array, that's it, it's stuck at that size.

When instantiating an array of objects,such as a Person array in our case, you create **one** array object, acting as container, *offering space to store references* to the actual objects.

After instantiation, your array will be empty. Each cell will contain `null`, which means it is not referencing any object, as illustrated by Example 2. If we tried to access the reference at this position we would get an error once we ran the program. As a side note, the proper term for an error like this is an 'exception' (you can read more about Exceptions in Java here).

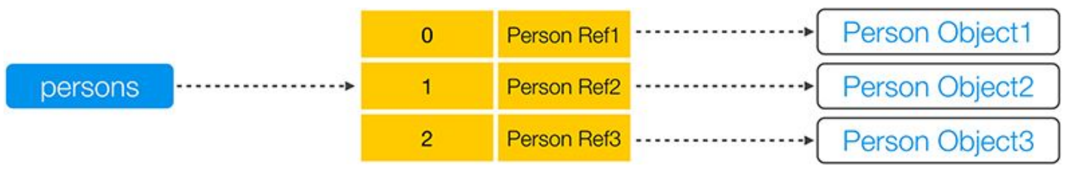


Example 2

Now, let's fill our array. When we do this, we're filling it with reference variables to `Person` objects.

```
persons[0] = new Person();
persons[1] = new Person();
persons[2] = new Person();
persons[3] = new Person();
```

Example 3



Example 4

Syntactically, you could also put the square brackets of the `persons` array variable after the variable declaration, as Example 5 shows you:

```
Person persons[] = new Person[4];
```

Example 5

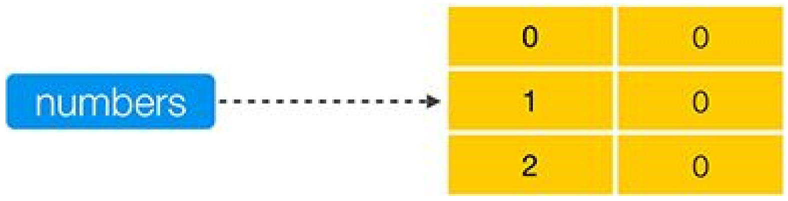
The code in Example 5 is a flaw of the Java Programming Language and it's highly recommend never to do that. Reading the code is less clear -

since if you only read part of the line, you could come to the wrong conclusion that it creates a reference variable of type Person. Put the square brackets **directly after** the Person class, to clearly indicate that this is a reference variable of type Person array, and **not** Person.
We can also create an array to store a primitive data type like an `int`:

```
int[] numbers = new int[3];
```

Example 6

You are probably not used to seeing `new` in front of a **primitive** int. However, this is syntactically correct; it creates an array object that can hold three values of type `int`, **and not a primitive type**. However, while an array of objects stores spaces for references, an array of primitives stores the primitive values themselves. Therefore, after initialization, the cells of the array will be pre-filled with `0`, the default value for an `int`, and not `null`, as Example 7 illustrates:



Example 7

Remember, whether it is storing object references or primitive data types, an array is an object. Since arrays are objects, you can call methods on them. For example, you can call `myInts.toString()` on the array in Example 6. You can also access the array's `public` attribute, `length`, that tells you the static length of the array. You might remember that in a previous chapter I talked about why you should make your instance variables `private` inside a class when using an object-oriented approach. This is yet another flaw in Java, a place where Java itself unfortunately violates basic object-oriented principles.

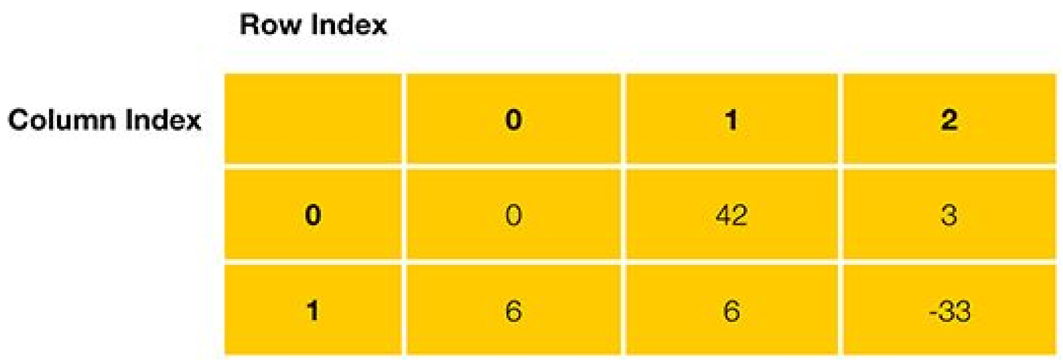
Multidimensional Arrays

You can also create a multidimensional array. A multidimensional array is an array of arrays:

```
int[][] numbers = new int[2][3];
```

Example 8

Conceptually, you can visualize a two dimensional array as a table with row and column indexes, as Example 9 illustrates:



Row Index

Column Index

	0	1	2
0	0	42	3
1	6	6	-33

Example 9

Each cell of the first array forms the rows of the table. Each row contains yet another array, where each array forms the cells of each row. Arrays of more than two dimensions are less common, but easily possible, as Example 10 shows you:

```
Person[][][] persons = new Person[2][4][3];
```

Example 10

Multidimensional arrays are read from left to right, with each value acting like a coordinate for each primitive value or object reference. The topmost array in the hierarchy is the leftmost array. It is storing the arrays that are referenced by the subsequent set of square brackets.

Shorthand Notation for Arrays

Besides the way I explained it in Example 1, there is an alternate way to create arrays. The most basic form of it is this:

```
Person[] persons2 = {};
```

Example 11

The code in Example 11 creates an empty array, with `person2` referencing it. The curly braces surround every object that we are putting into the array. This array is size `0`, which isn't really useful in any sense, but technically it's possible.

We can also use this method to actually fill an array without setting the size. When you create the array, you put in as many objects and/or `null` values as you want and that decides how large the array will be.

```
Person[] persons = new Person[3];
```

Example 12

Indexing in an array

Let's return to the array of Person references that we created before. Let's fill it with person objects. First we have to index to the 'compartment' in our array by putting its index value in square brackets like below:

```
persons[0] = new Person();
```

Example 13

Arrays start indexing at `0`, so our four 'compartment' array has indexes at `0`, `1`, `2` and `3`. We could, as we did in Example 3, create four references and assign them each to new objects. Alternatively, we can assign our new reference variables to existing objects, or even to objects that other cells of the array are referencing. Obviously, in such a simple example, it might not be necessary to introduce these complexities, but I'm doing it to demonstrate the concepts.

```
@Test
public void demonstrateArrays() {
    Person[] persons = new Person[4];
    persons[0] = new Person();
    persons[1] = new Person();
    persons[2] = persons[1];
    Person myPerson = new Person();
    persons[3] = myPerson;
}
```
Example 14

For Loops and Arrays

Loops and arrays are always a couple. For an array to be efficient it needs a loop, and loops are well equipped to work arrays. Let's say we wanted to index through every single 'compartment' in our array and apply the same code to each one. We can create a for-loop to do this for us. Our variable `i`, will be used as the value for our index. Now, inside this loop, we could create the objects of type person and access the objects.

```
@Test
public void demonstrateArrays() {
    Person[] persons = new Person[4];
    for (int i = 0; i < 4; i++) {
        persons[i] = new Person();
        person[i].helloWorld();
    }
}
```
Example 15

Inside this loop we could utilize each person in the array and have them call the `helloWorld()` method. Loops are an extremely convenient way to repetitively execute an operation on each cell of the array, without having to duplicate the code.

```java
for (int i = 0; i < persons.length; i++) {
    persons[i] = new Person();
}

Person myPerson = new Person();

Person myPerson2 = null;
Person[] persons2 = { persons[0], null, myPerson,
                                           myPerson2 };
```

Example 16

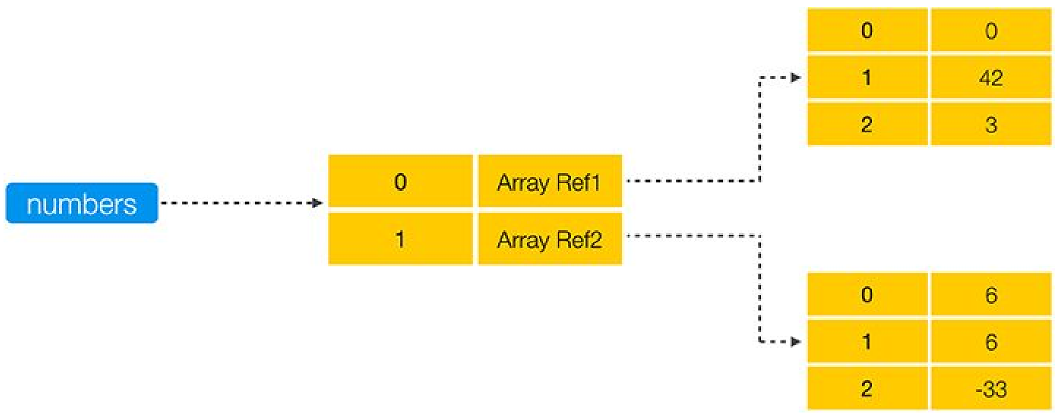
Shorthand Notation for Multidimensional-Arrays

You can also utilize this shorthand notation for multidimensional arrays. To do this, you surround each ‘inner array’ with curly brackets and separate each set of values with a comma:

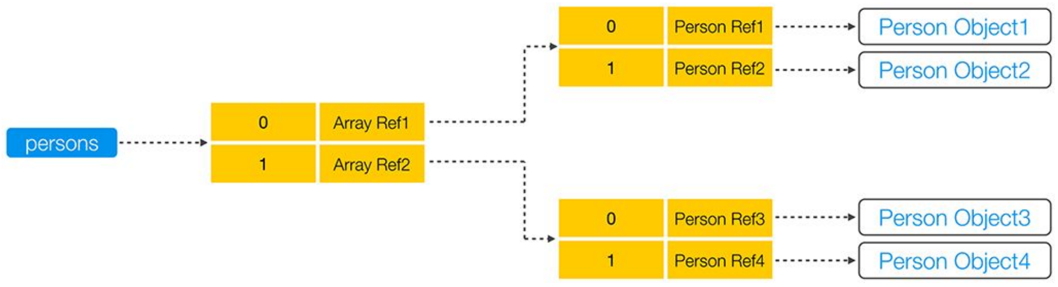
```java
int[][] numbers = {
    { 0, 42, 3, },
    { 6, 6, -33, },
};

Person[][] persons = {
    { person1, person2, },
    { person3, person4, },
};
```

Example 17

Examples 18 and 19 show how this will look in memory:



Example 18



Example 19

We can also use two for-loops, commonly known as a nested loop, to index a two dimensional array:

```java
@Test
public void demonstrateTwoDimensonalArrays() {

    Person[] persons = new Person[4][4];
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; i++) {
            persons[i][j] = new Person();
            person[i][j].helloWorld();
        }
    }
}
```
Example 20

Utilizing a For-Each Loop

As I mentioned in the last chapter about the For-Each Loop, for-each loops are extremely useful when applied to arrays. Utilizing the for-each loop we can iterate through every object in the array without having to know the length of the array.

```java
for (Person person : persons) {
    /* do something to each object in the array */
}
```
Example 21

Now we've learned in full about how the for-each loop and the array work, so I've shown you one of the most powerful pairings in Java. You can use arrays to store objects or primitives and iterate through them with loops- an extremely efficient and clean way to code.

Chapter 12

Enums

In this chapter, I will talk about enums. Enums are sets of constant values that can never be changed.

The Final Tag

To illustrate why they are useful, I'm going to start by introducing the concept of constant values in Java and talk about their flaws. Let's say we wanted to save an array, as is, so that any time it is used in the future, we know for sure which references are in which places. We could do this using the variable modifier "`final`". A variable declared as `final` is locked in and cannot have a new value assigned to it. For example, if we made `persons2 final`, and then tried to set it to `null`, the code will not compile.

```
final Person[] persons2 = { new Person(), null, null
                                                   };
 persons2 = null; /* Compilation Error */
```

Example 1

This could be very useful if you have a set of values that you want to stay constant. One of the things you could do is create a set of state values. State values, as the name implies, tell us what state our program is in. Let's say our three state values for our program are "`PENDING`", "`PROCESSING`", and "`PROCESSED`". We could place these `String` values into an array and add the "`final`" tag to preserve them. We're also going to make the array `static` so that it's shared among all the objects. We'll call this array `MY_STATE_VALUES`. Here's what this would look like:

```
private static final String MY_STATE_VALUES[] = {
                    "PENDING", "PROCESSING", "PROCESSED" };
```

Example 2

You might notice that my variable name does not follow the default "camel case" convention, but instead is composed of words in uppercase letters connected by underscores. This is because our "variable", isn't really variable. It's `static` and `final`. You just want it read, the reference variable can never be reassigned, nor should the object it references be modified; so to represent this difference, the naming convention is different. For `static final` variables, always use capital letters connected by underscores.

Note:
Although a "`final`" array reference cannot be reassigned to a different array, the array that it references can still be modified. So, from our previous example, "`MY_STATE_VALUES[2] = "DONE";`" would not cause any errors. For this reason, you should be careful when relying on array values to stay constant.

Going back to Example 2, because the `MY_STATE_VALUES` array reference is `private` and uses the uppercase naming convention to communicate our intent, it might be acceptable to assume that the array's values will never change. However, you should always be aware of the potential dangers when making assumptions about mutable values.

While we can't modify our array reference variable, we can iterate through it using a for-each loop. Inside this for-each loop, we could compare our states to a certain value and call some method when the value and the state match.

Comparing Strings

Since `String` instances are objects and not primitive values, the equals operator ('==') might not work as expected. Instead, `String` has a method `equals` to check for equality. We call it on one String instance and pass the second in as a parameter. For example, "`someString.equals(otherString)`". The `equals` method returns the `boolean` value `true` if the two String instances are equal, `false` otherwise. If you're interested in learning more about this, you should check out the chapter on Identity and Equality in Java, and my blog post about Hashcode and Equals.

In a for-each loop, I'm going to create three if statements, one for each state in our `MY_STATE_VALUES` array.

```java
for (String state : MY_STATE_VALUES) {

    if (state.equals("PENDING")) {
        /* call a method */
    }
    if (state.equals("PROCESSING")) {
        /* call a method */
    }
    if (state.equals("PROCESSED")) {
        /* call a method */
    }
}
```

Example 3

So this looks like it would work great. We iterate through everything in the `MY_STATE_VALUES` array and the three methods are all called in the proper order. However, as I mentioned before, the `MY_STATE_VALUES` array is modifiable. If one of the state values was changed to “blah” right before our if statements, that would be a problem. Don’t worry, though, there’s a solution: the enum.

Enum

An enum allows us to define an enumeration which means a complete ordered list of a collection. This enumeration cannot be modified once created. An example of an enumeration in real life would be a traffic light. It has three distinct values: green, yellow and red; no matter what happens you can never change these values. This could be extremely useful if applied to something like the set of states we talked about in Examples 2 and Example 3. Let’s create an enum named `LoggingLevel` which stores those three states:

```java
package com.marcusbiel.javaBook;

public enum LoggingLevel {
      PENDING, PROCESSING, PROCESSED;
}

```

Example 4

As you can see in Example 4 above, an enum is declared similarly to a class, but with the “`enum`” keyword replacing “`class`”. Inside the declaration, we list the enum value names, separated by commas. For simple enums like `LoggingLevel`, the ending semicolon is optional, but it is good practice to include it. And that’s it! We’ve defined our enum.

An enum is useful when we want to have a list of items, such as states or logging levels, that we want to use in our code, but we don’t want them to be changeable. While the references contained in the array object can be modified, the enum constant values cannot.

Uses of Enums

An enum consists of one or more constant values. You can create reference variables that point to enum values and print them. You can also check the equality of enum values using '==' instead of the `equals` method. This is because all enum values are unique and constant, and we know that no enum value is ever copied. Therefore, checking for identity is the same as checking for equality. If you're interested in the difference between these two, check out the chapter about Identity and Equality in Java.

```
LoggingLevel currentLoggingLevel =
                                     LoggingLevel.PROCESSING;
System.out.println("current LoggingLevel is: " +
                                         currentLoggingLevel);
System.out.println(currentLoggingLevel ==
                      LoggingLevel.PROCESSING); /* true */
```
Example 5

You can also search through your enumeration to find a specific enum value using a `String`. For example:

```
LoggingLevel currentLoggingLevel =
                      LoggingLevel.valueOf("PROCESSING");
LoggingLevel illegalLoggingLevel =
                      LoggingLevel.valueOf("processing");
                                        /* Error is thrown */
```
Example 6

Please note that the functionality displayed in Example 6 is **case sensitive**, meaning that the second line of code would cause an exception to be thrown when the code is run. For more information, check out the chapter on Java exceptions.

Adding State to Enums

Another feature of enums is that we can add fields to enums and construct each value with a specific state. For example, let’s add a primitive field to `LoggingLevel` and assign each level a different number. If we create a private variable `i`, and set it to the number we send our constructor, we can assign a meaningful value to each logging level. Since both the constructor and the value should only be visible inside the class, I’m going to make them `private`.

Since the enum will call the constructor on its own, you’re not allowed to construct enum values. This is why we make our enum constructor `private`.

If you want to show off and appear smart, you should know that you technically can also make an enum constructor package-private, however, it's about as useful as the human appendix. Using the package-private modifier doesn’t help because you still can’t construct an enum from outside.

```java
public enum LoggingLevel {
    PENDING(1), PROCESSING(2), PROCESSED(3);

    private int i;

    private LoggingLevel(int i) {
        this.i = i;
    }
}
```
Example 7

Applying Enums to our Example

Now we can go back to Example 3 and start using our newly-defined enum instead of the original state array. To get all the values of our enum we use the `values` method, which returns all of our enum values in an array.

```java
for (LoggingLevel state : LoggingLevel.values()) {
    if (state == LoggingLevel.PENDING) {
        /* call a method */
    }
    if (state == LoggingLevel.PROCESSING) {
        /* call a method */
    }
    if (state == LoggingLevel.PROCESSED) {
        /* call a method */
    }
}
```
Example 8

We can actually shorten our code even further if we add something called a *switch* statement to replace the *if* statement we used in Example 5, but we'll do that in the next chapter which discusses the Java switch statement in detail.

Chapter 13

Switch Statements

The switch statement is another type of conditional. It is similar to an if-statement, but in some cases it can be more concise. You don't have to think about the exact meaning just yet, as we will get to it in the example below:

We're going to use this series of status enums (shown in Example 1), from the chapter about enums, for our examples. But instead of using a series of if-statement branches, we use switch to operate on the enum's values.

```java
package com.marcusbiel.javaBook;

public enum LoggingLevel {
    PENDING, PROCESSING, PROCESSED;
}

```

Example 1

The switch statement is shown in Example 2. Syntactically, it is similar to an if-statement, but instead of writing `if`, you write `switch`. Inside of the switch we write the variable we are trying to compare, in this case, state. Each case is a potential value of switch we are trying to compare to. That is to say, `case PENDING:` is roughly the same as saying: `if (status == Status.PENDING)`. The inside of each case is similar to inside the curly brackets of an if-statement. To terminate a case, you can type the word `break`, or write a `return` statement. If you don't have either of these, the code 'falls through', an idea that I'll explain in the next section.

If you only have one value, then an if-statement is more concise, as switch statements have the added overhead of writing `switch (status) {...}`. However, a switch statement is preferable when you iterate through multiple values with short code blocks, such as a method call, inside in each condition. The more conditions you have, the shorter your switch statement is compared to the equivalent branches of if-statements.

```java
LoggingLevel state = LoggingLevel.PENDING;

switch (state) {
    case PENDING:
        onPending();
        break;
    case PROCESSING:
        onProcessing();
        break;
    case PROCESSED:
        onProcessed();
        break;
}
```
Example 2

Another alternative to avoid falling through is to return a value inside the switch statement to leave the surrounding method, as shown in Example 3. This exits the method, and therefore doesn't fall through to the rest of the switch statement. In this case, the return value of our process method is `String`. If no case is met, we will return a default `String` instead.

```java
private String process(LoggingLevel state) {
    switch (state) {
        case PENDING:
            return pendingAsString();
        case PROCESSING:
            return processingAsString();
        case PROCESSED:
            return processedAsString();
    }
    return defaultString();
}
```
Example 3

Falling Through

The code in Example 4 is an example of a switch statement lacking a **break statement** for the case `PROCESSING`. Given a `PENDING` status, we would call the pending logic, leaving the switch altogether with the break statement. This works as intended.

```java
switch (state) {
    case PENDING:
        onPending();
        break;
    case PROCESSING:
        onProcessing();
    case PROCESSED:
        onProcessed();
        break;
}
```
Example 4

However, given a PROCESSING status, we end up calling both the `onProcessing()` and the `onProcessed()` method. In other words, the `PROCESSING` case **falls through** to the `PROCESSED` case. In this example, both cases have their own independent code within, so it probably wasn't intended to fall through.

Falling through can be useful though when you want two cases to execute the same code. Since there's no way to say "and" in a switch statement, to avoid duplicate code you could fall through in the first case and then write the code in the second (See Example 5).

```java
switch (state) {
    case PENDING:
        onPending();
        break;
    case PROCESSING:
        /*
         * falling through
         */
    case PROCESSED:
        onProcessingOrProcessed();
        break;
}
```

Example 5

Switch statements that fall through are a big source of bugs and are very dangerous as the lack of break statements can be difficult to spot. If you do intend to have your switch statement fall through, leave a comment saying so, as if to tell the next programmer, "this switch-case is falling through on purpose." - Otherwise he might be tempted to "fix" it.

The Default Clause

The syntax for switch statements doesn't require a case for all possible values. You can leave cases out. When no matching case is found, our switch statement finishes without doing anything. An alternative to not doing anything when no cases match is to add a default clause, as I did in Example 6, which matches when none of the other case clauses apply.

```java
switch (state) {
    case PENDING:
        onPending();
        break;
    case PROCESSING:
        onProcessing();
        break;
    case PROCESSED:
        onProcessed();
        break;
    default:
        onDefault();
}
```
Example 6

The same rules and pitfalls of falling through apply here. On the surface, most switch statements with a default clause look innocent enough until you consider what can happen when the next guy comes along and adds a new value or two to our existing Status enum. Let's say he adds the ERROR value. Not knowing our enum's use in one or more switch statements, he would inadvertently cause unintended matches on the default clause due to the lack of an ERROR case! We usually don't want that, so I recommend that you either not use the default clause, or better yet, have it throw an error. (If you want to know more about this, read the chapter on checked and unchecked exceptions.)

A Few Caveats

Don't overuse switch — and by extension — don't overuse if-statements. These conditional statements, when overused, are infamous for overly complex code, often written with nested if, switch, or even for-loop statements.

I have seen too many codebases with methods having code margins up to eight levels or more deep, and almost always the culprits are poorly-organized, tedious, deep litanies of conditional statements and the blocks they contain. These are relics of the past eras of procedural programming—a style of programming that builds upon procedures which makes you think like a machine. But humans don't usually think like this. The cool thing about object-oriented programming is that it's very close to how humans think. This allows us to talk to business guys, and when done properly, makes our code read very nicely.

The switch statement is not object-oriented. In Object-Oriented-Programming (OOP), there are some corner cases where it fits the bill as it is quick and easy to use. These are often instances where you would only have very few cases making up a small switch statement. However, there are too many instances where people would write hundreds of lines of code amounting to *one* long, crazy, hard-to-read switch statement. There are object-oriented techniques for reducing the complexity of such statements that help us express the same meaning. For example, we could represent the `PENDING` status with an object, the `PROCESSED` state with another, and so on. I won't discuss this in any more detail because these ideas are beyond the scope of this chapter; in summary, stick to only using switch statements for a very small number of cases.

Chapter 14
Logging

The term *logging* originates from ancient times when a ship's captain used to write down everything that was happening with the ship. (For example, the speed the ship was going and what course the ship was on.) The book he wrote this information into was called a logbook. The *log* part of the word came from a special tool the sailors measured the ship's speed with, which consisted of a reel of string with a log tied onto one end.

As a programmer, you are the "captain" of your code. You want to "write down" – or "log" – events that occur while your program is running. For example, you could print what is going on to the console, to a file or by email. You want to monitor things such as if the system has been hacked, what methods your code is calling, and how efficiently it's working. You could then later go through your logging file and analyze the relevant information to help you improve and debug your code.

Logback

For many years, the most prominent logging framework was Log4j. This has changed. At the moment, LOGBack is used because it's more powerful. LOGBack was developed as an improvement on Log4j by the same developer who created both, Ceki Gülcü.

How do we start logging? First of all, we configure the dependencies using Maven. The configurations requires dependencies org.SLF4J, logback-classic and logback-classic. This is the implementation of SLF4J and is the core code of LOGBack, therefore, we need these three dependencies to get logging into our code.

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.5</version>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-core</artifactId>
    <version>1.0.13</version>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.0.13</version>
</dependency>

```
Example 1

Next, under the resources directory (`src/main/resources`), there should be a file called `logback.xml`, in which we can define certain things, such as how and where logging should be done (see Example 2). In this case, giving the class `ConsoleAppender` will allow the code to be logged directly to the console. There are also other implementations for logging that allow us to log into a file, into a database or into an email (to name just a few). We can also provide our own implementation for new logging sources that don't exist.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="true">
    <appender name="STDOUT" class="ch.qos.logback.core.
                                      ConsoleAppender">
        <encoder>
            <pattern>
                %d{HH:mm:ss.SSS} [%thread] %-5level
                                 %logger{35} - %msg %n
            </pattern>
        </encoder>
    </appender>

    <logger name="com.marcusbiel.javaBook"
                                        level="debug"/>

    <root level="INFO">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```
Example 2

In this case, we are using the `ConsoleAppender`, so we know the log is being written to the console. But how and when is a message written to the console? You should read the documentation for details, but, briefly, the printed pattern shown in Example 2 gives the time when the event occurred, the thread, and the logging level. The logging level is a sort of “grouping mechanism”. We have different levels of logging, which, to some extent, indicate a problem’s severity. Logging level `DEBUG`, as the name implies, is used for debugging purposes. Debugging is the process of finding and resolving bugs in your code. `DEBUG` should be used infrequently, since for debugging we could use the debugger included in an IDE or we could write tests. There are also `INFO` and `ERROR` logging levels. Logging level `ERROR` is used if there is an error in the code, and `INFO` is used for general information like for a server or application starting or stopping.

In the `logback.xml` file (under `src/main/resources`), we specify that we want to log on level `DEBUG`, and we provide a package name, which will specify that we are enabling logging for classes under this particular package. We have mentioned the `com.marcusbiel.javaBook` package, but we can extend that and log only for specific classes, for example `com.marcusbiel.javaBook.CarService`. For the level, we could put `ERROR` instead of `DEBUG` which would mean “Log it only if it’s an error.” We could also type `INFO`, which would mean “Only log it if it’s information.” `INFO` includes the info level errors, and `DEBUG` includes all three levels - debug, info and error. Different levels make the logging mechanism more flexible. Different loggers for different packages could be used as well. For example, we could have a different package like `com.marcusbiel.javaBook2`, and provide a different logging configuration for it. We could then have one package on `DEBUG` and the other one on `ERROR`. We must also set a tag for the root level, which is generally set to `INFO` level.

First of all, let’s go to our `CarService` class and add the logging details, as shown in Figure 3. We need to `import org.slf4j.Logger`. SLF4J knows to access LOGBack and it will find the dependency of LOGBack classic in the Maven configuration file. We also `import org.slf4j.LoggerFactory` (A factory is another type of design pattern, by the way. In short, factories allow you to centralize the place of object creation, allowing for cleaner code and preventing code duplication).

```java
package com.marcusbiel.javaBook;

import org.junit.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class CarService {

    private final Logger log =
                 LoggerFactory.getLogger(CarService.class);

    public void process(String input) {
        if (log.isDebugEnabled()) {
            log.debug("processing car:" + input);
        }
    }
}

```

Example 3

In the `CarService` class we create a `private final Logger` and call it `log`. We then say `LoggerFactory.getLogger(..)`. This is generally how you define a logger. For now, we don't need to worry about how the object is created, as the factory is smart and does all the necessary work on its own. Inside the constructor we pass the Logger constructor the name of the same class in which it was defined, i.e. `CarService.class`. We used the if-condition check to make sure that our logger was in debug mode before we logged, which is good practice, because otherwise your debugging could result in a decrease in performance. We would be creating strings without actually logging them, which would be using unnecessary hardware resources.

Example 4 shows the test that will use the logging service. Now the logging can be used in the `process()` function of the class `CarService`.

```java
package com.marcusbiel.javaBook;

import org.junit.Test;

public class CarServiceTest {

    @Test
    public void shouldDemonstrateLogging() {
        CarService carService = new CarService();
        carService.process("BMW");
    }
}

```
Example 4

The input to the `process()` method is given as “`BMW`”. This will create a `String` that will be concatenated with “`processing car:`”, and will be logged.

In this scenario, it’s not so bad to use logging without the check, as all that would happen is that the two strings’ arguments would be concatenated. But consider that the input string is of infinite length. This concatenation of two strings, as silly as this may sound, can take a lot of time. Here we just have a very simple, small program with just one debug line. Imagine a thousand users concurrently calling this method, and it taking 200 milliseconds per method call. 200 milliseconds might not seem like a lot of time, but if the method is called thousands of times, this could use a lot of processing power on a server or a PC, leading to a drop in performance. We only want to concatenate the strings and prepare them for logging *if* logging is enabled. If we call a logging method and logging is not enabled, without checking first, the strings will be concatenated, *but not written to the log*. As a result, the concatenation would have been done unnecessarily. This is something we would like to prevent, and is the reason why we use `isDebugEnabled`, `isInfoEnabled`, or `isErrorEnabled`, and so on.

Even though this String concatenation logging is already fairly simple with Log4j and SLF4J, there is a smarter, simpler way, as shown below in Figure 5. We can put in curly braces in place of the variables, replace the “+” sign with a comma and then write the input String. The framework will check if the user wants to log on debug level. If so, it will take this input, convert it to a string, if it’s not a string already, and concatenate both strings. Like in Example 3, the strings won’t be concatenated unless the system is in debug mode, which helps to keep your program as efficient as possible. Here, however, we don’t have to use a conditional to check this since the debug function internally concatenates the String. This is stylistically better because it takes up fewer lines, while still providing the same functionality and protection from inefficiencies.

```java
public void process(String input) {
    log.debug("processing car: {}", input);
}
```

Example 5

Here we have the input, given from the process method, and we want to print ”processing car:” concatenated with the input String. In our test we put BMW as the input String. In our configuration file, logback.xml, we set the com.marcusbiel.javaBook package’s log level as DEBUG, and the root level as INFO. Let’s run the test. On level DEBUG, the timestamp, the package, the class CarService and the String “processing car: BMW” were logged on the console.

Now, let’s try deactivating the debugging logger. Let’s say we only want to log on ERROR level for the package com.marcusbiel.javaBook. We make this adjustment in logback.xml. Let’s execute it again. There will be no BMW logged. Now let’s change the root level to be DEBUG, and change the package, for example com.marcusbiel.javaBook2. We’re not in com.marcusbiel.javaBook2, we are in the package com.marcusbiel.javaBook, which means that we should log on the root level which is DEBUG. We would expect the test to log again. Let’s try it. It logs: “DEBUG com.marcusbiel.javaBook.CarService –

processing car: BMW". Even though we aren't in the package `com.marcusbiel.javaBook2`, the root level is defined as `DEBUG`, which is why the `DEBUG` message is shown.

In the `logback.xml` file, there is a `ConsoleAppender` attribute in the appender tag (see Example 1). `ConsoleAppender` is a type of appender that logs to the console which we can also see in our IDE. If we can run this from a terminal, we will see it directly within the terminal.

Logger Documentation

If you look in the Logger documentation, in the section "typical usage pattern", the example code they show (see Example 6 below) will be very similar to what we did in our example. You can log with `logger.isDebugEnabled()`, as shown before, or use the better alternative of the curly braces. They are very similar, but the first way uses three lines of code that clutter your code, and the other way uses only one line, which makes your code cleaner and simpler. Logging is important, but it should not get in the way of what you are trying to achieve with your code. Be sure to read through the documentation for a better understanding.

```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Wombat {
    final Logger logger =
                  LoggerFactory.getLogger(Wombat.class);
    Integer t;
    Integer oldT;

    public void setTemperature(Integer temperature) {
        oldT = t;
        t = temperature;
        logger.debug("Temperature set to {}. Old
                        temperature was {}.", t, oldT);
        if (temperature.intValue() > 50) {
            logger.info("Temperature has risen above 50
                                              degrees.");
        }
    }
}

```
Example 6

Logging on Different Levels

Now we will look at logging on different levels, for example `ERROR`. We're still on `DEBUG` level, and `ERROR` is included within `DEBUG` because `DEBUG` is the most specific logging level. In the traces, we would expect to see `ERROR`, not `DEBUG`, as a prefix for the printed log line, in order to differentiate how important some logging events are. We might log, for example, "Some error occurred", in the case of an error. In our example, "`processing car:`" is not an error, therefore, we should not log that on `ERROR` level. We can also log on `WARN` level, in case of a warning. We would use *warning* for something very close to an error, but where it is not critical to the operation of the system and no one has to physically intervene in any way. `INFO`, for example, could be used when a server or a program starts up, so that the system admin's team knows that everything is going well.

Now, let's set the root level to `ERROR`. The package that we're not using `com.marcusbiel.javaBook2`, as well as the root level are now on the `ERROR` level. They are printed to STDOUT, which is the name I gave to the appender. When running that, we expect not to see any logging. We will see the info from LOGBack, but the logging itself has gone. If we change the package back to `com.marcusbiel.javaBook` we should see the log message again.

Appenders

Before I end off, I would also like to quickly go over Appenders. If you're interested in reading more, you can check out the appender documentation.
There are different types of Appenders. A few examples are: `ConsoleAppender`, `FileAppender`, `RollingFileAppender`, `DBAppender` and `SMTPAppender`. We have already used `ConsoleAppender` to write to the console. `FileAppender` is used to write into a file. `RollingFileAppender` means that new files are created when the file has a certain size or, for example, a new log file is created at the beginning of each day. This is important for servers, because if we're always writing into the same file, the file size could reach tons of gigabytes. Therefore, `RollingFileAppender` is used on servers very often. `DBAppender` allows us to directly log into a database, and is very easily configured. `SMTPAppender` is used for logging and sending an email.

There are a lot of Appenders that you can use out of the box. As previously mentioned, you can always extend it and write your own Appender, which would allow you to log to any other source.

Chapter 15

The Java Main Method

In this chapter, I will be discussing the `public static void main(String[] args)` method. Up until this point we have run our code only through the JUnit framework. This is a sound, methodological practice, however, this is different from how our program would be run in production. Now let's explore how our code would run outside of the development environment.

The Main Method

Initially, the code you write in a computer program is just static text lying around passively in a file. To execute the code, the Java Runtime Environment (JRE) is responsible for loading the compiled program and starting its run. To execute the code the JRE needs an entry point. When running a Java class from the command line, the entry point the JRE looks for is the following method:

```java
public static void main(String[] args) {
    /*
     * JRE starts by executing any code in here
     */
}
```
Example 1

Let's look at each part of the method in detail:

- ***public*** - allows the method to be called from outside the class.

- ***static*** - allows the method to be called without having an instance of the class created.
- ***void*** - it returns no value.
- ***main()*** - To execute your program, Java will specifically look for a method of the name "main".
- ***String[] args*** - You can call your program with a number of arguments. Your program can access those arguments from this array.

The `String` array is called `args` by default. However, you should avoid abbreviations in variable names, as they will make your code difficult to read - therefore, I recommend that you use `arguments` as the name of the array, as you can see in Example 2:

```java
public static void main(String[] arguments) {
    /*
     * JRE starts by executing any code in here
     */
}
```

Example 2

Both the codes in Example 1 and Example 2 are recognized by the JRE. Also, one thing about the main method that you might find interesting is that you don't even need to use an array - you can replace the array by a parameter of variable length - "vargs" in short:

```java
public static void main(String... arguments) {
    /*
     * JRE starts by executing any code in here
     */
}
```

Example 3

The "vargs parameter" shown in Example 3 is like a more flexible version of an array - if you directly call this method, for example from a test, it has the advantage of accepting a variable number of `String` arguments, for example `main(“BMW”, “Porsche”, “Mercedes”)`, without having to create an array upfront. To be honest, I never really use a vargs parameter for the main method, but it I think it is a nice detail to know and show off ;-).

The static main method that we use as an access point is very specific. If you modify it beyond the ways I’ve discussed, it won’t work as you intend it to. If you want to drive your colleagues nuts ;-), you are free to deviate from that pattern, for example by making the method `int` instead of `void`, as shown in Example 4:

```java
public int main(String[] arguments) {
    return 42;
}
```

Example 4

This will create a method of the name `main`, but it won’t be recognized as “THE” main method, and therefore the program won’t be able to run using this method as a starting point.

Coding Examples

In Example 5, we will create a class called `CarSelector` and add a main method to it. It prints out each of the command line arguments back to the console:

```java
package com.cleancodeacademy.javaBook;

public class CarSelector {

    public static void main(String[] arguments) {

        for (String argument : arguments) {
            System.out.println("processing car: " +
                                                  argument);
        }
    }
}

```
Example 5

With the help of the main method we can execute this code without using its test to call it, as we have done up until this point in this book.

Compiling using the Command Line

To run our program from the command line, we must first navigate to the root folder of our source code, as I show in Example 6. In our case, this is src/main/java. As a side note, this is the default folder structure for "Maven", a build management tool I highlighted earlier when I talked about Java Development Tools. This is how I would do this on a unix terminal:

```
marcus-macbook-pro:~ marcus$ cd src
marcus-macbook-pro:src marcus$ cd main
marcus-macbook-pro:main marcus$ cd java
```
Example 6

As the full name of our class is `com.cleancodeacademy.javaBook.car.CarSelector`, the Java source file is stored in a subfolder "`com/cleancodeacademy/javaBook/car/`". To compile the code we type:

```
marcus-macbook-pro:java marcus$
   javac com/cleancodeacademy/javaBook/car/CarSelector.java
```
Example 7

This will create a file called `CarSelector.class` in the same folder as `CarSelector.java`, and we can finally execute our program:

```
marcus-macbook-pro:java marcus$
         java com/cleancodeacademy/javaBook/car/CarSelector
```
Example 8

Alternatively, we could also refer to the class by using it's fully classified Java name:

```
marcus-macbook-pro:java marcus$
         java com.cleancodeacademy.javaBook.car.CarSelector
```
Example 9

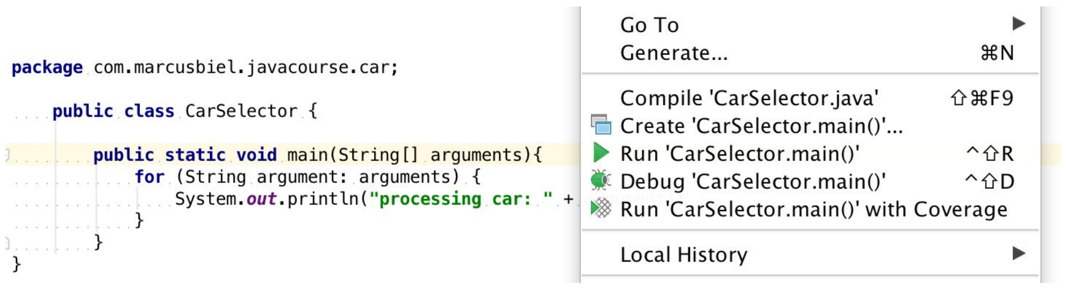
As you can see, calling our class without any arguments actually does nothing. So let's add some arguments:

```
marcus-macbook-pro:java marcus$
     java com.cleancodeacademy.javaBook.car.CarSelector BMW
                                        Porsche Mercedes
processing car: BMW
processing car: Porsche
processing car: Mercedes
```
Example 10

Hooray! We have successfully executed our own program from the console!

Running your Program using IntelliJ IDEA

To run our program from IntelliJ IDEA, we simply right click the method, and choose “Run ‘CarSelector.main’” from the context menu, as shown in Example 11.

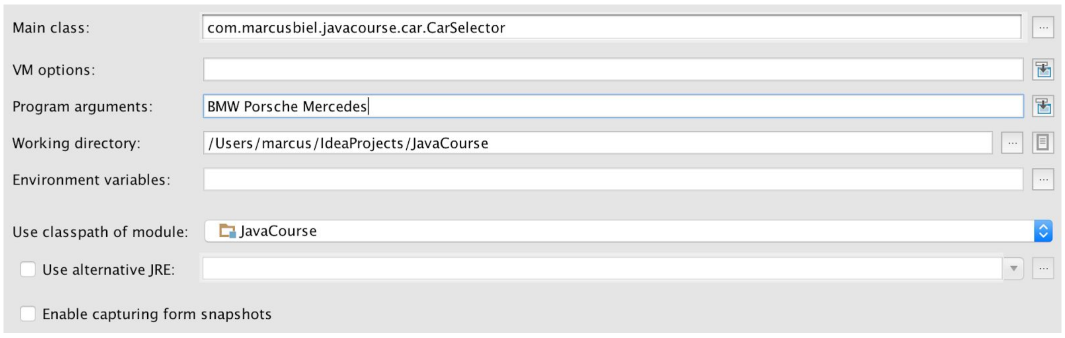


```
package com.marcusbiel.javacourse.car;

    public class CarSelector {

        public static void main(String[] arguments){
            for (String argument: arguments) {
                System.out.println("processing car: " +
            }
        }
}
```

Go To
Generate... ⌘N
Compile 'CarSelector.java' ⇧⌘F9
Create 'CarSelector.main()'...
Run 'CarSelector.main()' ^⇧R
Debug 'CarSelector.main()' ^⇧D
Run 'CarSelector.main()' with Coverage
Local History

Example 11

If we change the signature of the `main()` method, the “Run ‘CarSelector.main’” command will disappear from the context menu, as we no longer will have a valid entry point. However, when we run it, nothing is printed. This is because no one is passing the `main()` method any arguments. To do that in the IDE: from the “Run” menu, choose “edit configurations…”, and in the “configuration” tab add space separated strings to “Program Parameters”.



Main class:	com.marcusbiel.javacourse.car.CarSelector
VM options:	
Program arguments:	BMW Porsche Mercedes
Working directory:	/Users/marcus/IdeaProjects/JavaCourse
Environment variables:	
Use classpath of module:	JavaCourse
☐ Use alternative JRE:	
☐ Enable capturing form snapshots	

Example 12

Now when we run the `main()` method, we see our cars printed out:

```
processing car: BMW
processing car: Porsche
processing car: Mercedes
```

Example 13

Commentary

If you've read another Java book before this one, or even if this is your first one, you might be wondering why I have deferred the introduction of the `main()` method until this relatively advanced stage in the book. I did this for a few reasons. Firstly, I believe that it's important to give you the tools to completely understand something before I introduce it. If you didn't know what `public static void` meant, or you didn't know what an array was, it wouldn't have been fair to teach it to you. Now that you have some knowledge about all these things, you can begin to fully understand how this method works.

Another reason I chose to delay this discussion for so long is because in object oriented development, static variables and methods should be used sparsely. There are some instances where you would use static modifier, but I don't want to promote its use in this course.

Finally, you will rarely have to write a main method yourself. For every program (of any size) there is only one main method, and by the time you've joined a project, it probably was already written by someone else.

Part 3

Intermediate Concepts

Chapter 16

Exceptions

Handling exceptions is the process by which you handle an “exceptional condition”. These situations happen rarely, and in very specific instances. You could think of these as a specific type of bug, that you expect not to happen in normal programming, but you still want to protect against.

History of Exception Handling

The mechanism of exceptions does not exist in all languages. It was introduced in the Lisp programming language in 1972, which is about 30 years after the first programming language was invented. Initially there were only unchecked exceptions, but in 1995 the Java programming language introduced the concept of “Checked Exceptions”, which forces you to handle an exception before being allowed to run the program.

Java has also been **the first and last programming language to add checked exceptions**, which *may* imply that there are some disadvantages to this mechanism that I will discuss later in this chapter.

Unchecked Exceptions

Unchecked exceptions are exceptions subclassed from the class `RuntimeException` and are not enforced at compile time, but rather are simply thrown at runtime. An example of such an exception is the `NullPointerException` which is thrown when you access a member of a `null` reference. This type of exception will stop your code at runtime, so typically programmers use validation in their methods to prevent these exceptions from being thrown.

Exceptional cases are generally programming errors or (other) system failures. For example, if your code contains an enum that has states like `GREEN YELLOW RED` and a switch iterating over it, the current version of the code probably deals with the enum thinking it has only those three states. If at a later time the code was changed, and now the enum had more states, this would be a situation that you could not expect. You never know what will happen to your code in the future. If you had thrown a `RuntimeException` in the case of an unknown enum value, you'd be following a "fail early" approach. Throwing the exception allows you to quickly find and eliminate what is causing the exception. This is much better than 'hiding' the error until it becomes a bigger problem for the system.

Checked Exceptions

As briefly mentioned before, the Java designers devised checked exceptions as a mechanism to enforce the handling of exceptions. Checked exceptions are exceptions subclassed from class `Exception` which are checked at compile time. If you do not **handle** these exceptions in your code, it will not run. The only time you do not have to handle a method that throws an exception is if the method that you are writing declares that it too throws the exception, essentially passing the handling of the exception up the hierarchy.

In my opinion, the use of checked exceptions is contradictory to the idea of exceptions. As the name implies, **an *exception* is supposed to be *exceptional*. In other words, *unexpected***. Since you are *expected* to handle a Checked Exception, this implies that you know it can happen, so a Checked Exception **cannot**, by definition, be *exceptional*. If you know that under certain conditions something can happen, then you can plan for it and directly handle it. There will be no need for exception handling in this case. For example, it is common for users to send invalid input. You can validate the user's input and show an error to the user if the input was invalid.

Validating Input

So how do we validate our arguments before actually using them? I'll show you below how to write a method that does this. I'm going to call this method `isValid()`. It accepts one argument, and returns a `boolean` that tells us whether or not the argument is valid.

```
public class CarSelector {

    public static void main(String[] arguments) throws
                                                Exception {

        CarService carService = new CarService();
        for (String argument : arguments) {
            if (isValid(argument)) {
                carService.process(argument);
            } else {
                System.out.println("invalid argument:"
                                                + argument);
           }
        }
    }
}
```

Example 1

Now let's write the `isValid()` method. Since our `main()` method is `static`, this method needs to be `static`. We want the method to return `true` if the argument is valid and `false` if it isn't:

```java
private static boolean isValid(String argument) {
    try {
        CarState.valueOf(argument);
    } catch (IllegalStateException e) {
        return false;
    }
    return true;
}
```
Example 2

In the isValid method, we are going to have a `try`/`catch` block. First, we are going to `try` using the `valueOf()` method of the `enum` class. This method is part of the Java `enum`. It directly converts a `String` to the corresponding `enum` value. This method will throw an `IllegalStateException` in the case of an illegal value and if it does, we can `catch` that and return `false`. Otherwise our argument was valid. If you're interested in learning more about enums you can read the chapter about them.

The power of Exceptions

Exception handling is actually a very powerful mechanism. When an exception is thrown, the normal program flow is interrupted. But this power comes at a price: performance cost. However, this is not an issue for *exceptional cases* because they are only expected to happen occasionally and in the case of an exception, performance is not your biggest problem.

When you are creating exceptions never use Checked Exceptions. For unexpected exceptions, use a Runtime Exception and log the error. If you have something that is expected to be problematic, such as user input, instead of throwing an exception for invalid input, you should validate and directly handle invalid input.

Throwing an Exception

Let’s dive into a code example. In Example 3, we have an `enum CarState` with different state values, and a method `public static from()` that converts a `String` with a state name to an `enum` value (Since Java 7 it's been possible to use a `String` inside a switch-statement, but this is not generally recommended. However, in this case we'll use it, since the user input is in the form of a `String`).

We will add exception handling, in case the state name is not valid. We’ll do that by throwing an exception from the default case of the switch-statement. To throw an exception, you write the word `throw`, followed by creating a `new Exception()` object.

```
public enum CarState {
    DRIVING, WAITING, PARKING;

    public static CarState from(String state) {
        switch (state) {
            case "DRIVING":
                return DRIVING;
            case "WAITING":
                return WAITING;
            case "PARKING":
                return PARKING;
           default:
                throw new Exception();
        }
    }
}
```
Example 3

In this case, the exception we are throwing occurs whenever there is an invalid state being sent to the `from()` method.

The exception that we threw in Example 1 is a **Checked Exception**, so we have to handle it whenever we call the `from()` method. Typically, you don't want to handle an exception in the method where it is created, as this isn't the point of an exception. Our goal is to have this exception handled when another class uses this method, so that it is protected in case the exception is thrown. To pass along this exception we add `throws Exception` to the method declaration:

```java
public enum CarState {
    DRIVING, WAITING, PARKING;

    public static CarState from(String state) throws
                                                    Exception {
        [...]
    }
}
```

Example 4

Now, let's write a class that calls the `from()` method. We'll call this class `CarService`. Again, I'm just gonna pass this exception up the chain, so to speak.

```java
public class CarService {
    private final Logger log =
                  LoggerFactory.getLogger(CarService.class);

    public void process(String input) throws
                                                    Exception {
        log.debug("processing car:" + input);
        CarState carState = CarState.from(input);
    }
}
```

Example 5

Ok, now we get the next method in our code, but again, I'm just gonna write `throws Exception`.

```
public class CarSelector {

    public static void main(String[] arguments) throws
                                                    Exception {

        CarService carService = new CarService();
        for (String argument : arguments) {
            carService.process(argument);
        }
    }
}
```
Example 6

As you see, we ended up adding the exception in every calling method, all the way up to our `main()` method. The exception is not handled, and if it occurs, the program will stop with an error. This is a typical situation, in which you end up adding the `throws` statement to each and every function in your code. This is why a checked exception is frankly not very helpful; we never even bother to handle it and we end up getting the same stack trace printed as our output.

Instead of doing this, we could write an unchecked exception. If we write an unchecked exception, we can remove every single `throws Exception` that we have just written, and replace our `Exception()` throw with `throw new RuntimeException();` With this exception, we can also provide some information on why our exception was thrown. In this case, I’m going to tell the user that they sent us an unknown state, and then tell them which state it was that caused this exception to be thrown.

```java
public enum CarState {
    DRIVING, WAITING, PARKING;

    public static CarState from(String state) {
        switch (state) {
            case "DRIVING":
                return DRIVING;
            case "WAITING":
                return WAITING;
            case "PARKING":
                return PARKING;
            default:
                throw new RuntimeException("unknown
                                          state: " + state);
        }
    }
}
```

Example 7

Handling an Exception

Let’s say we don’t want our code to stop running when the user sends us an unknown state. We could handle the exception by surrounding our `from()` method call with a `try/catch` block. Usually, you want to have this try/catch block **as early on as possible** in the code, so wherever the problem will potentially be caused, you should surround that call in the try/catch. In this case, that is in our `main()` method.

First, we wrap our method in a try block. After the try block ends, we add a catch block that accepts the Exception as its argument. All exceptions have a method called `printStackTrace()` that prints a list of all the methods called in reverse order. This is great for debugging because we can see exactly where in each class the exception happened.

In Example 8's try block, if an exception occurs, none of the lines after the line that caused the exception will be executed. If there isn't an exception, then you can safely assume that the call succeeded in the following lines. Because of this, we can write our try block of the code as if everything works, while handling the exceptions separately.

```java
public class CarSelector {

    public static void main(String[] arguments) {

        CarService carService = new CarService();
        for (String argument : arguments) {
            try {
                carService.process(argument);
            } catch (RuntimeException e) {
                e.printStackTrace();
            }
        }
    }
}
```
Example 8

The finally-Block

Exceptions, as I've noted, interrupt the normal program flow, but often we have code that we want to run whether or not our try block works. For example, when handling resources such as IO or database connections, we want to properly free them in the case of an exception.

In fact, it is a classic bug for programs to leak resources or improperly close them when exceptions occur, because the code that handles freeing them is interrupted, or isn't executed.

For this reason Java defines the `finally` keyword, to add an optional `finally` block to the `try/catch` construct.

```
try {
    carService.process(argument);
} catch (RuntimeException e) {
    LOG.error(e.getMessage(), e);
} finally {
    System.out.println("I print no matter what");
}
```
Example 9

With a valid argument, the code in the `try` block will be executed. With an invalid argument the normal program flow will be interrupted, and the code in the `try` block will not be executed, but we will instead log our error. No matter what, the line *"I print no matter what*" will still print.

Ways to Handle Exceptions

Many Java books and tutorials never teach how to properly handle an exception and append a comment along the lines of "do proper exception handling here", without explaining what to "properly" write. This might be why many programmers often print a stack trace instead of choosing a more beneficial way to handle their errors.

`e.printStackTrace` prints the exception's stack trace to the standard error output, which usually is the raw console. In simple terms, the stack trace shows the order of nested method calls that lead to the exception, which can be helpful to find the cause of the exception. In other words, this is a basic attempt to generically signal that an error occured, along with the potential source of the error.

This can be useful in very specific situations, like for small developer tools run from the console. Usually however, this approach is not optimal. If possible, you should always handle an exception. For example, if reading a value from a database fails, it might be possible to return a cached value instead. If you can't handle the exception, the minimum you can do is to log the exception as well as keep hold of the complete state the system was in when the exception happened - as close as possible.

Generally, you want to have information such as the time when the error occurred, the name of the system, the full name of class and the method, and the exact line number where the error occurred in the code. Additionally, you should log the status of relevant objects, such as the parameters used by the faulty method. With the help of a logging framework like Logback, this can be achieved without much effort. It can be configured to log to a file or a database, which will permanently persist the error. To learn more about logging, read the chapter on Logging with SLF4J and LOGBack.

Using a logging framework for error notification is generally preferable to directly printing out the error to the console, as it is a more flexible and powerful way of getting hold of the cause of an exception and to persist it to a storage medium such as a file or a database.

Chapter 17

Interfaces

Definition of the Interface

“Interface” is a generic term that is used widely across fields. Wiktionary defines it as “the point of interconnection between entities”. This term has been adapted to programming and plays a key role in Object Oriented Programming (OOP). In OOP, an interface is defined as “a piece of code, defining a set of operations that other code must implement”.

History of Interfaces

Even though many believe that Java was where the idea of an interface was initially introduced to programming, it actually was introduced by Brad Cox and Tom Love, the creators of Objective-C, with the concept of the protocol.

While the concept did not originate in Java, it was adopted into the language from the very beginning. You could say that Java was the successor to C++ and in C++ they used a model involving multiple inheritance. Multiple inheritance is more complicated and problematic than the single inheritance model that is used in Java. It also is more difficult to implement multiple inheritance in a compiler. Funnily enough, it seems as though interfaces weren't introduced into Java to create “cleaner, more modular, and clearly separated code”, but rather just to compensate for the fact that Java doesn't support multiple inheritance. Nowadays however, that’s exactly what interfaces are useful for.

Writing an Interface

To demonstrate interfaces, we're going to create a `CarService` class. We know that we want our `CarService` class to have a method called `drive`. When `drive` is invoked in the `CarService` class, we're going to cause all the cars in our service to drive.

```java
package com.marcusbiel.javaBook;

public class CarService {

    public void drive() {
        BMW bmw = new BMW();
        Porsche porsche = new Porsche();
        Mercedes mercedes = new Mercedes();
        bmw.drive();
        porsche.drive();
        mercedes.drive();
    }+
}

```
Example 1

As you can see in Example 1, our `CarService` creates three more objects. At this moment, the `CarService` class is deeply coupled to these other classes of cars. Each car is its own class, with no real connection besides the fact that they each have a `drive` method. This is something we don't want and I'll explain why later in more detail.

Here is where we can improve our code using an interface. In our code we have three different cars. All of these cars can drive, and you can already see that we have three different drive methods. We could just create a `Car` class and a `carType` value in that class, but the `drive` method in this hypothetical class will force all our different types to drive the same way. This is an issue, because different cars drive in different ways. For example, the specific car types, (BMW, Mercedes, and Porsche), all have their own unique engines. On the other hand, we've already recognized that all these cars drive and that they also have other similarities like four wheels and two axles. These common features can be grouped together and accessed using an interface, without any knowledge of the particular car type. Example 2 below illustrates this nicely.

```java
package com.marcusbiel.javaBook;

public interface Car {
    void drive();
}

```

Example 2

Now we have defined an interface called Car which contains the declaration for the drive method. Please note that by default all methods in an interface are public abstract, so we don't need to include those modifiers in our methods. Doing so would just clutter the code. An abstract method requires any class that implements this interface to provide concrete implementation of the abstract method. Similarly, all class level variables declared in an interface have the default modifiers "public static final". Typically, while you can, you don't want to include constants in an interface. If you create a constant called MAX_SPEED at an interface level, you are adding concrete values to an interface. The goal of interfaces is to be "lightweight", without any implementation. Like a contract or a blueprint, interfaces define "what", but not "how". These implementation details should be put within a class, or even better, in an enum.

Subclass Implementing Interface

Let's modify our BMW class so that it "implements" the Car interface. This defines the BMW as a type of Car and it will adhere to the contract, or role, specified by `Car`. For a concrete class to successfully implement an interface, it needs to override all the abstract methods of that interface. In this case, `BMW` must override the `drive` method. We're also going to implement the `Loggable, Asset,` and `Property` interfaces. To implement multiple interfaces, separate each interface name with a comma.

```java
package com.marcusbiel.javaBook;

public class BMW implements Car, Loggable, Asset,
                                                    Property {

    @Override
    public void drive() {
        System.out.println("BMW driving...");
    }

    @Override
    public int value() {
        return 80000;
    }

    @Override
    public String owner() {
        return "Marcus";
    }

    @Override
    public String message() {
        return "I am the car of Marcus";
    }
}

```

Example 3

Overriding a method is quite straightforward. In an implementing class, declare a method with the same signature and return type as the method you want to override The signature of a method consists of its name and return type. No two methods in the same class can have the same signature.

Finally, as you can also see in Example 3 above, it is good practice to add the `@Override` annotation wherever you override a method. This clearly signals that the method is meant to override another method.

Besides overriding interface methods, you can also override a method of a super class. We will talk about this more in detail in the chapter on Inheritance in Java.

For each of the interfaces I implemented, I had to override a method as I did above. Now we can revisit our `CarService` class. Assuming we implemented `Car` for the `BMW`, the `Mercedes`, and the `Porsche`, we can clean up our code. The first thing we can do is instantiate all of our car types as `Car`:

```
Car bmw = new BMW();
Car porsche = new Porsche();
Car mercedes = new Mercedes();
```
Example 4

The objects keep their specific types, however, we are referencing them with an interface to the object. By doing this our reference variable will only allow us to use methods provided by the given interface, and now the object plays a certain "role" in the given context.

As shown in Example 3, BMW also implements the other interfaces `Loggable`, `Asset` and `Property` that would only allow us to use different sets of methods. These "lenses" that our reference variable could act as allows a `BMW` object to fulfill different roles in different contexts.

Now we can use `Car` for all three of the cars and it would make our code much more flexible.
Even if later on we add new types of cars that implement the `Car` interface, we can still deal with them even if we didn't know of their existence when we wrote `CarService`. Also, because all of our cars are actually implementing the `Car` interface, we can clean our code up even more using a foreach loop. We can now also retrieve all of our cars from a database, because no matter the type of car, they all can be stored in one array.

```
package com.marcusbiel.javaBook;

public class CarService {
    public void drive() {

                [...] /* dynamically retrieving cars from a
                                              database */

        for (Car car : cars) {
            car.drive();
        }
    }
}

```
Example 5

Now our `CarService` doesn't have any specific implementations anymore. This is called “decoupling”. The `CarService` class only uses and knows about the `Car` interface, not any specific types of cars.

Pros and Cons of Interface

To conclude, I’m going to discuss the pros and cons of interfaces in Java.

Interfaces play an important role in decoupling Java code. Declaring a reference variable of an interface type allows you to substitute for different car types ***at runtime***. For example, our `CarService` class can deal with either a `BMW` or a `Mercedes` based on different scenarios.

This advantage is especially important in large projects with many developers. For a team to work efficiently, good communication is essential. However, with every additional team member, the communication effort increases exponentially. This means that teams larger than a certain size can no longer work efficiently. To counteract this, large components may be divided into subcomponents. Interfaces enable communication between the components. They constitute a binding contract for both sides and offer guidelines that both sides can follow without having to know the exact implementation details of the other side.

On the other hand, the drawback of using an interface is that it adds additional complexity to a system. Each interface provides code that needs to be written, read, understood and maintained in the long term. Also, an interface acts as layer of "indirection" between the code that performs a function and the code that relies upon that function. This can make it more difficult to navigate the system and, for those unfamiliar with it, to understand the system. Even if a single interface on its own does not add much additional complexity, the effort can quickly add up in a system with many interfaces. As a Clean Coder, it is essential to fight any kind of artificial complexity rigorously if you want to maintain a truly clean system.

You should never introduce an interface simply for the sake of it. Please think very carefully about where an interface could be useful and where not. Base your decision only on concrete requirements, and not on possible changes that might arise later on. Software is soft. We can introduce an interface, should this prove useful, also at a later time.

That concludes my chapter on interfaces in Java. Remember, interfaces are one of the most powerful tools Java has to offer, but they should be used wisely.

Chapter 18

Inheritance and Polymorphism

In this chapter, I will be discussing inheritance in Java. Similar to interfaces, inheritance allows a programmer to handle a group of similar objects in a uniform way which minimizes code duplication. However, if inheritance is not utilized properly and at the right time, it can make the code unmaintainable or even cause bugs in the long run.

A Word of Warning

Inheritance is one of the most powerful features of object-oriented languages. It sounds very promising and useful when first described. However, I think it is just a bit too powerful. If used incorrectly, it can damage your code base—leaving it very vulnerable to bad design, so take care to use it judiciously. There definitely are cases where the use of inheritance is justified, but only a few. It requires some experience on the part of the programmer to wield inheritance properly.

What is Polymorphism?

Polymorphism is a concept deeply related to interfaces and inheritance. The term “polymorphism” originates from the Greek roots *poly morphs* – *many forms*. Polymorphism allows us create different objects on the right side of a variable declaration, but assign them all to a uniform object type on the left side. Let’s take a look at the coding example below:

```java
package com.cleancodeacademy.javaBook;

import org.junit.Test;

public class ZooTest {

    @Test
    public void shouldFeedAllAnimals() {
        Zoo zoo = new Zoo();
        Animal[] animals = { new Dog(), new Gorilla(),
                                        new Tiger(), };
        zoo.feed(animals);
    }
}

```
Example 1

We are not actually testing something here; this is only a demonstration. This example is actually taken from the book [*Head First Java* by Kathy Sierra]. She's my favorite author for any Java-related book. I highly recommend that you read *Head First Java* and any of her other books as they're extremely creative and fun to read on top of being very informative.

Let's focus on our array of `Animal` objects. Our array has a `Dog`, a `Gorilla` and a `Tiger`. The usage here is indicative of polymorphism. We're using a reference variable of our supertype, or parent class, `Animal`, on the left side of our statement and on the right side the instances are any subclass or subtype of `Animal`: `Dog`, `Gorilla`, and `Tiger` in our example. These different animals can be assigned to indices of `animals` because polymorphism means that we can have different implementations on the right side that are handled uniformly on the left.

If you look in our `Zoo` class, you can see the `feed` method below that applies the same code to any `Animal`, no matter whether it's a `Dog` or a `Tiger`.

```java
package com.cleancodeacademy.javaBook;

public class ZooTest {

    public void feed(Animal[] animals) {
        for (Animal animal : animals) {
            animal.eat();
            animal.grow();
        }
    }
}

```

Example 2

We are iterating through the array of animals with an arbitrary example where each `animal` eats and grows. All of these animals probably eat in a different way, but as we can see, they are told to eat and grow no matter what. For example, a gorilla would eat leaves and insects while a tiger would eat meat. The idea is for each `Animal` subtype to have its own concrete implementation; its own way of eating as specified by its `eat` method.

To reinforce the idea, we also have the `grow` method. My idea is for `grow` to be a concrete method, while `eat` is an abstract method – with both methods residing in the `Animal` class. This would mean that all animals grow at the same rate while eating differently.

Inheritance

Let’s jump to our first iteration of the `Animal` class:

```java
package com.cleancodeacademy.javaBook;

public class Animal {

    public void eat() {
        System.out.println("Animal is eating");
    }

    public void grow() {
        System.out.println("Animal is growing");
    }
}
```

Example 3

As you can see, our `Animal` class has two concrete methods: `eat` and `grow`. Later on we will make `grow` abstract. We have both methods print out to the console, which is something we should generally avoid. However, for our example where we're demonstrating something, this would be an exception to the rule, but avoid doing it in real code. As you can see, this class looks very much like any regular class you have seen so many times before.

Now, let's take a look at the `Dog` class. Say we want to declare `Dog` as an `Animal`. If `Animal` were an interface, we would use the `implements` keyword. Since it's a `class`, we express its subclass relationship with the `extends` keyword instead. Quite simply, we say "`Dog extends Animal`" and *voilà*, we've just used inheritance.

```java
package com.cleancodeacademy.javaBook;

public class Dog extends Animal {

}

```

Example 4

Now it might look like the `Dog` class is empty but in fact we can already use `eat` and `grow` with `Dog`. We can also create the `Gorilla` and `Tiger` classes in the same way, but I won't show them here. Instead, let's move on and mix implementing interfaces and extending classes by implementing two interfaces: `Loggable` and `Printable`. The question now becomes, which comes first: `extends` or `implements`? There is a syntax rule stating that `extends` should come first followed by `implements` because of Java's single inheritance model. We can only extend a single class but we can implement any number of interfaces. We reflect this in a change to the declaration of the `Dog` class:

```java
package com.cleancodeacademy.javaBook;

public class Dog extends Animal implements Loggable,
                                           Printable {

}

```

Example 5

Note the comma-separated list of interfaces, namely `Loggable` and `Printable`, which `Dog` declares as its interfaces. These interfaces tell us that our `Dog` class can play the role of a `Loggable`. For example, we can print and log our `Dog` object since it can play the role of a `Loggable`. You can learn more about this in the chapter about Interfaces in Java.

If we wanted all animals to implement `Loggable` and `Printable`, we could implement these interfaces in the `Animal` class (and remove them in `Dog` to avoid duplicate code).

Multiple Inheritance

Although we can implement several interfaces, in Java we can only extend up to one class, so we can't say something like "`Pig extends Herbivore,Carnivore`". Note that this works in C++ and can be even nastier than single inheritance. The use of multiple inheritance can lead to something called the Deadly Diamond of Death.

Abstract Classes

An abstract class is like a hybrid between an interface and a class. Simply put, we are allowed to create both abstract and non-abstract methods in an abstract class. Recall that for interfaces, we don't have to explicitly declare our methods abstract since all methods are abstract by default. Because an abstract class can mix both abstract and non-abstract methods, however, we have to explicitly use the abstract modifier for abstract methods. Please note that, just like with interfaces, you cannot create an instance of an abstract class.

An abstract class can extend another abstract class, which means it can add further methods (both abstract and non-abstract), as well as implement or override (abstract) methods of its superclass. The first concrete class in a hierarchy of classes extending one another must ensure that all abstract methods of the hierarchy are implemented.

Let's make `eat` abstract in `Animal`. Since we know that each of our subclasses will implement this method differently, there's no reason to provide a concrete implementation of this method in the `Animal` class.

```
public abstract void eat();
```
Example 6

However, declaring `Animal` abstract changes its meaning and behavior. Since abstract classes cannot be instantiated, the compiler will no longer allow us to instantiate an `Animal` object with `new Animal`. This is what our `Animal` class looks like now:

```java
package com.cleancodeacademy.javaBook;

public abstract class Animal {

    public abstract void eat();

    public void grow() {
        System.out.println("Animal is growing");
    }
}

```

Example 7

When Should I Use Abstract?

Firstly, I'd like to note that there is no "golden rule of abstract". Everything with inheritance is extremely case by case, and even modifying your program could change whether or not abstract should be used.

You should make a parent class abstract when you don't intend for it to be created. What is an animal? In this case you don't intend to create an `Animal` object; you're only going to be creating tangible animals like tigers, dogs and gorillas. That's when you should make a class abstract. When you only intend to use its children in your program.

You should make your methods abstract when you intend every child to have a different implementation. If you know that there's no "default" eating behavior for an `Animal`, there's no reason to write one that is going to be overridden by every single class. If some or all of your children will implement a method in the same way, you may want to write that method and override it when necessary. This after all is why inheritance is useful in the first place.

Implementing an Abstract Method

Going back to our classes, we still have a few more bumps to iron out. Now that the `Animal` class' `eat` method is abstract, we are forced to implement the abstract `eat` method for the other subclasses.

Implementing a method of an abstract class is done the same way as implementing a method of an interface: by overriding. I briefly explained overriding in the chapter on interfaces in Java. Here's a quick recap, with some extra detail:

Overriding a method is quite straightforward. In a subclass, declare a non-abstract method with the same signature and return type as the method you want to override The signature of a method consists of its name and return type. No two methods in the same class can have the same signature.

Additionally, it is good practice to add the `@Override` annotation wherever you override a method. This clearly signals that the method is meant to override another method, and helps the compiler alert us to any mistakes we might make when trying to override a method.

For example, imagine that, when attempting to override a method, you declare a method with the same name and return type as the original, but a different parameter list. The compiler will consider this a completely new method, and, without an `@Override` annotation, the code will compile normally but not work as expected at runtime. By including the annotation, the compiler knows to check that the method you are trying to override actually exists, and it will throw an error if that check fails, before the code is ever run.

Note:
Declaring multiple methods with the same name, but a different parameter list is a common practice in Java programming known as

overloading. Overloading a method can be quite useful in certain cases, such as when some parameters of the original method are optional. As per good naming conventions, a group of overloaded methods should all perform the same or a very similar function.

As a practical example, let's override the `Animal` `eat` method in the `Tiger` subclass:

```java
package com.cleancodeacademy.javaBook;

public class Tiger extends Animal {
    @Override
    public void eat() {
        System.out.println("Tiger is eating...");
    }
}

```

Example 8

Here, we have an example of a subclass' implementation of `eat`. Each `Animal` subclass would have its own specific logic for that method. A real world implementation would probably not use a print line statement for implementing `eat`, however, it's good enough for our demonstration.

All of the logic that is common for all `Animal` subclasses, like `grow`, belongs in the `Animal` class. It's a double-edged sword, though. On the one hand, it is very handy for code reuse, but on the other hand it makes our program more complicated to understand. For example, someone reading the code can't tell for sure whether a `Gorilla` ages and eats with the `grow` and `eat` methods of the `Animal` class or if it has its own overriding `grow` and `eat` methods like the ones below.

```java
public class Gorilla extends Animal {

    @Override
    public void grow() {
        System.out.println("Gorilla is growing...");
    }

    @Override
    public void eat() {
        System.out.println("Gorilla is eating...");
    }
}

```
Example 9

This is similar to how you can just define any arbitrary exception to some rule. You could say, “Well, my `Gorilla` doesn’t use the `Animal grow` method but instead replaces it with its own.” But you want to make sure you use this tool effectively. If your intention is to modify the functionality of a method that was written in `Animal`, you can override it.

If you are intending to have a new method of the same name, but with different parameters, you are not overriding, but “overloading” the method. For example, we could have our `grow` method that increases an animal’s size based on a default value and a “`grow(int amount)`” method that grows by `amount`. Two methods with the same name can exist as long as they have different parameters.

Changing Visibility

There is yet another dimension to inheritance, and for this we have to introduce a new visibility modifier. The `protected` modifier gives visibility of a member to any class in the same package, as well as to any subclasses. Protected visibility is different from the package-private visibility modifier in that the protected visibility modifier allows for a method to be accessed outside the package through inheritance. In this case, let's apply it to `grow`.

```java
package com.cleancodeacademy.javaBook;

public abstract class Animal {
    public abstract void eat();

    protected void grow() {
        System.out.println("Animal is growing");
    }
}

```
Example 10

In our example, the `protected` modifier implies that the `grow` method of `Animal` is visible to anything in the `com.cleancodeacademy.javaBook` package, as well as to any `Animal` subclass outside of that package. Both properties of `protected` make using it complex, so much so that we will be going into even more depth to understand why we should avoid using it. Among the visibility modifiers in Java, only `public` has more access privileges than `protected`.

When you override a method in a Java class, you **are** allowed to change its visibility. However, you can only increase the visibility of members in the child classes. This means that in this case, the only visibility modifiers we can use for `grow` in `Animal` subclasses are `protected` and `public`. In any `Animal` subclass, trying to make `age private` will result in a compiler error which says, "Attempting to assign weaker access privileges." In the end, our abstract class `Animal` defines a contract or a protocol, exactly like with interfaces. You are in fact promising any client class in the Java universe that the client can in fact use any `Animal` instances with all the `public` methods they provide. Just imagine how messy it would be if it were optional for a class and its subclasses to fulfill the visibility contract. You would have a practically broken inheritance model!

When we inherit the members of a class, we inherit both instance methods and instance variables. This means that if `Animal` had a protected `weight` variable, all its subclasses would inherit it. Say for instance that when our `Tiger` grows, it adds a kilogram to its weight. It would look something like our example below:

```java
public class Tiger extends Animal {
    @Override
    public void grow() {
        super.grow();
        weight += 1.0;
    }
}

```
Example 11

One of the issues with the `protected` visibility modifier is that it has incredibly complicated rules that make it very easy to mess up while programming. It also violates **encapsulation** because `Animal` should be the only one with access to its internal variables. Even though it's possible, I would never make use of the `protected` visibility modifier.

Now that we've completed our classes, we are finally in a position to run our `ZooTest`. When we run it, each `Animal` subclass implements its own `eat` method and if it lacks its own `grow` behavior it uses the one specified in `Animal`.

Weaknesses in Inheritance

Inheritance can easily become extremely confusing and difficult to use. For example, say you have five classes inheriting from one another in a hierarchy. To understand the behavior of an object originating from this class hierarchy, you would have to understand the entire network of classes as well as its interwoven dependencies. Every single method could be overwritten in any or all of the classes in the hierarchy. Even worse, a method of a child class can call all non-private methods of its parent class by putting a preceding “super” before the method name of the parent class. This allows the behavior of a (supposedly simple) method to turn into a very tight sticky mush, and to turn basic inheritance into an intransparent network of classes, highly dependent on one another. But what if your client only needs you to create a few classes? Well, even that could end up not working out.

Let me illustrate this briefly: Imagine you have an abstract `Duck` class with several variants to implement, like `MallardDuck` and `RedHeadDuck`. Both are Ducks, both `fly` and `swim` in the same way, so this could be a justified reason to use inheritance. We provide a default implementation of `fly` and `swim` in the `Duck` class for both `MallardDuck` and `RedHeadDuck`, and an abstract method `quack` that both implement in their own unique way.

Months later, we extend our system adding a `RubberDuck` and a `DecoyDuck`. The abstract method `quack` has to be implemented for both, however, `RubberDuck` will internally `squeak` instead. `DecoyDuck` neither quacks nor squeaks, so we provide an empty implementation `quack` in `DecoyDuck`. We also don’t want our `RubberDuck` and `DecoyDuck` to fly. All of these changes are implementable, but our design that once shone with beauty and flexibility is starting to be more of a pain than a help.

There is a better solution. We could use a more flexible approach, known as the *Strategy Pattern*! To start, you replace abstract class `Duck` with an `interface Duck`. Since it’s an `interface` it can only contain abstract methods for `fly`, `swim` and `quack`. Second, you define concrete classes for `MallardDuck`, `RedHeadDuck`, `RubberDuck` and `DecoyDuck`, each implementing the interface `Duck`.

Now we define three more interfaces: `FlyingBehavior`, `SwimmingBehavior`, and `QuackingBehavior` – and provide various implementations for each. For example, `FlyingWithWings`, `Quacking` and `Squeaking`, and so on. As `RubberDuck` and `DecoyDuck` both don't `fly` and `DecoyDuck` doesn't even `quack` we also have to provide a `NotFlying` and `NotQuacking` implementation. Still not very nice, but much nicer than using inheritance. Now, all ducks won't have a default implementation.

Using the strategy pattern, it will never happen that a `Duck` is suddenly flying on production, when it should not. Things are more clear. On top of this, this design is way more flexible – you can exchange the behavior at runtime.

```java
if (summer) {
    flyBehavior = new FlyingWithFastWings();
} else {
    flyBehavior = new FlyingWithWings();
}
```
Example 12

In conclusion, there are sometimes justified cases of inheritance, but later the requirements change, and using inheritance can quickly become a maintenance nightmare. On the other hand, even in cases where using inheritance is advised, it usually doesn't hurt to still decide against its usage. I wanted to illustrate for you how problematic inheritance is, although it may not seem so at first. I also wanted to show that there are great alternatives to using it. The above example of a strategy pattern was taken from *Head First Design Patterns*, a book I highly recommend that you read.

As a rule of thumb, don't use inheritance. For the last five years of Java development, I haven't used inheritance a single time. Instead you should utilize interfaces whenever possible, as I have been doing in my work.

Chapter 19

The Object Finalize Method

In this chapter, I will be discussing the `Object` `finalize` method in Java. The class `Object`, which is the superclass of all classes, defines the `finalize` method as well as other methods including clone, toString, hashCode and equals.

What is the Purpose of the Finalize Method?

`finalize` is a “hook method” of the class `Object`. In programming, a “hook” is a mechanism to “hook” into an existing process and extend its functionality. A “hook method” is one way to realize a hook in Java by using inheritance. The hook method is an empty method of a base class. Typically, the hook method is already called by users of the base class, with no effect, since the base method is empty. However, by overriding this method in a child class, the overridden implementation can be executed automatically by callers of the hook method.

According to the Java 8 documentation, the `finalize` method is called by the **Garbage Collector** when there are no more references to the object from active parts of an application. The usual purpose of this method is to perform cleanup actions just before the object is irrevocably discarded.

The Garbage Collector

The garbage collector, as its name suggests, collects the “garbage” of your program. The garbage that’s being cleaned up are the objects that you created, processed on, and then later put aside when you no longer needed them. The garbage collector's main responsibility is to free up memory resources so that your program hopefully never runs out of memory. However, the garbage collector runs asynchronously, and we have no control or influence over if and when it will be running. We can give it recommendations, but they are just that - recommendations.The garbage collector is not bound by them.

A computer uses several types of hardware resources, and these are physically limited by nature. Therefore, we have to use these resources wisely. In Java, unlike in older programming languages like C++, memory is automatically managed by the garbage collector. However, there are other resources, usually "IO resources", that we need to manage by ourselves.

The "I" of "IO" stands for input, and the "O" stands for output. Therefore, IO is a general term for input or output functionality (also referred to as communication) from a resource such as a file, an external system, or a database. Input is **read into** our system, while output is **written from** our system. A typical name for a class or interface that is used to read input is "Reader", while a typical name for a class or reader that is used to write output is "Writer".

These resources must be requested when needed, and released when not needed anymore. As this is rather tedious, the idea was to automatically release them just before the objects that are using them are discarded, and hence, the `finalize` method was born. However, as there is no guarantee that the `finalize` method will ever be called, there is no guarantee that those resources will ever be released either, and therefore the `finalize` method is useless. If you're interested in learning more about this you could read Effective Java by Joshua Bloch, where he's done a lot of interesting research on the subject. For example, he's tested and found that creating an object and destroying it using an overridden `finalize` method is 430 times slower than using the inherited method.

Overriding Finalize

In order to have the `finalize` method run, let’s implement a `Porsche` class that overrides it.

I introduced overriding and overloading, and the fundamental differences between the two in a previous chapter, Interfaces in Java. If you are unfamiliar with either concept, I suggest you check out that chapter before continuing.

```
public class Porsche {

    private IOReader ioReader = new IOReader();

    @Override
    protected void finalize() {
        ioReader.close();
    }
}
```

Example 1

Implementing the `finalize` method is easy. We simply release all resources that the object about to be garbage collected currently holds.

For example, let’s say that we have an `IOReader` that reads a character stream from a file. Once we are finished, we would like to close this reader so that the file can be used by another process. To do this, the `IOReader` class already provides a `close` method, so our `finalize` method can simply call that.
However, there are problems with the `finalize` method. First, as already mentioned, we do not have any guarantee that this method will ever be called. It is totally under the JVM’s control, and outside of our influence. The second problem is that if within this code we have any exception that is not handled, the process will stop and the objects will remain in a weird “zombie” state, which slows down garbage collection.

The Alternative to Finalize

As you can tell, the problem with the `finalize` method in Example 1 has nothing to do with our specific implementation, but rather with the way the `finalize` method actually works, and there is actually no way for us to fix that. In fact, since the release of Java 9, `finalize` has been officially deprecated, so no new code should ever use it.

The recommended alternative is to create our own `close` method which cleans up/closes all the resources no longer in use, in this case `IOReader`. This way we have more control over our resources, and aren't depending on the garbage collector.

```java
public class Porsche {

    IOReader ioReader = new IOReader();

    public void close() {
        ioReader.close();
    }

}
```

Example 2

Let's also write a draft of closing an object in the `CarSelector` class. First, let's create the `CarSelector` class and add a Porsche object with the following line: `Porsche porsche = new Porsche;` and use our new `close` method. We'll surround this in a try, catch, finally block and use our finally block to clean up our Porsche. The critical point here is that you have a finally section at the end, because the cool thing about finally is that, even if an exception occurs, it is guaranteed to always be executed, which is exactly what we need to solve our problem and make sure all non-memory resources are always freed. Here is how you can do it properly in Java:

```java
package com.marcusbiel.javaBook;

public class CarSelector {

    public static void main(String[] arguments) {
        Porsche porsche = new Porsche();

        try {
            /* some code */
        } finally {
            porsche.close();
        }
    }
}

```

Example 3

This way we guarantee that once we are done with our `porsche` object, we will close all of the affiliated resources involved without overwriting `finalize`. The finally block does exactly what we wanted our `finalize` method to do.

The `finalize` method is extremely flawed. The garbage collector has a mind of its own and we can't truly control when and how it operates. Since overriding the `finalize` method doesn't effectively solve our problem, you can instead use a try-(catch)-finally block to close any extra resources when you're done with an object. Since the close method is our own method, unaffiliated with the garbage collector, it works exactly as intended, every time.

Chapter 20

The Object Clone Method

In this chapter, I will be discussing the Java `Object` `clone` method. The `clone` method is defined in class `Object` which is the superclass of all classes. The method can be used on any object, but generally, *it is not advisable to use this method.*

The Clone Method

The `clone` method creates a copy of an object. Depending on the type of implementation, this copy can either be a shallow or a deep copy. In any case, it creates a new, distinct object that has the same value as the original. (For more details see the chapter about Identity and Equality in Java).

Using Clone on an Object

Let's take a look at how the `clone` method works. In Example 1, we create an object of class `Porsche` named `marcusPorsche`, giving me a very nice new car. However, I'm a nice guy and I'd like to give away another Porsche, so I'll clone the `marcusPorsche` object and assign the new object to the reference variable `peterPorsche`. If your name is Peter, congratulations! You just got a new Porsche!

```java
package com.marcusbiel.javaBook;

public class PorscheTest {

    @Test
    public void shouldClonePorsche() {
        Porsche marcusPorsche = new Porsche();
        Porsche peterPorsche = porsche.clone();
        assertNotSame(marcusPorsche, peterPorsche);
    }
}

```
Example 1

However, something is still wrong. The `clone` method is red. The problem is that the `clone` method from class is `Object` protected. Every object can call `protected` methods inherited from the `Object` class *on itself*. However, it can never call such a method on *other objects.* While `clone` is accessible to *both* our `Porsche` class and our `PorscheTest` class, the `PorscheTest` can never call the inherited `clone` method of a `Porsche` object.

To fix this problem, we have to override the `clone` method in the `Porsche` class and increase its visibility. First, we change the access modifier of the `clone` method to `public` and change the method return type from `Object` to `Porsche`, which makes our code clearer since we don't have to cast cloned objects into Porsches. To implement the clone method, we call "`super.clone()`". The `super` keyword means that we are calling a method from a superclass, which in this case is the `Object` class.

Note also that the clone method of the `Object` class is declaring a checked `CloneNotSupportedException`, so we have to decide whether we want to declare it or handle it. Declaring a `CloneNotSupportedException` while implementing (supporting) the `clone` method is contradictory. Therefore, you should omit it. All possible error situations are serious errors, so the best you can possibly do is to throw an `Error` instead.

```java
package com.marcusbiel.javaBook;

public class Porsche implements Car {

    @Override
    public Porsche clone() {
        try {
            return (Porsche) super.clone();
        } catch (CloneNotSupportedException e) {
            throw new AssertionError(); /* can never
                                             happen */
        }
    }
}

```
Example 2

However, when we run the above code, we encounter another issue: a `CloneNotSupportedException`. To correct that, we have to implement the `Cloneable` interface. The `Cloneable` interface does not contain any methods, it is a marker interface – an empty interface used to mark some property of the class that implements it.

```java
public class Porsche implements Car, Cloneable {
```
Example 3

Now when we run the test in Example 1, it passes successfully.

Modifying a Cloned Object

At this point, we're going to add a test to check the content of the new object. We want to verify that our owner has been correctly identified in each object. The `asString` method will be used to return a `String` representation of our `Porsche` object. The expected result when we are finished is for the object to be `"Porsche of Peter"`. First, I'm going to create the `asString` method in our `Porsche` class and an owner attribute.

```java
String ownerName;

[...]

public String asString() {
    return "Porsche of" + ownerName;
}
```
Example 4

In the code below, the `assertEquals` method is used to compare the output of the `asString` method. When we run the test, it will fail, as the owner of both `Porsche` objects is still `"Marcus"`.

```java
@Test
public void shouldClonePorsche() {
    Porsche marcusPorsche = new Porsche("Marcus");
    Porsche peterPorsche = porsche.clone();
    assertNotSame(porsche, peterPorsche);

    assertEquals("Porsche of Peter",
                                  peterPorsche.asString());
}
```
Example 5

To fix this test, we need to create a method for transferring ownership of the `Porsche`. I'm going to call this method `sellTo`.

```
public void sellTo(String newOwnerName) {
    ownerName = newOwnerName;
}
```
Example 6

Now I will call the `sellTo` method on the cloned `Porsche` object, to transfer ownership of the cloned `Porsche` to `"Peter"`. As a final proof that cloning the `Porsche` object created a **fully independent** second `Porsche` object, I will test whether transferring the ownership of the cloned `Porsche` object to `"Peter"` did not influence the original `Porsche` object. In other words, I will test whether the original `Porsche` object still belongs to `"Marcus"` or not.

```
@Test
public void shouldClonePorsche() {
    Porsche marcusPorsche = new Porsche("Marcus");
    Porsche peterPorsche = porsche.clone();
    assertNotSame(marcusPorsche, peterPorsche);

    peterPorsche.sellTo("Peter");
    assertEquals("Porsche of Marcus",
                           marcusPorsche.asString());
    assertEquals("Porsche of Peter",
                            peterPorsche.asString());
}
```
Example 7

This time, when we run the test, it will pass. This proves that we have created two independent objects that can be changed independently, so our cloning method works as expected.

Using Clone on an Array

As I've said before, it is generally not advisable that the clone method be used. However, one case where I *do* recommend its use is for cloning arrays, as shown in example 8. Here, we create an array of type String. We call the array clone method and assign the new cloned array to a reference variable called copiedArray. To prove that this not just a reference to the same object, but a new, independent object, we call the assertNotSame with our original array and our newly created copiedArray. Both arrays have the same content, as you can see when we print out the copiedArray in a for-each loop.

```
@Test
public void shouldCloneStringArray() {
    String[] array = { "one", "two", "three" };
    String[] copiedArray = array.clone();
    assertNotSame(array, copiedArray);
    for (String str : copiedArray) {
        System.out.println(str);
    }
}
```
Example 8

The clone method copies every String object in the array into a new array object that contains completely new String objects. If you ran the code above, the clone method would work as expected and the test would be green. In this case, the clone method is the preferred technique for copying an array. clone works great with arrays of primitive values and “immutable objects”, but it doesn’t work as well for arrays of objects. As a side note, in the Java Specialists’ Newsletter Issue 124, by Heinz Kabutz, a test was performed that showed that the clone method is a bit slower for copying very small arrays, but for large arrays, where performance matters most, it’s actually faster than any other method of copying arrays.

Alternatives to Clone: The Copy Constructor

There are two recommended alternatives to using the `clone` method that deal with the shortcomings of `clone`. The first method is using a **copy constructor** that accepts one parameter – the object to clone. A copy constructor is really nothing very special. As you can see in Example 8, it is just a simple constructor that expects an argument of the class it belongs to.

The constructor then copies (clones) all sub-objects. If the new object just references the old sub-objects, we call it a shallow copy. If the new object references truly copied objects, we call it a deep copy. You can learn about this topic by reading the chapter Shallow vs Deep Copy.

```java
package com.marcusbiel.javaBook;

public class BMW implements Car {

    private Name ownersName;
    private Color color;

    public BMW(BMW bmw) {
        this.ownersName = new Name(bmw.ownersName);
        this.color = new Color(bmw.color);
    }
}

```
Example 9

Alternatives to Clone: The Static Factory Method

The other alternative is a static factory method. As the name implies, a static factory method is a static method, used to create an instance of the class. It can also be used to create a clone. There are a few common names for static factory methods; I'm going to use newInstance. I've also created the same method for Name and Color, in order to recursively perform the operation on all sub-objects.

```java
package com.marcusbiel.javaBook;

public class BMW implements Car {

    private Name ownersName;
    private Color color;

    public static BMW newInstance(BMW bmw) {
        return new BMW(Name.newInstance(bmw
            .ownersName), Color.newInstance(bmw.color));
    }

    public BMW(Name ownersName, Color color) {
        this.ownersName = ownersName;
        this.color = color;
    }
}

```

Example 10

Both of these alternatives are always better than the `clone` method and produce the same result: a clone of the original object. The static factory method might be a bit more powerful than the copy constructor in certain cases, but generally both lead to the same result.

Note:

One of the things that make static factory methods slightly more powerful than public constructors is their ability to have a polymorphic return type. For example, you can declare a static factory method to return a `Car`, but then return a `BMW` in the method's implementation. In the case of copying objects, this advantage isn't that significant, and a public copy constructor will serve you just as well. However, there are cases where a polymorphic return type can be incredibly useful in encapsulating certain parts of a system. For more information, check out the chapter on polymorphism, and my blog post about encapsulation.

Immutable Classes

The last thing I'd like to do in this chapter is take a look at the `Name` class:

```java
package com.marcusbiel.javaBook.car;

public class Name {
    private String firstName;
    private String lastName;

    public static Name newInstance(Name name) {
        return new Name(name.firstName, name.lastName);
    }
}

```

Example 11

Notice however, that the `String` instances are passed by reference, without creating a new object. The reason for that is that `String` objects are immutable. An immutable object is an object that, once created, can never be changed. Therefore it can safely be shared without the need for cloning.

That's all I have to say about the `clone` method. For more information on the different types of object copies, feel free to check out the Shallow vs Deep Copy chapter.

Chapter 21

The Java Object toString() Method

The `Object toString()` method returns a string that represents the object to the user that can be printed to the console or a user interface. Let's take a look at the default `toString()` method of the class `Object`.

```java
public String toString() {
    return getClass().getName() + '@'
                     + Integer.toHexString(hashCode());
}
```

Example 1

Example 1 shows how the `toString()` method concatenates the `getClass().getName()` method, an '`@`' symbol and a hexadecimal value for the object's hashCode to create a string that represents the object. The `getClass()` method returns the runtime class of the class the object belongs to. The `getName()` method then returns the full-fledged class name. For example, if you have a class "`BMW`" in the package "`com.cleancodeacademy.javaBook`" and you instantiate a `BMW` object, a call to `getClass().getName()` will return "`com.cleancodeacademy.javaBook.BMW`". The `Integer.toHexString(hashCode())` method creates a hexadecimal representation of the object's hashCode. Here is a brief example of a method that would utilize a `toString()` call:

```java
@Test
public void shouldConvertBMWToString() {
    BMW bmw = new BMW(new Name("Marcus", "Biel"), new
                                              Color("silver"));
    System.out.println(bmw.toString());
    System.out.println(bmw);
}
```

Example 2

Both of the `System.out.println()` lines in Example 2 call `Object toString()`. This is because the `println()` method is overloaded, meaning that it exists in several different variations that expect different arguments. The first variation is expecting to print a string. Meanwhile, the second call is expecting an object, which it then proceeds to call the `String.valueOf()` method, which will then call the `toString()` method. Please note that in production, generally speaking, you should use logging instead of `System.out.println()`. While `System.out.println()` works well for debugging or diagnostic information in the console, it lacks the flexibility of logging in terms of output. A logger also normally yields better performance.

Returning to the method above, either `System.out.println()` call will return `BMW@e2144e4`. That String isn't very useful to us, especially if we are debugging the code and trying to understand the current state of the object. Presumably, if we are calling a `BMW` object `toString()` we know it's a `BMW` object. For that reason, you should override the `toString()` method for most entity classes.

Overriding the Object toString() Method

```java
package com.cleancodeacademy.javaBook.car;

public class BMW implements Car, Cloneable {

    private Name ownersName;
    private Color color;

    public BMW(Name ownersName, Color color) {
        this.ownersName = ownersName;
        this.color = color;
    }
}

```

Example 3

Here you can see the `BMW` class that I referenced in my previous example. As you saw in the last section, when we call `println(bmw.toString())`, we get something like `BMW@e2144e4`. That is because we have not overridden the `toString()` method as of yet. Before we override the method, we should define what we want it to return. In the case of this class, it has two attributes: `ownersName` and `color`. We also may want to return what type of class the object is, and we can easily do that by calling the `getClass()` method I highlighted before.

```java
@Override
public String toString() {
    return getClass().getName() + " [" + ownersName
                                + ", " + color + "]";
}
```

Example 4

Above, I have overridden the `toString()` method for the `BMW` class. I used the `@Override` as a tool that I can use even though it is not necessary for the code to run. It causes my compiler to make sure that I'm actually overriding a method (and not just writing a new method), and allows someone reading my code to realize that I'm overriding a method. Another point that I'd like to highlight is that I'm not writing `color.toString()`. This is unnecessary because the "+" sign between Strings allows the compiler to realize that I am concatenating strings, and automatically calls the `toString()` method for these objects.

```
@Test
public void shouldConvertBMWToString() {
    BMW bmw = new BMW(new Name("Marcus", "Biel"), new
                                        Color("silver"));
    System.out.println(bmw.toString());
}
```
Example 5

If I run this method again, assuming that we have created the `Name toString()` and the `Color toString()` methods, our output will now be "`BMW [Marcus Biel, silver]`". Now when we call the `toString()` method we have something more meaningful than the hashCode that we can print to the console, log, or print to a user interface that will allow the user to see the content of the object.

StringBuilder: An Alternative to String Concatenation

The final thing I'd like to highlight in this chapter is the `StringBuilder` class. String-concatenation with the "+" can cost a small amount of performance per call, and if you have a loop concatenating millions of strings this small difference could become relevant. However, since the compiler will replace string concatenation and use a `StringBuilder` in most cases, you should go for the code that is the most readable first. Further optimize for performance only when needed, covered by tests.

Below, here is an alternative `toString()` method that uses `StringBuilder` rather than concatenating the string. It will create the string dynamically, without all the plusses.

```java
@Override
public String toString() {
    return new StringBuilder("BMW [").append(ownersName)
     .append(",").append(color).append("]").toString();
}
```

Example 6

Chapter 22

Static Imports, Data Types, and More!

In this chapter, I will explain a variety of topics that haven't yet been covered.

Static Imports

In this section I'm going to discuss static imports. In Example 1 below, we can see a small, simple test class that imports two classes of the JUnit framework, `Test` and `Assert`:

```java
package com.cleancodeacademy.javaBook;

import org.junit.Test;
import org.junit.Assert;

public class DemoTest {

    @Test
    public void shouldDemonstrateStaticMethodCall() {
        Assert.assertTrue(true);
    }
}

```

Example 1

In line 10, we are calling the static `assertTrue` method of the `Assert` class. As it is a static method, we have to prefix the method with its class name. There is, however an alternative way of importing and using static methods, as Example 2 below demonstrates:

```
4  import static org.junit.Assert.assertTrue;
```

Example 2

This is a so called “static import”. It allows us to use the `static` method `assertTrue` *without* having to prepend its class name, as you can see in Example 3, line 10, below:

```
 1  package com.cleancodeacademy.javaBook;
 2
 3  import org.junit.Test;
 4  import static org.junit.Assert.assertTrue;
 5
 6  public class DemoTest {
 7
 8      @Test
 9      public void shouldDemonstrateStaticImport() {
10          assertTrue(true);
11      }
12  }
13
```

Example 3

An even more flexible alternative is demonstrated in Example 4 below:

```
4  import static org.junit.Assert.*;
```

Example 4

The * allows you to use all currently visible static variables and methods of a class, without having to prepend the class name.

You should note that if a static method or variable with the same name exists in two different classes, you can only import one method by using a static import. For the second method you would have to use the full fledged name, similar to the regular import statement.

Note:
The ability to statically import methods is a useful feature of the Java language, and its use can make code clearer and less cluttered. However, care should still be taken when using the feature, as it could also have exactly the **opposite effect**.
In short, before statically importing a method, first consider whether or not each one's use would be clear **in that context** for other developers reading the code.

Data Types and their Default Values

In this next section of the chapter I'm going to discuss the default initial values for instance and static variables. Let's have a look at all possible default values in Java by simply printing them out:

```java
private byte myByte;
private short myShort;
private int myInt;
private long myLong;
private float myFloat;
private double myDouble;
private Object myObject;
private boolean myBoolean;
private char myChar;

/* continued below */

```

```java
@Test
public void shouldDemonstrateDataTypeDefaultValues() {
    System.out.println("byte default value: " + myByte);
    System.out.println("short default value: " +
                                          myShort);
    System.out.println("int default value: " + myInt);
    System.out.println("long default value: " + myLong);
    System.out.println("float default value: " +
                                          myFloat);
    System.out.println("double default value: " +
                                         myDouble);
    System.out.println("Object default value: " +
                                         myObject);
    System.out.println("boolean default value: " +
                                        myBoolean);
    System.out.println("char default value: " + myChar);
    System.out.println("char default value as int: " +
                                     (int) myChar);
}
```

Output:

```
byte default value: 0
short default value: 0
int default value: 0
long default value: 0
float default value: 0.0
double default value: 0.0
Object default value: null
boolean default value: false
char default value:
char default value as int: 0
```

Example 5

For all of the number types, their default values are zero. Objects are set by default to null, and booleans are by default false. You may wonder why the char default value appears to be blank. The default value of char is actually '\u0000', which is known as the null character. As you can see, there is no visual representation of this character. However, if you convert the default char value to an int, it is printed as "0".

Number Types

There are seven different number types that can be utilized in Java. I'm going to explain them for you now in more detail. The first four number types that I'll highlight are the `byte`, `short`, `int`, and `long` datatypes. All four of these can only store integer values. However, they have different ranges, as you can see in the table at the end of this section. Usually, programmers use int because the difference in memory space between number types is, in most cases, practically irrelevant these days. In some cases you might need to use long if your numbers are too large to be stored as an int.

Float and double are two floating point number types. Again, you can see their ranges below. If you have a decimal value in your code, it is automatically considered a double. If you want to store it as a float, you have to add a capital or lowercase `F` at the end of the decimal value, for example, you'd type '`13.63F`' instead of just '`13.63`'. If, however, you wrote `float myFloat = 13.63`, that would cause an error indicating that the compiler has found a `double` when a `float` is required. You would have to instead type '`float myFloat = 13.63F`.'

Number Ranges:

Number Type	**Number of Bytes**	**Minimum Value**	**Maximum Value**
byte	1	-128	127
short	2	-32768	32767
char	2	0	65535
int	4	-2147483658	2147483647

long	8	-9223372036854775808	9223372036854775807
float	4	1.4E-45	3.4028235E28
double	8	4.9E-324	1.7976931348623157E308

Table 1

Signed vs. Unsigned Data Types

Another thing I'd like to highlight is the difference between signed and unsigned data types. Signed means that if you were to print the value of the data type, you might see a negative sign. For example, the `byte` has a range from `-128` to `127`. You might wonder why the range ends with `127` instead of `128`. This is because Java uses that space to store the positive or negative sign of the number.

An example of an unsigned data type is a `char`. A `char` stores a single character. However, you can cast a `char` to a number as shown in Example 6 below. In this case it is always a positive number, making the valid range of `char` from `0` to `65535`.

```
char myChar = ‘s’;
System.out.println("char default value as int: " +
                                         (int) myChar);
```

Output:

```
116
```

Example 6

Wrapper Types

Another nuance about primitive types is that they exist in parallel as objects, known as "Wrapper Types".

```
Byte b = Byte.valueOf(myByte);
```

Example 7

In Example 7 above, we have a variable `b` initialized using the `valueOf()` method. The `valueOf()` method is a `static` method that converts the primitive data type `byte` into a `Byte` object. By using `valueOf()`, a cached `Byte` object is returned which will save us some memory. To create a fresh object, you would say, ‘`new Byte(myByte)`’, but generally that should not be necessary.

One of the reasons wrapper objects are useful is that they can be used in Collections. Collections are a structure similar to arrays, but they’re much more flexible. Collections are very useful because you can throw objects in and take them out at will. However, you cannot throw primitive types into a collection. To work around this, we have number objects that are “wrapped” around primitive values.
All number wrapper types extend the abstract type Number. By doing this, they all implement
Number functions and can be handled uniformly. This allows them to be converted back into primitive types.

Auto-Boxing and Auto-Unboxing

In Java 5, automatic conversion of numbers to and from their corresponding wrapper objects was introduced, called auto-boxing and auto-unboxing, respectively. It is heavily used by developers, however, I recommend that you avoid using it because it could cause nasty `NullPointerException` errors or performance issues. All primitive values can’t be null, so using them will never throw a *NullPointerException*. However, when you don’t initialize an object, the value will be null and then when you call, say *b.byteValue()*, this throws a `NullPointerException`. You won’t actually see it when you write the code, because the compiler will automatically convert the wrapper object into the primitive data type, but you will see it when you run the code. Instead, you should typically use the static *valueOf()* method to convert your primitive values to their wrapper types.

Base 2, Base 8, and Base 16

Java enables us to not only save numbers using Base 10, but also Base 2, Base 8, and Base 16. This can be useful if, for example, you have a hexadecimal value in your documentation and you want to have the same value in your code. No matter what number system you use, it has nothing to do with how the values are stored. The computer will always store them in memory as zeros and ones, no matter the format. Moreover, they will by default be printed out in Base 10 just like a regular number.

Base 2 was introduced in Java 7. Before that, you could only store in Base 10, Base 8, and Base 16. In the code below, you can see some values in various bases. In the example below, I apply underscores to make the numbers more readable. You can add them almost anywhere and in any amount. The only places you cannot add an underscore are at the beginning or end of the number.

```java
@Test
public void shouldDemonstrateBases(){
    int binary = 0B1010_1011_0111;
    int octal = 017;
    int hex = 0xAB_45_CB;
    System.out.println(binary);
    System.out.println(baseEight);
    System.out.println(hex);
}
```

Output:

```
2743
15
11224523
```

Example 8

As demonstrated in Example 8 above, you can initialize primitive variables also with values from alternate number systems. Also you should notice that they are printed in decimal format by default.

A binary value is indicated by preceding the value with "0B", as line 12 shows. For octals (base 8), the value starts with a “0”. A hexadecimal value is indicated by preceding the value with "0x", as in line 14.

We've covered quite a few different topics here, focusing a lot on variables. We had a look at static imports, as well as the default initial values for instance and static variables. We looked in detail at number types and their ranges, and examined the difference between signed and unsigned data types. We also dealt with wrapper types, then looked briefly at auto-boxing and auto-unboxing, and why you should avoid using them. We ended with a quick look at different number bases that we can use to store numbers in Java. That's quite a few different topics, but I hope you've managed to follow along.

Chapter 23

Java Collections Framework

In this chapter, you will be given a high-level introduction to the Java Collections Framework. The term ‘Collection’ has several meanings, unfortunately. To clear things up, we will first discuss the word's different meanings.

‘Collection’ can refer to:

- its day-to-day meaning as *a compilation or group of things*.
- the collection of interfaces and classes that make up the Java Collections Framework.
- some data structure like a box or container that can hold a group of objects like an array.
- the `java.util.Collection` interface, one of the two main interfaces of the framework.
- `java.util.Collections`, a utility class which can help to modify or operate on Java collections.

This piece is based on Chapter 11 of the OCA/OCP Study Guide, a book packed with knowledge on Java programming. As a great fan of the authors, Kathy Sierra and Bert Bates, I recommend that you read the book even if you don’t plan on being a certified Java programmer.

What is the Java Collections Framework, from a high-level perspective? First of all, it is a library, a toolbox, of generic interfaces and classes. This toolbox contains various collection interfaces and classes that serve as a more powerful, object-oriented alternative to arrays. Collection-related utility interfaces and classes also make for better ease of use.

Overview

In this section, we will be going into more detail as we delve into the interface and class hierarchy for collections. Unlike arrays, all collections can dynamically grow or shrink in size. As I said before, collections hold groups of objects. A *map* can store strongly-related *pairs* of objects together, each pair being made up of a *key* and a *value*. A value does **not** have a specific position in a map, but can be retrieved using the key it is paired with. Don’t worry if this is too much to take in right now as we will take a more detailed look later on.

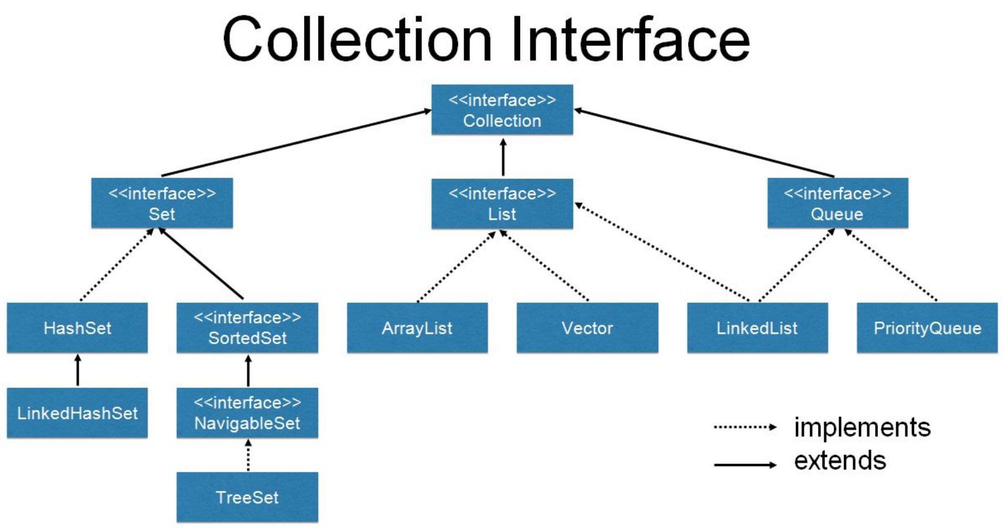


Figure 1, the Collection hierarchy

Figure 1 shows the hierarchy of classes and interfaces extending or implementing the Collection interface – it would be useful to at least familiarize yourself with the names listed there. The Collection interface sits on top of a number of sub-interfaces and implementing classes. A Collection can hold a group of objects in different ways which Set, List and Queue provide. A Set is defined as a group of unique objects. What is considered to be unique is defined by the `equals` method of the object type it holds. In other words, a Set cannot hold two equal objects. A List is defined as a sequence of objects. In contrast to a Set, a List can contain duplicate entries. It also keeps its elements in the order in which they were inserted. A Queue has two sides. Entries are added to its tail end, while entries are removed from the top or the head. This is often described as "first-in, first-out" (FIFO), which is often much like waiting in line in real life, i.e. the first person queuing up is the first person to leave the queue.

The Set Interface

HashSet, LinkedHashSet and TreeSet

`HashSet`, `LinkedHashSet` and `TreeSet` are implementations of `Set`, located around the left end of the `Collection` interface hierarchy in Figure 1. `HashSet` is the default implementation used in most cases. `LinkedHashSet` is like a combination of `HashSet` and `List` in that it does not allow duplicate entries as with a `Set`, but traverses its elements in the order they were inserted, like a `List` would do. `TreeSet` will constantly keep all its elements in some sorted order. Keep in mind, however, that there is no such thing as a *free lunch* and that every added feature comes at a certain cost.

SortedSet and NavigableSet

After looking at three classes implementing `Set`, let's also take a look at the two sub-interfaces we haven't talked about yet. As the name implies, `SortedSet` is a `Set` with the property that it is always sorted. The `NavigableSet` interface, added with Java 6, allows us to navigate through the sorted list, providing methods for retrieving the next element greater or smaller than a given element of the `Set`.

The List Interface

ArrayList and LinkedList

`ArrayList` is the default implementation for `List`, located to the middle of the collection hierarchy in Figure 1. Like any `List` implementation, it does allow duplicate elements and iteration in the order of insertion. As it is based on arrays, it is very fast to iterate and read from, but very slow to add or remove an element at random positions, as it has to rebuild the underlying array structure. In contrast, `LinkedList` makes it easy to add or remove elements at any position in the list, while being slower to read from at random positions.

Vector

As a side note, we briefly mention `java.util.Vector`, a class that has been around since JDK 1, before the Collections Framework which was added with Java 2. Long story short, its performance is suboptimal, so new code should never use it. An `ArrayList` or `LinkedList` simply does a better job.

The Queue Interface

Lastly, we take a look at the classes implementing `Queue`. Another thing to mention about `LinkedList` is that while it implements `List`, it actually also implements `Queue`. It does so based on the fact that its actual implementation as a doubly-linked list makes it quite easy to also implement the `Queue` interface.

PriorityQueue

Besides `LinkedList`, another common `Queue` implementation is `PriorityQueue`. It is an implementation that keeps its elements ordered automatically. It has functionality similar to `TreeSet`, except that it allows duplicate entries.

The Map Interface

We now take a look at the `Map` interface, one which has no relation to the `Collection` interface. A `Collection` operates on one entity, while a `Map` operates on two: a unique key, e.g. a vehicle identification number, and an object related to the key, e.g. a car. To retrieve an object from a `Map`, you would normally use its key. Map is the root of quite a number of interfaces and classes, as depicted on Figure 2.

Hashtable, HashMap, LinkedHashMap and TreeMap

The `Hashtable` class was the first collection that was based on the hash table data structure. However, as with Vector, new code must never use Hashtable, because of its suboptimal performance. We can forget about it and use the other `Map` implementations instead.

`HashMap` is the default implementation of an unordered, unsorted `Map`. `LinkedHashMap` is an implementation of an ordered `Map`. It allows us to iterate the map in the order of insertion. Finally, `TreeMap` is a sorted `Map`. It automatically sorts all the elements it stores.

Basically, if you need an ordered `Map`, go for a `LinkedHashMap`. If you need a sorted `Map`, go for a `TreeMap`. In all other cases, go for a `HashMap`, as it offers the best performance.

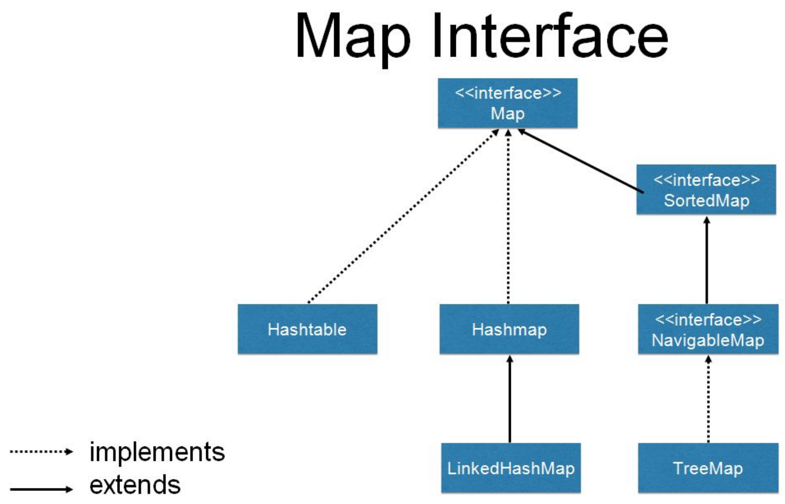


Figure 2, the Map hierarchy

SortedMap

Let’s look at the interfaces that extend `Map`. As the name implies, `SortedMap` extends `Map` and defines the contract of a constantly sorted map. `NavigableMap` takes it even further, adding methods to navigate sorted maps. This allows us to get all entries smaller or bigger than a given entry, for example. There are actually many similarities between the `Map` and `Set` hierarchies. The reason is that `Set` implementations are actually internally backed by `Map` implementations.

The Bigger Picture

You might have noticed that Java’s `Collection` classes often contain data structures based on their name. To choose the best collection for a given situation, you have to compare and match the properties of data structures like `LinkedList`, `Hashtable` or `TreeSet` to the problem at hand. There is no single best option as each one has its own advantages and disadvantages. This overview has only covered a tiny section of the huge scope of `Collection` and `Map` classes. There are even concurrent containers in the Java Collections.

Generics

The subject of generics is very broad. For now, we'll just look at the minimum we need to be able to understand the Java Collections Framework. Save your questions for later- all will be made clear in time!

```
List<String> myList = new ArrayList<String>(100);
```

Example 1

Notice the usage of the angle brackets. To the left side, we define a `List` variable `myList` with the `String` parameter in the angle brackets. We tell the compiler that the `myList` variable will never refer to anything other than a list of `String` objects. We then create an object of type `ArrayList` and again tell the compiler that the list is supposed to only contain `String`s. In other words, this is what makes the containers *type-safe*. Also note the use of the `List` type for the variable instead of `ArrayList`. This makes our code more flexible. You will only ever create the object once, but you will end up using it in many places. That being said, when you declare a `List` instead of an `ArrayList`, you get to replace the `ArrayList` with a `LinkedList` later on, and all you had to change was that one line of code.

```
Collection<String> myList = new ArrayList<String>(100);
```

Example 2

In case you don't really need methods specific to List, you could take it a bit further and use `Collection` instead. It is a good idea to always use the least specific, smallest interface possible as a variable type. Note the use of `100` as the `ArrayList` constructor argument. In this case, it's a performance optimization. Since `ArrayList` and all hash-table-based collections internally operate on arrays, when such a collection grows in size, it creates larger arrays on the fly and transfers all contents from the old array to the new one. Although this takes some extra time, modern hardware is so fast that this usually isn't a problem. On the other hand, knowing the exact, or even just an approximate, size for the collection is better than settling for the default size. Knowing what data structures Java collections are based on helps you to get a better understanding of performance in cases like this one. Paying attention to such small details is often the difference between a regular developer and a software craftsman.

```
Map<VIN, Car> myMap = new HashMap<>(100);
```

Example 3

See how a `Map` is declared and how `HashMap` is constructed above. A map is a relation of one identifying **key element** to one **value element** and both can be of different types. In the example above, `VIN` – the vehicle identification number – is used as the key while a `Car` object is the value. The *type parameters* are added as a comma-separated list in angle-brackets. As of Java 7, if you declare the variable and create the object all in the same line, you can leave the second pair of angle brackets empty as the compiler infers the type of the object from the generic type of the reference variable. The empty angle brackets are called the *diamond operator*, because of their shape.

What has been discussed so far is just the usage of generic types, where we define the types for the classes to operate on. All of this is only possible if some **method**, **interface**, or **class** has been defined to be used in a generic way beforehand.

Writing Generic Code

Example 4 shows a generically defined interface. In the first line, the interface is defined as one operating on two generic types that will have to be specified at a later time. When these types are locked in, the types the interface methods use are automatically specified. If you see one-letter types in code, it could mean that it can be used in a generic way.

```java
public interface MyInterface<E, T> {
    E read();

    void process(T object1, T object2);
}
```

Example 4

Other Utility Interfaces

- `java.util.Iterator`
- `java.lang.Iterable`
- `java.lang.Comparable`
- `java.lang.Comparator`

Listed above are some additional utility interfaces from the Java Collections Framework. They are implemented by classes of the framework or the JDK in general. Additionally, they can also be implemented by your own classes, leveraging the features and interoperability of the Collections Framework. Strictly speaking, `java.lang.Iterable` is not part of the framework, but more precisely sits on top of it. It is the super-interface of `java.util.Collection`, which means that every class that implements Collection also implements `java.lang.Iterable`.

java.util.Iterator

- `boolean hasNext();`
- `E next();`
- `void remove();`

An iterator is an object that allows us to traverse a collection, in the same way that a remote control traverses through the channels of a TV.

`hasNext()` returns true if a collection has more elements, `next()` returns the next element in the iteration, while `remove()` removes the last element returned by an iterator from its underlying collection.

java.lang.Iterable

- `Iterator iterator()`

`Iterable` provides only one method which returns an `Iterator`. Every `Collection` that implements this interface can be used in the **for-each loop**, greatly simplifying the usage of your home-made collections. To make the for-each loop available for your collection, just execute two simple steps: First, write an `Iterator` for your collection and implement at least its `hasNext`, and `next` methods. Second, implement the `Iterable` interface by adding an `iterator` method that returns an instance of the `Iterator` implementation you wrote in the first step.

java.lang.Comparable

- `int compareTo(T o)`

Implementing the `java.lang.Comparable` interface defines a natural sort order for your entities. The interface contains only one method you need to implement, `compareTo`, which compares your Comparable with **`T o`**, the argument representing another entity of the same type. Return a *negative integer* if the object is less than the given argument **`o`**, *0* if the object is equal to the **`o`**, and a positive integer if the object is greater than **`o`**.

What it means for one thing to be “lesser or greater than another” is for you to define. For numbers, it would easily follow that 1 is smaller than 5. But what about colors? This entirely depends on what *you* believe to be the natural ordering of your entities. When you put `Comparable` objects into a `TreeSet` or `TreeMap` for example, it will use your custom-built `compareTo` method to automatically sort all elements in your collection. As you can see, the Java Collections Framework has been greatly designed with extension in mind, offering many possibilities for you to plug in your own classes.

java.lang.Comparator

- `int compare(T o1, T o2)`

This interface is very similar to `Comparable`. It allows you to define additional sorting orders, e.g. a reverse order. The sorting logic is not directly implemented in your entity class. Instead, it is defined in an external sorting strategy class that can optionally be attached to a `Collection` or a sorting method to define alternative sorting orders for your collections of entities. The same rules for the interface contract of `Comparable` apply: return a negative integer if the first argument, `o1`, is less than the second argument `o2`, `0` if both arguments are equal, and a positive integer if `o1` is greater than `o2`.

Collections and Arrays

Last, but not least, we take a look at the two utility classes `java.util.Collections` and `java.util.Arrays`. Like a Swiss army knife, both provide static helper methods that greatly enhance the general usefulness of the `Collection` classes. `Collections` offers methods like `sort`, `shuffle`, `reverse`, `search`, `min`, and `max`. `Arrays` is actually quite similar to `Collections` except that it operates on raw arrays, i.e. it allows us to sort or search through arrays, for example.

Summary

That's all you need to know for now about the Java Collections framework. In summary, the Java Collections framework is one of Java's most powerful subsystems that every Java developer needs to know. It consists of a set of classes that represent containers, and utility classes that help you to operate on these containers. The containers serve to store and hold similar objects.

Make sure you know the different usage scenarios as well as the pros and cons of the most important containers inside out!

For further details, you can look up the great Java 8 Collections Framework API anytime.

Chapter 24

ArrayList

In this chapter, I will be giving you a basic overview of the Java class `java.util.ArrayList`. I will first explain the meaning of the size and capacity of an `ArrayList` and show you the difference between them. After that, I will explain some `ArrayList` methods, divided between the interfaces `Collection` and `List` to which the methods belong. I will finish off by giving you a few practical coding examples that will, for instance, show you how to add and remove elements from an `ArrayList`.

java.util.ArrayList

`ArrayList` implements the `List` interface, which again extends the `Collection` interface. Figure 1 below shows an overview of the most important classes that implement the `Collection` interface, including the class `ArrayList` and how it fits in.

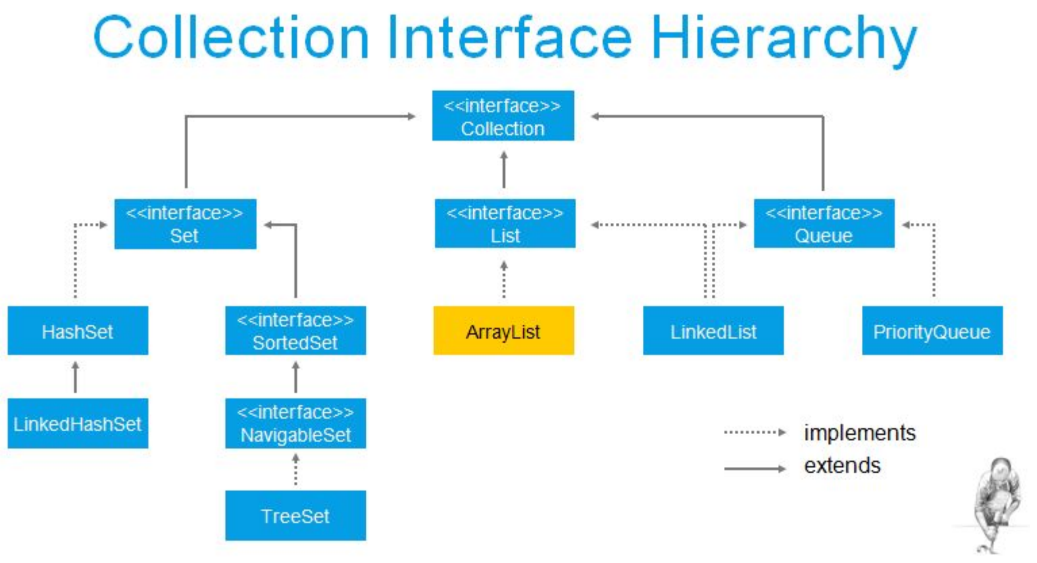


Figure 1

As is typical of `List` implementations, we can have duplicate elements in our `ArrayList` and we can go from element to element in the same order as they were inserted. As the name implies, `ArrayList` is based on an array data structure. Therefore, `ArrayList` provides fast access, but slow element insertion and removal at arbitrary positions, as changes to it require reorganizing the entire list. Fast access, however, is crucial for most applications, which is why `ArrayList` is the most commonly used collection. To store data that changes frequently, a better alternative *could* be a `LinkedList`, for example. However, keep in mind that performance must be tested - **never do premature optimization**.

Size and Capacity

There are two different terms which are important to understand in the context of an `ArrayList`: **size** and **capacity**.

Size refers to the number of elements the `ArrayList` currently holds. For every element added to or removed from the list, the size grows and shrinks by one respectively.

Capacity, on the other hand, refers to the number of elements that the underlying array can hold. An `ArrayList` starts with an initial capacity which grows incrementally. Every time that adding an element would exceed the capacity of the array, the `ArrayList` copies data over to a new array that is about fifty percent larger than the previous one. Let's say you want to add 100 elements to an `ArrayList` with an initial capacity of 10. After all the elements have been added, it will have created six more arrays to take the place of the first. More specifically, the first array is replaced with a new array that can hold 15 elements, then a second one which can hold 22 elements, then arrays with capacities of 33, 49, 73 and finally, 109 elements – all of this to hold the growing list as pictured in Figure 2 below:

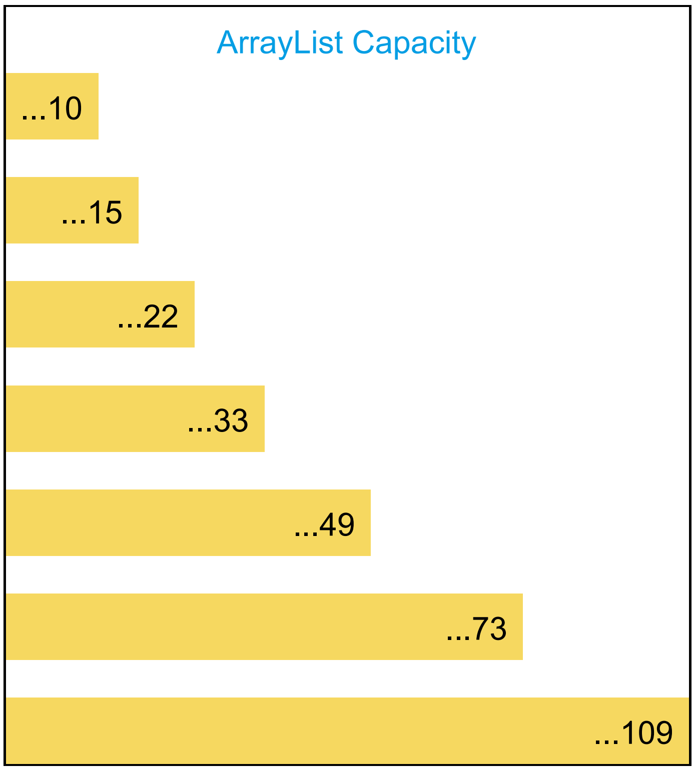


Figure 2

These restructuring arrangements may negatively impact performance. You can instantly create an array of the correct length to minimize these rearrangements by defining the correct capacity at instantiation. If you don't know the final size of the `ArrayList` before creating it, then make the best guess possible. Choosing a capacity that is too large or too small can backfire, so choose this value carefully. In addition, it is advisable to explicitly set the capacity at creation time as this documents your intentions. For most projects, you won't have to worry about optimizing performance, but that doesn't excuse sloppy design and poor implementation.

Example 1 below shows a very simplified excerpt of the class `ArrayList`.

```java
package java.util;

public class ArrayList<E> {
    private static final int DEFAULT_CAPACITY = 10;
    private Object[] elementData;
    private int size;

    public E get(int index) {
        /* implementation omitted ... */
    }

    public boolean add(E e) {
        /* implementation omitted ... */
    }

    /* other methods omitted ... */
}

```

Example 1

`DEFAULT_CAPACITY` represents the initial length of the array when you don't specify it as recommended before. `elementData` represents the array used to store the elements of the `ArrayList`. `size` represents the number of elements the `ArrayList` currently holds. `get`, `add` and `remove` are a few of the methods `ArrayList` provides.

Note:
After reading this chapter, I recommend that you take a look at the full source code of `ArrayList` on grepcode.com. Alternatively, you can find the archived source code as "`src.zip`" in the subdirectory `./lib/` of your JDK. Finally, you can also browse the source code in the JDK in your IDE of choice.

I'm emphasizing this because I really want to encourage you to take a look at the actual Java source code from time to time. It gives you a much deeper understanding of how certain classes work! Don't be afraid, the JDK code won't bite you! Go for it!

ArrayList’s Methods

In the following section, we look at some of the most common methods of the interfaces `java.util.Collection` and `java.util.List`. Both are implemented by the class `ArrayList`. In contrast to `List`, `Collection` does not provide any index or order-related methods. This is because the `Collection` interface does not guarantee any particular ordering of its elements.

java.util.Collection

`boolean add(E e)`

The method `add` appends the element to the end of the collection. For `ArrayList`, the end of the list refers to the next empty cell of the underlying array.

`boolean addAll(Collection<? extends E> c)`
Appends all given elements to the end of the `Collection`. The stuff in the angle brackets is related to generics. In short, it ensures that no one can call such a method with the wrong arguments.

`boolean remove(Object o)`
Removes the first occurrence of the element you specify from the `Collection`.

`boolean removeAll(Collection<?> c)`
Removes the given elements from the `Collection`.

`Iterator iterator(E e)`
Returns an object you usually use in a loop to go through a `Collection` one element at a time, from one element to the next. We say “we iterate over the `Collection`”, hence the name, `iterator`.

`int size()`
This method returns the current number of elements in the `Collection`.

`boolean contains(Object o)`
Returns `true` if the `Collection` contains at least one instance of the element you specify.

`void clear()`
Removes all elements from the `Collection`.

`boolean isEmpty()`
Returns `true` if the `Collection` contains no elements.

`T[] toArray(T[] a)`
Returns a raw array containing all of the elements of the `Collection`.

java.util.List

`boolean add(int index, E element)`
This method acts like an insertion method. It allows you to insert an element at any index position in the list, instead of just adding the element to the end of the list. In the process, the elements of the backing array for `ArrayList` will be shifted to the right and migrated to a larger array if necessary.

`E remove(int index)`
Removes an element from any index position of the list. Similar to `ArrayList`'s `add` method, it might require shifting the remaining elements of the underlying array to the left.

`E get(int index)`
Returns, but does not remove, an element from any given position in the list.

`int indexOf(Object o)`
Returns the index of the first occurrence of the given argument in the list or `-1` if the argument is not found.

`int lastIndexOf(Object o)`
Returns the index of the last occurrence of the given argument in the list or `-1` if the argument is not found.

`void sort(Comparator<? super E> c)`
This method sorts the list following the order of the given comparator.

Note that the methods described above are somewhat similar to the `Collection` methods that we looked at previously. They differ in that they require an ordering of the elements in the list.

Instantiating an ArrayList

Next, we will look at how best to instantiate an `ArrayList`. There is no magic involved; we simply have to create an instance of an `ArrayList` and assign it to a variable to access the object. As you can see in Example 2 below, I assign an `ArrayList` instance to a variable of type `Collection` instead of type `ArrayList`.

```
Collection<String> elements = new
                            ArrayList<>(INITIAL_CAPACITY);
```
Example 2

Doing this works because `ArrayList` implements the interface `Collection`. Why should we do this, though? Well, it makes our code more flexible and maintainable. The less we commit ourselves to concrete implementation details, the easier it will be to change our minds later on. We determine the concrete instance only in one place- where it is created. This enables us to react quickly to changing business requirements. For example, what if there was suddenly a new requirement that we should not be allowed to store duplicates in our `elements` `Collection`? Instead of an `ArrayList` we could use a `HashSet`. A `HashSet` is not an `ArrayList`, but they both implement the interface `java.util.Collection`. If we were to use an `ArrayList` reference variable, we would have to change the code in many places, and that would not only be costly but also very dangerous, because any change would risk introducing a bug. When declaring a reference variable, you should therefore always use the class or interface that is as specific as necessary to meet the business requirements, but as unspecific as possible, so that it can be adapted quickly later on if requirements change.

To quote Robert C. Martin:

> *“A good architecture is one that maximizes the number of decisions not made.”*

Don’t just hack down some code “because it works”. You should think much more carefully about what you truly need. Do you need an unordered collection of objects? Then the `Collection` interface should be sufficient. Do you need the ability to access an element at a specific position? Then you need a list.

This, by the way, is a direct application of polymorphism. Check out the chapter on Inheritance and Polymorphism for further details.

`ArrayList` also has methods that are specific to the class and that are not assigned to any interface. The method `trimToSize` is an example of this. It optimizes the memory consumption of an `ArrayList` instance by copying the element into an array that can hold the exact number of elements the `ArrayList` currently holds. If we want to call `trimToSize` in our code, we are *forced* to define a reference variable of type `ArrayList`. As in most cases, memory optimization here comes at the expense of maintainability. To preserve maintainability, we should only optimize performance when there is proof that the system is not performing well enough.

> As a side note, in Example 2 above, notice on the same line of code the use of “<>”This “diamond operator” was introduced with Java 7. Since

we declare and instantiate `elements` on the same statement, the compiler can infer that our `ArrayList` is typed to `String`. This makes our code a bit shorter and easier to read. Last, but not least, notice the initial capacity which is a constant previously defined as follows: `private final int INITIAL_CAPACITY = 5`. We could have simply typed `5` as our initial capacity, but we would have been writing a "magic number", which is a primitive or object literal whose meaning is not explained in code. This makes it much harder for the next developer reading our code to know what the `5` actually means in this context. Instead, we should document our intentions clearly and use a named value.

Collection Coding Samples

Next, we will look at some examples to help us get a better grasp of what `ArrayList` is all about. The first example is the `Collection` interface method `add` shown below in Example 3 (lines 12-17). Since we've declared elements as a `Collection` of String, we can only ever add `String` objects to it. Although our variable is a `Collection`, the fact that here we have instantiated an `Arraylist` allows us to duplicate elements such as "`A`" and "`E`". Recall earlier that the constant `INITIAL_CAPACITY`, set to `5`, has also been set as the initial capacity for our `ArrayList`.

By adding a sixth element, which is the second “E”, the ArrayList instance will internally create a new and larger array. As we saw before, all existing data will be copied to this new array.

Note:
The internal restructuring of `ArrayList` does cost performance. Nevertheless, the underlying array structure allows the fastest possible iteration and random access times. In most cases, this outweighs the time spent performing occasional restructurings. `ArrayList` is the best

general-purpose `List` implementation in the Java Collections Framework. In the few cases where it is not the optimal choice and testing reveals that the underlying static array data structure poses a problem for a program that dynamically adds and/or removes many elements from a list, consider using a [`LinkedList`](LinkedList) instead.

```
Collection<String> elements = new
                                ArrayList<>(INITIAL_CAPACITY);

elements.add("A");
elements.add("B");
elements.add("A");
elements.add("C");
elements.add("E");
elements.add("E"); /* a new, larger backing array is
                                       created here. */
```

Example 3

Printing a Collection

Here are two different ways to print out an `ArrayList`. We can use its `toString` method, which `println` internally calls when we use it as follows:

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

```
Output:
[A, B, A, C, E, E]
```

Example 4

As an alternative, we can use the `ArrayList` in a *for-each* loop and print out each element separately.

```
for (String element : elements) {
    System.out.print(element + " ");
}

```

```
Output:
A B A C E E
```

Example 5

Because `ArrayList` implements `Collection`, which extends `Iterable`, we can iterate over it using a for-each loop without worrying about the details required to iterate over the `Collection`. This is all done for us internally by the compiler.

Notice that the `Collection` is printed out in the same order as the elements were added, "`A B A C E E`" in our case. If our `Collection` had been instantiated to some instance of `Set`, we would not have been guaranteed the same iteration order as is the case with `List`.

Removing a Collection Element

We can just as easily remove elements from a `Collection`. As mentioned earlier, the `Collection` interface does not provide any index based methods. So to remove an element from a `Collection`, we specify the value to be removed. The collection, an instance of `ArrayList` in our case, then searches through each of its elements for a match to the value specified, determined by calling the `equals` method. But what happens if the collection contains the element searched for several times? Will the first, the last or maybe all of them be removed? Whenever you come across such a question, you could look for the answer in a book, a video tutorial, or the API. However, the quickest way to be sure is to write some simple test code by yourself. *Code never lies*. A code experiment will always give you a reliable answer that is beyond any doubt. In Example 6 below we will try this out:

```
10 Collection<String> elements = new
                              ArrayList<>(INITIAL_CAPACITY);
11
12 elements.add("A");
13 elements.add("B");
14 elements.add("A");
15 System.out.println(elements);
   [...]
21 elements.remove("A");
22
23 System.out.println(elements);
```

```
Output:
[A, B, A]
[B, A]
```

Example 6

First, we create a collection and add the elements “`A`”, “`B`” and “`A`” again. The output of Line 15 shows us that our collection now contains the element “`A`” twice. In line 21 we call `remove("A")` on the `Collection` instance. The output of line 23 shows us that the first "`A`" element has been removed. The second "`A`" element is still included. So this answers our question. The `remove` method does indeed remove from the `Collection` the first match that's found.

For more details on how exactly the `equals` method works, check out my blog post about equals and hashcode.

Determining a Collection’s Size

The `Collection` interface also provides a method to check its current size, which can also be used to check if the collection is empty, that is, if its size is zero. In Example 7, Line 12 below, we print out the size of our `Collection` after instantiating it. As expected, `0` is printed out since we have zero elements. On Line 19, after adding four strings, we print out the size again and get `4`.

```
Collection<String> elements = new
                                ArrayList<>(INITIAL_CAPACITY);

System.out.println(elements.size());

elements.add("A");
elements.add("B");
elements.add("A");
elements.add("C");

System.out.println(elements.size());
```

```
Output:
0
4
```

Example 7

Conveniently, `Collection` also provides a method to directly check if it is empty. We get `true` on Example 8, Line 12 after creating a new, empty `ArrayList` and get `false` on Line 16 after having added `"A"` to the list.

```
10 Collection<String> elements = new
                                ArrayList<>(INITIAL_CAPACITY);
11
12 System.out.println(elements.isEmpty());
13
14 elements.add("A");
15
16 System.out.println(elements.isEmpty());
```

```
Output:
true
false
```

Example 8

Clearing a Collection

To remove all the elements from a `Collection`, we use the `clear` method. This will empty the collection so that if we call `isEmpty` on a list with elements, `true` is returned. This is demonstrated in the code excerpt in Example 9 below. Interestingly enough, on Line 22, we learn that printing out an empty collection doesn't throw an Exception. Instead, it only prints out a pair of square brackets, "`[]`". Writing a little code experiment helped us once again to gain practical knowledge of the `ArrayList` class.

```
10 Collection<String> elements = new
                                 ArrayList<>(INITIAL_CAPACITY);
11
12 elements.add("A");
13 elements.add("B");
14 elements.add("A");
15 elements.add("C");
16
17 System.out.println(elements.isEmpty());
18
19 elements.clear();
20
21 System.out.println(elements.isEmpty());
22 System.out.println(elements);
```

```
Output:
false
true
[]
```

Example 9

List Coding Samples

From here on, we switch over to the List interface in most of our examples, i.e. we will now have a reference variable of type List, giving us access to all of the Collection interface’s methods and List’s on top of that.

Inserting a List Element

We can insert an element by specifying where to place it using an index. Example 10, Lines 16-17 shows how we can add, say, “T”, to the start of our list at index 0 and “S” at index 2 – which ends up being right between “A” and “B” after inserting “T”. Both insertions shift all other elements to the right of the specified index, whereas removing elements by index does the opposite, shifting all other elements to the left. Printing it out as shown on Line 19, we get the output [T, A, S, B, A, C].

```
10 List<String> elements = new
                                 ArrayList<>(INITIAL_CAPACITY);
11
12 elements.add("A");
13 elements.add("B");
14 elements.add("A");
15 elements.add("C");
16 elements.add(0, "T");
17 elements.add(2, "S");
18
19 System.out.println(elements);
```

```
Output:
[T, A, S, B, A, C]
```

Example 10

Removing a List Element by Index

Let’s say we want to remove the second element from our list. As Example 11, Line 16 shows, this can be done by calling `remove` with index `1`. This removes `“B”` from our list, leaving us with the elements `[A, A, C]`. Note that although this index-based remove method can sometimes be useful, it can easily cause unseen errors in a program, such as when an incorrect index is passed to it.

```
10 List<String> elements = new
                                ArrayList<>(INITIAL_CAPACITY);
11
12 elements.add("A");
13 elements.add("B");
14 elements.add("A");
15 elements.add("C");
16 elements.remove(1);
17
18 System.out.println(elements);
```

```
Output:
[A, A, C]
```

Example 11

Determining a List Element’s Index

Sometimes, you will want to ask for the index of a certain element in a `List`. The `List` interface gives us two methods to do that – `indexOf` and `lastIndexOf`. Since `Lists` allow for duplicate elements, to get the first index of an element, we would use `indexOf`. If we call `indexOf` with `“A”`, we get the first index, `0`. To get the last index of `“A”`, we would call `lastIndexOf`, which returns `2`. You can see all of this in action in Example 12 below.

```
int initialCapacity = 100; /* backing array starts at
length of 100 */
List<String> elements = new
ArrayList<>(initialCapacity);

elements.add("A");
elements.add("B");
elements.add("A");
elements.add("C");

System.out.println(elements.indexOf("A"));
System.out.println(elements.lastIndexOf("A"));
```

```
Output:
0
2
```

Example 12

That’s all I have to say about `java.util.ArrayList` for now. I hope this chapter has helped you gain a better understanding of `ArrayList`, its methods and its performance characteristics.

Chapter 25

Linked List Data Structure

Terminology

First of all, let’s have a look at the term “linked list”. Why is it called that? To understand the term “link”, you can think of a hyperlink on a web page that can take you from one page to the next. To understand the term "list", think of a shopping list. It is a list of items related to one another because you plan to buy them all. So, a linked list is a collection of related objects with a link taking us from one item to the next.

Singly Linked Lists

Technically, every item in our singly linked list is “wrapped” in an object, called a “node”. In Figure 1 below, you can see an example of a singly linked list composed of four nodes that contain the numbers 23, 3, 9, and 42.

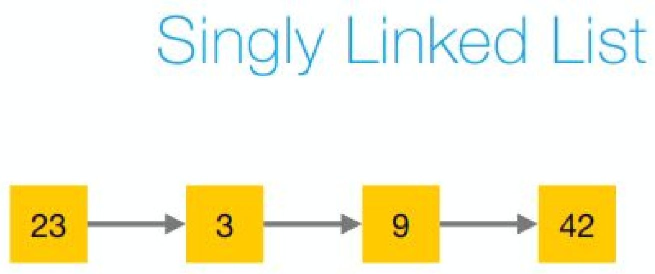


Figure 1

Each node has a *link* or *pointer* to the next element of the list – the next node. For every
item that we want to add to the linked list, we create a node and store the item in it. The first item (or beginning) of the list is usually called the *head* while the last item (or end of the list) is called the *tail*.

When you add or remove elements from the beginning or end of the list, this will of course have an effect on which element is considered to be the head or tail of the list. In Figure 1 above, the element 23 is its *current* head and the element 42 is its *current* tail.

Linked list is a dynamic data structure that can hold any number of elements, as long as enough memory is available. After the list has been created, navigating through the list is done by calling a method like `next` which follows the link from one item to the next. We call this type of linked list a singly linked list because the links only go in one direction, from the beginning of the list (head) to its end (tail). This means that when we navigate to the previous element, such as from element 42 to element 9 as in Figure 1, we have to go back to the head of the list and call the `next` function on every single element until we reach the element 9. If the list contains a lot of elements, this may take some time.

Inserting an element after the current one is relatively easy with a singly linked list. Say we start with 9 as our current element, as you can see in Figure 2:

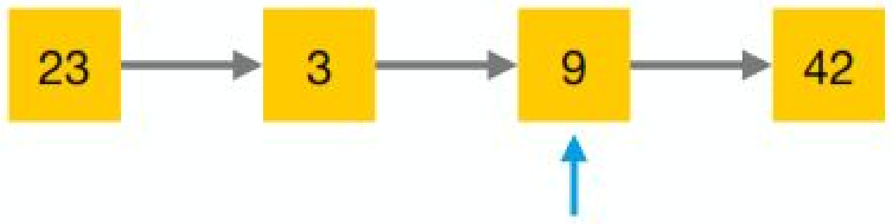


Figure 2

Now we create a new node containing the new element 17, as in Figure 3:

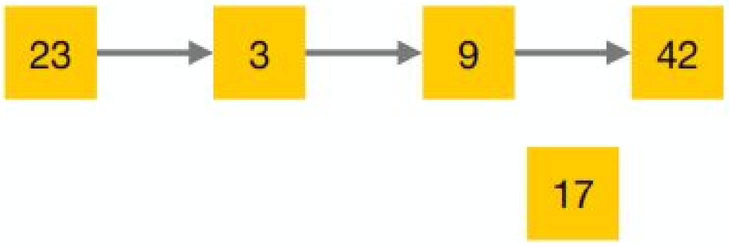


Figure 3

We then link the new element "17" from the current element "9", as in Figure 4:

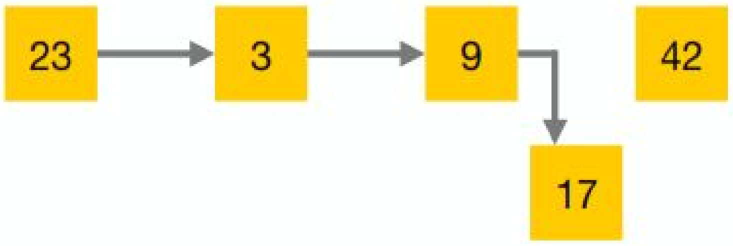


Figure 4

Finally, we add a link pointing from the new "17" element to the existing "42" element, as in Figure 5:

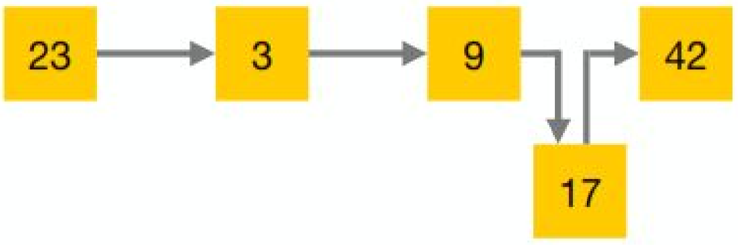


Figure 5

With that, the element has now been inserted, as you can see in Figure 6:

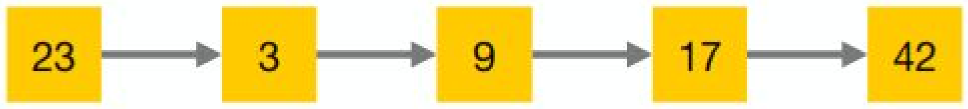


Figure 6

Inserting the same element before the current one is possible in a singly linked list, but is usually not very efficient. It requires us to navigate to the previous element, starting from the beginning of the list as shown before. Removing an element from a singly linked list has the same issue – it is possible, but is generally not efficient.

These operations become much easier when we add a second link to each node which points to the previous one. This allows us to navigate in both directions on the list, forwards and backwards. However, the extra link comes at the cost of extra system resources needed to build the more complex structure.

Whether the overhead is justified depends a lot on the use case. If performance is an issue, then different options must be tested.

Doubly Linked List

A linked list that contains nodes that provide a link to the next and the previous nodes is called a **doubly linked list**. For every element added to a doubly linked list, we need to create two links, making doubly linked lists somewhat more complicated than their singly linked counterparts. Navigating both ways in a doubly linked list is easier, but it’s done at the cost of having a more complex structure. Thanks to the two-way link structure, adding or removing an element before or after the current element is relatively easy. To demonstrate this, we will add an element “17” before the current element “9”.

Okay, let’s go step by step again. First, we create a new element “17”, as you can see in Figure 7 below:

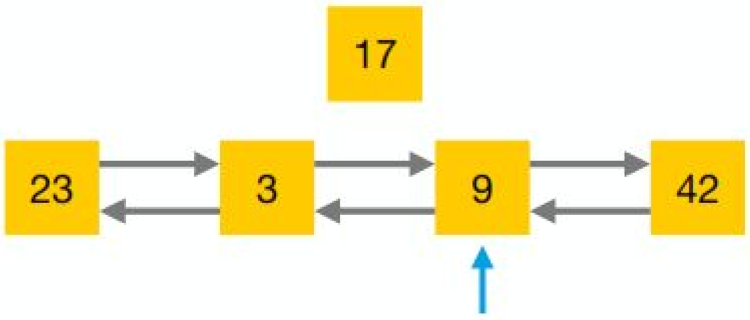


Figure 7

Then, we remove the backlink from element “9” to element “3” and replace it with a backlink to the new element “17”, as in Figure 8 below:

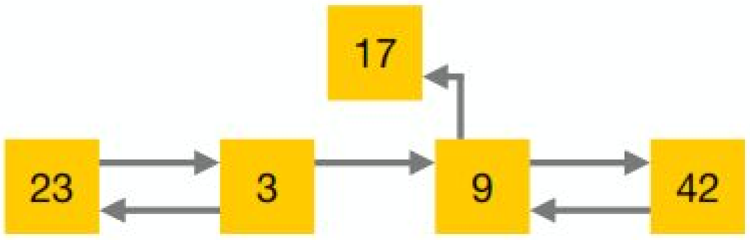


Figure 8

Next, we add a forward link from the new Element “17” to “9”, as shown in Figure 9:

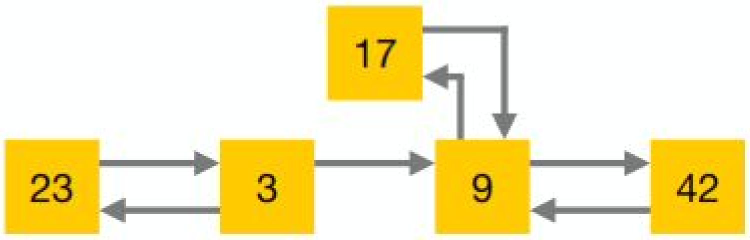


Figure 9

After that, we place a backlink from “17” to “3”, as in Figure 10:

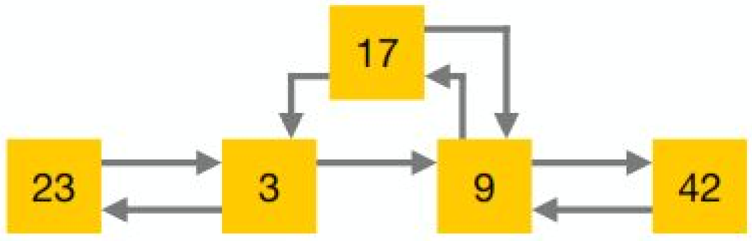


Figure 10

Finally, the old link from “3” to “9” is replaced by a link from “3” to “17”, as in Figure 11:

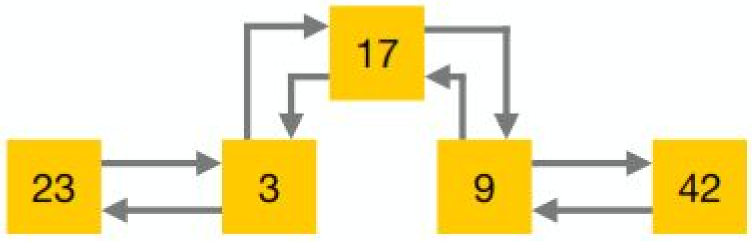


Figure 11

To remove an element from a doubly linked list, we follow the same steps, but in reverse. For example, to remove the “17” element, we redirect the link from the “3” element to “9”, and then the backlink from “9” to “7”.

Example of a Singly Linked List Implementation

Let’s look at the code in Example 1 below. It shows a singly linked list node. The method `item` (lines 12-14) returns the item of the node. The method `next` (lines 16-18) allows to go to the next node.

```java
public class Node {
    private E item;
    private Node next;

    public Node(E element, Node next) {
        this.item = element;
        this.next = next;
    }

    public E item() {
        return item;
    }

    public Node next() {
        return next;
    }
}

```

Example 1

The code of a doubly linked list node would look very similar. It would just have an additional reference and method to go to the previous node.

```java
public class LinkedList<E> {
    private Node currentNode;
    private Node headNode;

    public E get (int index) {...}
    public boolean add (E e) {...}
    public E remove (int index) {...}

    [...]
}
```

Example 2

Example 2 above shows the code excerpt of a linked list. Besides the direct link to the `currentNode` and the `headNode`, a linked list may also provide a direct link to its tail node. This is common for a doubly linked list, but is also useful to have in a singly linked one. As it stands, we can't say whether Example 2 shows a singly or a doubly linked list. Both would be possible. It depends on the concrete node implementation used as well as on the concrete implementation of the linked list methods, which we have omitted here for the sake of simplicity.

Application

You can implement many other data structures on the basis of a linked list data structure. In the following, we will take a look at which kind of linked list (singly or doubly linked) would be better suited to implement a list, queue, stack or double-ended queue (deque). Of course, each of these data structures could (and, in some cases, should) be implemented in a different way, for example, on the basis of an array. Here, however, the main objective is to further explore the differences between singly and doubly linked lists. If you want to know more about queues, deques and stacks, you may want to check out the `java.util.LinkedList` chapter.

- *Lists* usually require random access to their elements. In this case, a doubly linked list offers more flexibility, as it allows direct traversion in both directions of any given element of the list. In a singly linked list, on the other hand, going back to a previous element has to be simulated by going forward (multiple times) from the head of the list to the desired element.
- In a “first-in-first-out” (FIFO) *queue*, new elements are inserted at the tail and removed from the head of the queue, so random access is not required. This makes singly linked lists the better choice.
- A *stack* provides a “last-in-first-out” (LIFO) sequence of its operations.The requirements for a stack are very simple. You never iterate over the elements of a stack- elements are only added and removed from the head of the stack. Therefore, you should use a singly linked list for a stack, because its simple structure is more than a doubly linked list.

- A *double-ended queue* or “deque” is a very dynamic data structure. It allows access, addition and removal from both ends.Since you have to navigate all the way from the head of a singly linked list to remove from its tail, it is more efficient to implement a deque as a doubly linked list.

Although it is possible to implement these data structures with arrays, for instance, the focus of this chapter is purely on linked lists. To find out how a list might be implemented using an array, check out the previous chapter on `ArrayList`.

That’s about all I have to say about the linked list data structure. In the next chapter, we will look at the class `java.util.LinkedList`, which is a concrete implementation of a doubly linked list.

Chapter 26

java.util.LinkedList

In this chapter, I will talk about the class `java.util.LinkedList`. The Java Collections Framework has two general-purpose classes for representing lists of things, namely `LinkedList` and `ArrayList`. In the previous chapter, we covered the linked list data structure. As the name implies, `LinkedList` is internally based off a linked list.

Linked Lists and java.util.LinkedList

Linked lists and `java.util.LinkedList` actually represent two different concepts. As an analogy, think of the difference between the abstract concept of a car on paper and a real BMW.

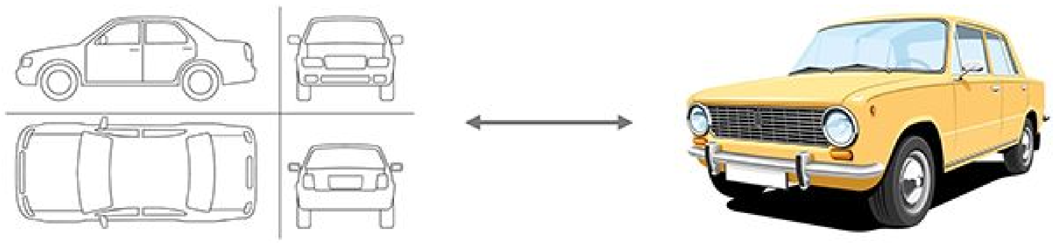


Figure 1

A linked list is an abstract concept of a data structure, independent of any programming language or platform, whereas the `LinkedList` Java class is a concrete implementation. Among other interfaces, `LinkedList` implements `java.util.List`. You can have duplicates in a `List` and you can go through each element in the same order as inserted.

Differences between ArrayList and LinkedList

As was covered in a previous chapter, `ArrayList` and `LinkedList` are similar to each other in that they both implement the `java.util.List` interface, but that is where the similarity ends. First of all, `ArrayList` is based on an array, while `LinkedList` is based on a doubly-linked list.

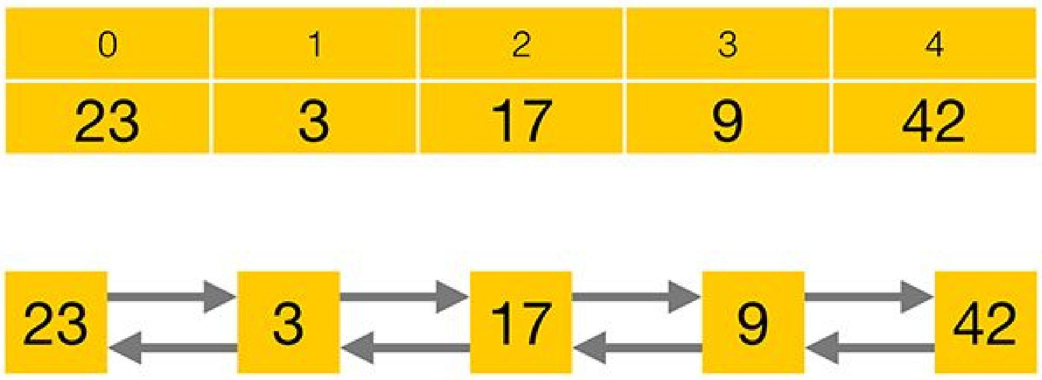


Figure 2

Since an `ArrayList` is backed by an array, which allows direct access of each element by using its index, it provides the most efficient retrieval of single elements. In a linked list, you have to navigate to the desired element, potentially traversing a large number of elements to get there.

On the other hand, the removal of an element requires shifting all the elements to the left to take up the empty space. In large lists, this can be a time-consuming operation, especially when the element to be removed is located at the beginning of the list.

Given those characteristics, you cannot automatically say that an `ArrayList` or a `LinkedList` is best for a certain scenario, since it depends on a variety of factors. For example:

1. How large will the list be?
2. How many elements will be removed?
3. Where are the elements to be removed located?
4. How many times do you need to iterate over the list?

Also, what might be true on one day, or on one machine, might not be true on another. So, to know for certain which type of list is best used in your case, the performance of each must be tested.

That said, however, in the majority of use cases, `ArrayList` offers the best overall performance, mainly due to its fast random access. Therefore, my advice is to always start with an `ArrayList`. Should tests reveal that its underlying static array structure is too slow for the given use case, check if a `LinkedList` offers better performance.

LinkedList Implementation

```java
package java.util;

public class LinkedList implements List, Deque {
   private Node first;
   private Node last;

   public E get(int index) {...}
   public boolean add(E e) {...}
   public E remove(int index) {...}
   [...]
}
```

Example 1

Example 1 is a simplified code excerpt from the `java.util.LinkedList` class. As you see - the code is rather straightforward and is not black magic.

Note:
As described more in detail in the ArrayList chapter, you should check the actual ArrayList source code online, or by looking at the archived source code in the `./lib/` subdirectory of your JDK. However, don’t rely too much on source code internals as they may change at any time if they are not defined in the Java API.

As you can see in Example 1 above, LinkedList implements the List, Queue and Deque interfaces, as Deque extends Queue. Also, as you can see, the class has functions such as get, add, or remove, to access, insert or delete elements from the list. Finally, lines 4 and 5 of the code excerpt above show that a LinkedList maintains references to its first and last nodes. These references are shown as red arrows in Figure 3:

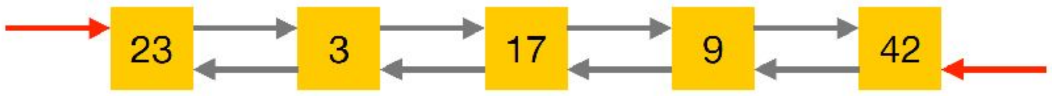


Figure 3

Every single element in a doubly-linked list has a reference to its previous and next elements as well as a reference to an item, which has been simplified as a number inside a yellow box in Figure 3 above.

```
public class Node {
   private E item;
   private Node previous;
   private Node next;

   public Node(E element, Node previous, Node next) {
      this.item = element;
      this.previous = previous;
      this.next = next;
   }
   [...]
}
```
Example 2

A code excerpt of a linked list node implementation is shown in Example 2. It has private members for the item it holds, as well as for the previous and the next nodes in the list. As users of the Collections class LinkedList, we never directly access the nodes. Instead, we use the public methods that LinkedList exposes, which operate internally on the private Node members.

Having already covered the methods of the `List` interface in a previous chapter, we move on now to the methods of the `Queue` interface as implemented by `LinkedList`.

Queue

From a high-level perspective, the `Queue` interface consists of three simple operations: **add an element** to the end (tail) of the `Queue`, **retrieve an element** from the front (head) of the `Queue` without removing it, and **retrieve and remove an element** from the front of the `Queue`.

In the lifetime of a `Queue`, there can be **special situations**, like trying to remove an element from an empty queue, or trying to add an element to a full, limited-capacity queue.

In *one* business case, those situations may be **normal, expected** and just how the system works. However, in *another* business case, they may be a sign that something has gone wrong and may be an **unexpected, exceptional situation** that needs to be addressed. To provide for both situations, the `Queue` interface presents each of its operations in two flavours, as shown in Figure 4 below.

	Throws Exception	**Returns Special Value**
Add	add	offer
Retrieve	element	peek
Retrieve & Remove	remove	poll

Figure 4

The first blue column shows the methods that throw an Exception when a “special situation” occurs. The second blue column shows the methods that return a special value in those cases - either `null` or `false`.

If you try to add an element to a full `Queue` implementation, the `add` method will throw an `IllegalStateException`, while `offer` will return `false`. However, you should know that this does not apply to a `LinkedList`. Like most `Queue` implementations, it has a virtually unlimited capacity, by “design” it will never be full. `LinkedBlockingQueue` on the other hand is a `Queue` implementation with a fixed capacity. Once this has been reached, no further elements may be added. As just described, `add` would throw an `IllegalStateException` in this case, and `offer` would return `false`.

Next up are `element` and `peek`. Both allow you to retrieve an element from the front of the queue without removing it. If the queue is empty, `element` throws an Exception while `peek` returns `null`.

Finally, you can retrieve and remove an element from the front of the queue. If a queue is empty, `remove` throws an `Exception` while `poll` returns `null`.

Deque

Now we will look at some methods from the `Deque` interface as implemented by `LinkedList`. `Deque` is short for “double-ended queue”, making it a queue that can be accessed from either end. Just like a queue, a deque allows for **adding**, **retrieving**, and **retrieving and removing** an element. However, since it can be accessed from either end, the `Queue` methods we saw before now exist in two variations – one for the first and one for the last element of the deque, as shown in Figure 5.

Add	addFirst addLast	offerFirst offerLast
Retrieve	getFirst getLast	peekFirst peekLast
Retrieve & Remove	removeFirst removeLast	pollFirst pollLast

Figure 5

Again, let's look at this in more detail. Just like the `add` method of the `Queue` interface, `addFirst` and `addLast` will throw an `Exception` when the `Deque` is full. `offerFirst` and `offerLast` will return `false` instead of throwing an `Exception`. As before, this only applies to a `Deque` implementation with a limited capacity. `LinkedBlockingDeque` is one such example.

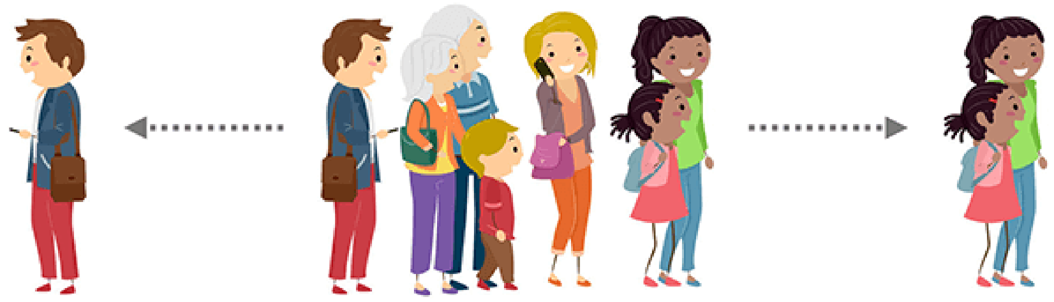


Figure 6

`getFirst` and `getLast`, as well as `peekFirst` and `peekLast`, allow you to retrieve an element without removing it. When the `Deque` is empty, `getFirst` and `getLast` will throw a `NoSuchElementException`, while `peekFirst` and `peekLast` will return `null` in this case. Finally, `removeFirst` and `removeLast` retrieve *and* remove elements from a `Deque`, or throw a `NoSuchElementException` when the `Deque` is empty, while `pollFirst` and `pollLast` return `null` in this case.

Stack

A stack is a very simple data structure that can only be accessed from the top. As an analogy, think of a stack of books:

- `push` adds an element to the top of the stack, equivalent to the `addFirst` method.
- `peek` retrieves, but does not remove an element from the top of the stack, just like the `peekFirst` method.
- `pop` retrieves and removes an element from the top of the stack, as though calling the `removeFirst` method.

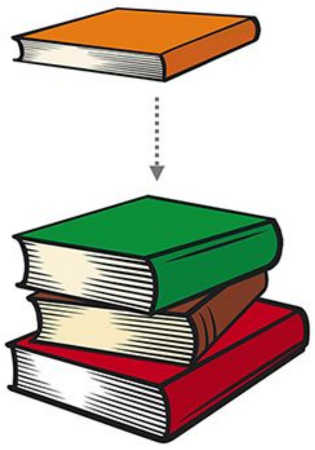


Figure 7

The Java Collections Framework does not provide a stack interface. There is a `Stack` class, but you should never use it. `Stack` extends `Vector`, which provides suboptimal performance, as mentioned in my Java Collections Framework Introduction.

All you need to implement a stack data structure is a way to add, retrieve, and retrieve and remove an element from one side of a list. To simplify this, the `Deque` interface directly supports the three stack methods, `push`, `peek` and `pop`. Alternatively, you could also use `addFirst`, `peekFirst` and `removeFirst` to implement a stack. However, you should prefer `push`, `peek` and `pop` instead, as they document your intentions clearly in code.

LinkedList Coding Samples

To round off this chapter, we will now look at some coding examples. Please note that you have to import classes like `java.util.Deque`, `java.util.concurrent.LinkedBlockingQueue` and `java.util.LinkedList` to compile the code samples.

add, addLast, and addFirst

As you can see in Example 3 below, we start by creating a new `LinkedList` instance and assign it to a new reference variable, called `queue`.

```
Queue<String> queue = new LinkedList<>();
queue.add("A");
queue.add("A");
queue.add("B");
System.out.println(queue);
```

```
Output:
[A, A, B]
```

Example 3

Note the use of the **diamond operator, <>**, introduced with Java 7. The operator spares us the trouble of writing “`String`” again before the parentheses. Using a reference variable of type `Queue` restricts us to the methods of `LinkedList` which are defined by the `Queue` interface. Other methods theoretically supported by the `LinkedList` instance won’t be accessible through the `Queue` reference variable.

Replacing `queue` with a `Deque` reference variable, `deque`, gives us the same outcome, as you can see in Example 4 below:

```
Deque<String> deque = new LinkedList<>();
deque.add("A");
deque.add("A");
deque.add("B");
System.out.println(deque);
```

```
Output:
[A, A, B]
```

Example 4

Since the `Deque` extends `Queue`, all `Queue` methods can also be used.

`LinkedList` also implements the `Collection` and `List` interfaces, whose methods aren't covered in this chapter. If you wish to know more about those interfaces and their methods, see the ArrayList chapter.

As we're using a `Deque` reference variable, we get to use `addLast` to add elements to the end of `deque`. Example 5 below demonstrates this:

```
Deque<String> deque = new LinkedList<>();
deque.addLast("1");
deque.addLast("2");
deque.addLast("3");
System.out.println(deque);
```

```
Output:
[1, 2, 3]
```

Example 5

Starting over with an empty `Deque`, we can use `addFirst` to add elements in reverse order to `addLast`, as Example 6 demonstrates:

```
Deque<String> deque = new LinkedList<>();
deque.addFirst("1");
deque.addFirst("2");
deque.addFirst("3");
System.out.println(deque);
```

```
Output:
[3, 2, 1]
```

Example 6

Example 7 shows that we can even use both `addFirst` and `addLast` in conjunction, but this can get really confusing so we should try to avoid doing so.

```
Deque<String> deque = new LinkedList<>();
deque.add("1");
deque.addLast("2");
deque.addFirst("3");
System.out.println(deque);
```

```
Output:
[3, 1, 2]
```

Example 7

offer, offerFirst and offerLast

Comparable to the three `add` methods, we have the three `offer` methods, which aren't much different from the `add` methods when it comes to `LinkedList` instances. `LinkedBlockingDeque`, however, is a `Deque` implementation that allows us to set a maximum number of elements in the deque. Trying to add another element to a full `LinkedBlockingDeque` is a situation that results in the special behavior explained earlier. The `offer`, `offerFirst` and `offerLast` methods will refuse to add the element and will return `false` in this case. Example 8 below illustrates this:

```
Deque<String> deque = new LinkedBlockingDeque<>(2);
deque.offer("1");
deque.offerFirst("2");
boolean wasAdded = deque.offerLast("3");
System.out.println(wasAdded);
System.out.println(deque);
```

```
Output:
false
[2, 1]
```

Example 8

The `add`, `addFirst` and `addLast` method in this case will also refuse to add the element, but will throw an `IllegalStateException` instead, as you can see in Example 9 below:

```java
Deque<String> deque = new LinkedBlockingDeque<>(2);
deque.offer("1");
deque.offerFirst("2");
deque.add("3");
```

```
Output:
java.lang.IllegalStateException: Deque full
```

Example 9

element

`element` allows to retrieve an element from the front of a `Deque` or `Queue` without removing it. Example 10 below illustrates this:

```java
Deque<String> deque = new LinkedList<>();
deque.add("1");
deque.add("2");
deque.add("3");
String element = deque.element();
System.out.println(element);
System.out.println(deque);
```

```
Output:
1
[1, 2, 3]
```

Example 10

getFirst and getLast

To retrieve an element from the front of a `Deque` without removing it, we can also use `getFirst`. It works exactly the same as the `element` method in our last example, as we can see in Example 11 below:

```
Deque<String> deque = new LinkedList<>();
deque.add("1");
deque.add("2");
deque.add("3");
String first = deque.getFirst();
System.out.println(first);
System.out.println(deque);
```

```
Output:
1
[1, 2, 3]
```

Example 11

Similarly, `getLast` allows us to retrieve an element from the back of a `Deque`, without removing it. See Example 12 below:

```
Deque<String> deque = new LinkedList<>();
deque.add("1");
deque.add("2");
deque.add("3");
String last = deque.getLast();
System.out.println(last);
System.out.println(deque);
```

```
Output:
3
[1, 2, 3]
```

Example 12

peekFirst and peekLast

Example 13 below shows how `peekFirst` and `peekLast` work identically to `getFirst` and `getLast`, except when used on an empty `Queue` or `Deque`.

```
Deque<String> deque = new LinkedList<>();
String first = deque.peekFirst();
System.out.println(first);
System.out.println(deque);
```

```
Output:
null
[]
```

Example 13

```
Deque<String> deque = new LinkedList<>();
String last = deque.peekLast();
System.out.println(last);
System.out.println(deque);
```

```
Output:
null
[]
```

Example 14

Calling the methods `element`, `getFirst` or `getLast` on an empty `Deque` causes a `NoSuchElementException` to be thrown. Examples 15-17 below demonstrate this:

```
Deque<String> deque = new LinkedList<>();
String element = deque.element();
```

```
Output:
java.util.NoSuchElementException
```

Example 15

```
Deque<String> deque = new LinkedList<>();
String element = deque.getFirst();
```

```
Output:
java.util.NoSuchElementException
```

Example 16

```
Deque<String> deque = new LinkedList<>();
String element = deque.getLast();
```

```
Output:
java.util.NoSuchElementException
```

Example 17

Removing from Deques and Queues

`remove` allows us to retrieve and remove an element from the front of a `Deque` or `Queue`. This can also be done using `removeFirst` on a `Deque`. Similarly, `removeLast` retrieves and removes an element from the back of a `Deque`. See Example 18 below:

```
Deque<String> deque = new LinkedList<>();
deque.add("1");
deque.add("2");
deque.add("3");

String first = deque.removeFirst();
System.out.println(deque);

String last = deque.removeLast();
System.out.println(deque);

String element = deque.remove();
System.out.println(deque);
```

```
Output:
[2, 3]
[2]
[]
```

Example 18

Calling `remove`, `removeFirst` or `removeLast` on an empty `Queue` or `Deque` will throw a `NoSuchElementException`. Example 19 below demonstrates this:

```
Deque<String> deque = new LinkedList<>();
deque.remove();
```

```
Output:
java.util.NoSuchElementException
```

Example 19

The `poll` methods work in a similar way to the `remove` methods. However, when you call `poll`, `pollFirst` or `pollLast` on an empty `Deque`, the methods will return `null` instead of throwing an exception. Example 20 below illustrates this.

```
Deque<String> deque = new LinkedList<>();
System.out.println(deque.poll());
System.out.println(deque.pollFirst());
System.out.println(deque.pollLast());
```

```
Output:
null
null
null
```

Example 20

Using a LinkedList as a Stack

Example 21 below shows us how we can use a `LinkedList` as a stack. To add an element to a stack, we call the method `push`. A stack is a so-called “last in, first out” data structure. When we add `“redbook”` and then `“brownbook”` to our stack, we can think of the brown book as being on top of the red one. In line 22, `peek` retrieves but does not remove `"brownbook"` as the top element. In lines 24-27, we `pop` elements off the stack one by one. In line 28, we call `peek` on a now empty stack, which returns `null`. Finally, line 29 demonstrates that calling `pop` on an empty stack throws a `NoSuchElementException`.

```
18  Deque<String> stack = new LinkedList<>();
19  stack.push("redBook");
20  stack.push("brownBook");
21  System.out.println("stack:" + stack);
22  System.out.println(stack.peek());
23  System.out.println("stack:" + stack);
24  System.out.println(stack.pop());
25  System.out.println("stack:" + stack);
26  System.out.println(stack.pop());
27  System.out.println("stack:" + stack);
28  System.out.println(stack.peek());
29  System.out.println(stack.pop());
```

```
Output:
stack:[brownBook, redBook]
brownBook
stack:[brownBook, redBook]
brownBook
stack:[redBook]
redBook
stack:[]
java.util.NoSuchElementException
```

Example 21

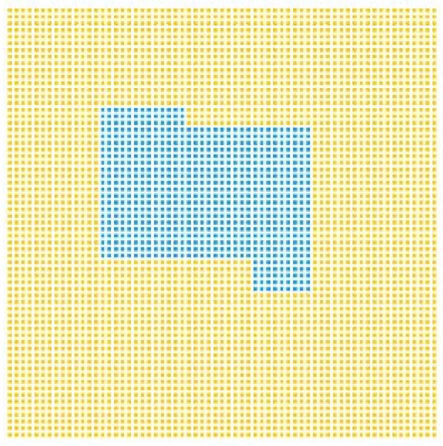
And that’s it: all you need to know about `java.util.LinkedList` and how it can be used as a `Queue`, `Deque` or a stack.

Chapter 27

Object Identity vs Equality

Object Identity

When we create objects in Java, the computer stores them in its memory. To be able to locate an object, the computer assigns it an address in the memory. Every new object we create gets a new address. If this yellow area represents an area of the computer's memory, the blue area represents our object being stored in the memory at some address.



Example 1

To illustrate this feature, let us imagine the building featured in Example 2 below. If we are looking at the building, we might be wondering if it is **the** White House or just another white house object. To check, we can compare this object's unique address to the White House's address. We would check our object's identity using '==', the equals operator. Hopefully the address of that house is "1600 Pennsylvania Avenue North West, Washington DC", otherwise we're looking at a different white house object, and the president isn't waiting inside to meet us.

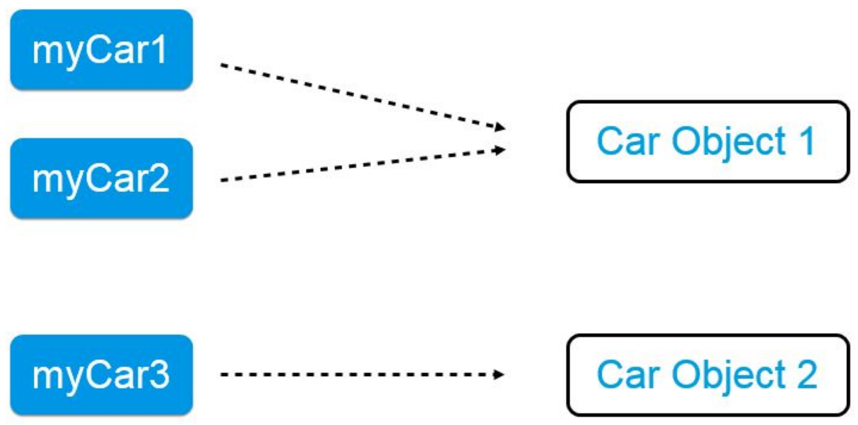
Example 2

Now, let's declare three variables and discuss their memory locations:

```java
Car myCar1 = new Car("blue");
Car myCar2 = myCar1;
Car myCar3 = new Car("blue");
```

Example 3

In Example 3 we have reference variables `myCar1`, `myCar2` and `myCar3`. `myCar1` was assigned a new Car object, as was `myCar3`, but `myCar2` was assigned the value of `myCar1`. This difference is key. `myCar2` is not a new object. It is simply a second reference variable 'pointing' to the same object in the memory. So while we have three variables that we created, we actually have only placed two objects in the memory (Example 4).



Example 4

Now, let’s take these reference variables and compare them using the equals operator, ‘==’. When we use the equals operator, we can see if both variables refer to the same object in the memory. Take a look at the three ‘if’ statements below:

```
if(myCar1 == myCar1) { /* true */
}
if(myCar1 == myCar2) { /* true */
}
if(myCar1 == myCar3) { /* false */
}
```
Example 5

When we compare `myCar1` to itself, it evaluates to `true`, because they are referring to the same object in the memory. Similarly, `myCar1 == myCar2` evaluates to `true` as well. Again, although they are different reference variables, they are referencing the same object in the memory. Finally, `myCar1 == myCar3` evaluates to `false`, because they are pointing to different objects in the memory.

Object Equality

Another way that one can test equality is by using the `equals()` method. The equals method tells us if two objects are considered equal. Let us suppose that our program requires that two cars are 'equal' if they are of the same color. So let's look at the same three 'if' statements:

```
if(myCar1.equals(myCar1)) { /* true */
}
if(myCar1.equals(myCar2)) { /* true */
}
if(myCar1.equals(myCar3) { /* false */
}
```
Example 6

Based on what you've read so far, you'd think that all three statements would evaluate to `true`. However, that is not how the default `equals()` method works. If you look at the default `equals()` method of the Object class, it actually calls '`==`', giving it the same functionality as simply saying `obj1 == obj2`.

```
public boolean equals(Object obj) {
    return (this == obj);
}
```
Example 7

Obviously, this isn't what we want. In our example, we want to judge if two Cars are equal based on their color. So, we will have to override the `equals()` method:

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    Car other = (Car) obj;
        return this.color.equals(other.color);
}

@Override
public int hashCode() {
    return color.hashCode();
}
```
Example 8

Now, we are expressing in code what we consider equal or unequal. Again, this totally depends on what our client considers equal or unequal. We have to override these methods not because the creators of Java thought that it would be a good idea, but because there wasn't any other option. When they wrote the object class, they didn't really have in mind our car class and the specific way in which we would compare them, so they came up with a generic method that they welcome us to change. You might also notice that I didn't just overwrite the `equals()` method. I also overrode the `hashCode()` method. Java specifies that **equal objects must have equal hashCodes as well**. For more detail on why we have to override both methods, check out my blog post about equals and hashcode.

Chapter 28

The Java Comparable Interface

How should we compare and sort things? Now that might seem like a weird question, but I want you to really think about it. Let's say we have a set of apples:

Example 1

How do we want to sort them? Do we want to sort them by weight? If so, are we sorting them from lightest to heaviest or heaviest to lightest? When we are sorting them, we need to repeatedly compare two apple's weights until all the apples are in the correct order. Is apple 1 heavier than apple 2? Is it heavier than apple 3? We need to keep doing that until the list is sorted. The comparable interface helps us accomplish this goal. Comparable can't sort the objects on its own, but the interface defines a method `int compareTo(T)`.

How compareTo() Works

Let's begin by utilizing the `compareTo()` method to see which apples are heavier.

Example 2

The `compareTo()` method works by returning an int value that is either positive, negative, or zero. It compares the object by making the call to the object that is the argument. A negative number means that the object making the call is “less” than the argument.If we were comparing the apples by size, the above call would return a negative number, say `-400`, because the red apple is smaller than the green apple. If the two apples were of equal weight, the call would return 0. If the red apple was heavier, `compareTo()` would return a positive number, say `68`.

The Flexibility of compareTo()

If we called the `compareTo()` method above repeatedly, we could sort our apples by size, which is great, but that’s not the end of the story. What if we want to sort apples by color? Or weight? We could do that too. The key is that our client, let’s call him Fatty Farmer, (see Example 3), needs to precisely define how the apples need to be sorted before we can start development.

Example 3

He can do this by answering these two questions:

1. How does he want the apples to be sorted? What is the characteristic he would like us to compare?
2. What does ‘less than’, ‘equal to’, and ‘greater than’ mean in that context?

It’s also possible to use multiple characteristics, as we’ll see a little bit later.

Example 1: Sorting Apples by Weight

For our first example, we’re going to sort our apples by weight. It only requires one line of code.

```java
Collections.sort(apples);
```

Example 4

The above line of code can do all the sorting for us, as long as we’ve defined how to sort the apples in advance (That’s where we’ll need more than one line).

Let’s begin by writing the `Apple` class.

```java
public class Apple implements Comparable {
    private String variety;
    private Color color;
    private int weight;

    @Override
    public int compareTo(Apple other) {
        if (this.weight < other.weight) {
            return -1;
        }
        if (this.weight == other.weight) {
            return 0;
        }
        return 1;
    }
}
```

Example 5

This is our first version of class `Apple`. Since we are using the `compareTo` method and sorting the apples, I implemented the `Comparable` interface. In this first version, we're comparing objects by their weight. In our `compareTo()` method we write an if condition that says if the apple's weight is less than the other apple, return a negative number, to keep it simple, we'll say `-1`. Remember, this means that this apple is lighter than apple `other`. In our second if statement, we say that if the apples are of equal weight, return a `0`. Now if `this` apple isn't lighter, and it isn't the same weight, then it must be greater than the other apple. In this case we return a positive number, say, `1`.

Example 2: Sorting Apples by Multiple Characteristics

As I mentioned before, we can also utilize `compareTo()` to compare multiple characteristics. Let's say we want to first sort apples by variety, but if two apples are of the same variety, we should sort them by color. Finally, if both of these characteristics are the same, we will sort by weight.

While we could do this by hand, in full, like I did in the last example, we can actually do this in a much cleaner fashion. Generally, it is better to reuse existing code than to write our own. We can use the `compareTo` methods in the `Integer`, `String`, and `enum` classes to compare our values. Since we aren't using Integer objects, rather we are using ints we have to use a static helper method from the `Integer` wrapper class to compare the two values.

```java
public class Apple implements Comparable<Apple> {
    private String variety;
    private Color color;
    private int weight;

    @Override
    public int compareTo(Apple other) {
        int result = this.variety.compareTo(other
                                                .variety);
        if (result != 0) {
            return result;
        }
        if (result == 0) {
            result = this.color.compareTo(other.color);
        }
        if (result != 0) {
            return result;
        }
        if (result == 0) {
            result = Integer.compare(this.weight, other
                                                 .weight);
        }
        return result;
    }
}
```
Example 6

In Example 6, we compare the first quality of the apples that our client prioritized, their variety. If the result of that `compareTo()` call is non-zero, we return the value. Otherwise we make another call until we get a non-zero value, or we’ve compared all three characteristics. While this code works, it isn’t the most efficient or clean solution. In Example 3, we refactor our code to make it even simpler.

```java
@Override
public int compareTo(Apple other) {
    int result = this.variety.compareTo(other.variety);
    if (result == 0) {
        result = this.color.compareTo(other.color);
    }
    if (result == 0) {
        result = Integer.compare(this.weight, other
                                              .weight);
    }
    return result;
}
```

Example 7

As you can see, this greatly shortens our code and allows us to make each comparison in only one line. If the result of a `compareTo()` call is zero, we just move on to the next “round” of comparisons within the same if statement. This, by the way, is a good example of what you do as a Clean Coder. Usually, you don’t instantly write Clean Code; you start with a rough idea, make it work, and then continuously improve it until you’ve made it as clean as you can.

Comparable, hashCode, and equals

You may notice that the `compareTo()` looks a little bit like the `hashCode()` and `equals()` methods. There is one important difference, however. For `hashCode()` and `equals()`, the order in which you compare individual attributes does not influence the value returned, however in `compareTo()` the order of the objects is defined by the order in which you compare the objects.

Conclusion

To conclude I just want to underscore how important the `Comparable` interface is. It is used in both the `java.util.Arrays` and the `java.util.Collections` utility classes to sort elements and search for elements within sorted collections. With collections like `TreeSet` and `TreeMap`, it’s even easier - they automatically sort their elements which have to implement the `Comparable` interface.

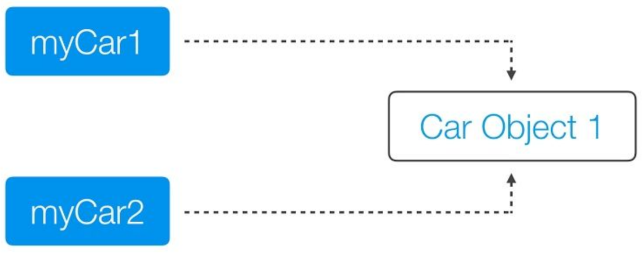
Part 4

Advanced Concepts

Chapter 29

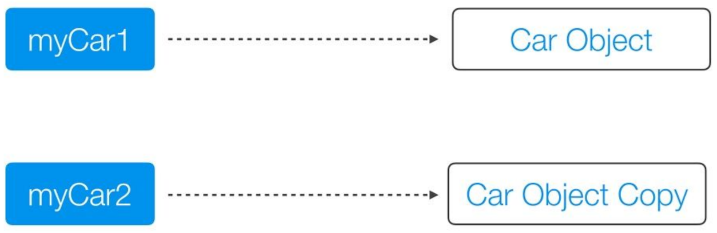
Shallow vs Deep Copy

To begin, I'd like to highlight what a copy in Java is. First, let's differentiate between a reference copy and an object copy. A **reference copy**, as the name implies, creates a copy of a reference variable pointing to an object. If we have a Car object, with a `myCar` variable pointing to it and we make a reference copy, we will now have two `myCar` variables, but still one object.



Example 1

An **object copy** creates a copy of the object itself. So if we again copied our `car` object, we would create a copy of the object itself, as well as a second reference variable referencing that copied object.



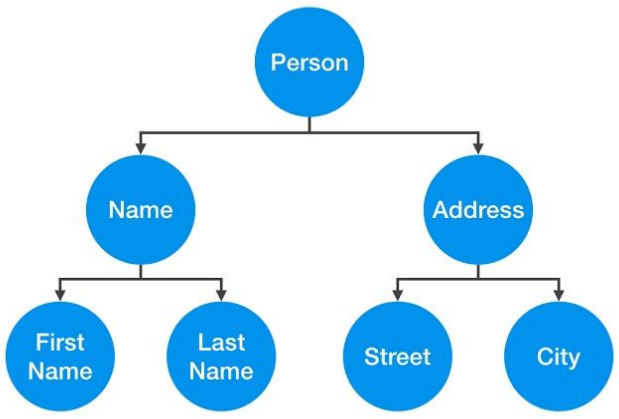
Example 2

What is an Object?

Both a Deep Copy and a Shallow Copy are types of object copies, but what really is an object? Often, when we talk about an object, we speak of it as a single unit that can't be broken down further, like a humble coffee bean. However, that's oversimplified.

Example 3

Say we have a `Person` object. Our `Person` object is in fact composed of other objects, as you can see in Example 4. Our `Person` contains a `Name` object and an `Address` object. The `Name` in turn, contains a `FirstName` and a `LastName` object; the `Address` object is composed of a `Street` object and a `City` object. So when I talk about `Person` in this chapter, I'm actually talking about this *entire network of objects*.



Example 4

So why would we want to copy this `Person` object? An object copy, usually called a clone, is created if we want to modify or move an object, while still preserving the original object. There are many different ways to copy an object that you can learn about in another chapter. In this chapter we'll specifically be using a copy constructor to create our copies.

Shallow Copy

First let's talk about the shallow copy. A shallow copy of an object copies the 'main' object, but doesn't copy the inner objects. The 'inner objects' are shared between the original object and its copy. For example, in our `Person` object, we would create a second `Person`, but both objects would share the same `Name` and `Address` objects.

Let’s look at a coding example. In Example 5, we have our class `Person`, which contains a `Name` and `Address` object. The copy constructor takes the `originalPerson` object and copies its reference variables.

```java
public class Person {
    private Name name;
    private Address address;

    public Person(Person originalPerson) {
        this.name = originalPerson.name;
        this.address = originalPerson.address;
    }
    [...]
}
```

Example 5

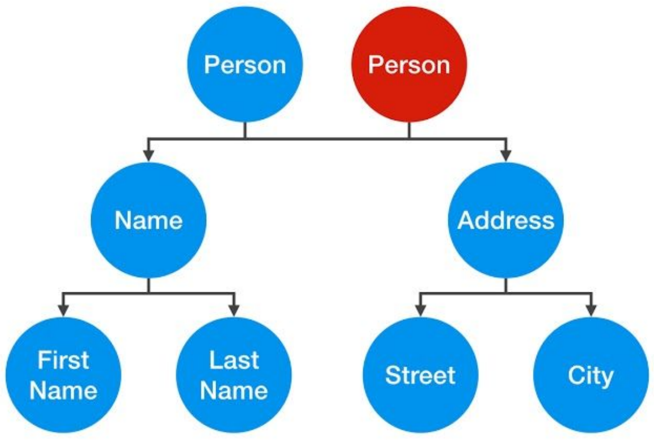
The problem with the shallow copy is that the two objects are not independent. If you modify the `Name` object of one `Person`, the change will be reflected in the other `Person` object.

Let’s apply this to an example. Say we have a `Person` object with a reference variable `mother`; then, we make a copy of `mother`, creating a second `Person` object, `son`. If later on in the code, the `son` tries to `moveOut()` by modifying his `Address` object, the `mother` moves with him!

```java
Person mother = new Person(new Name(...), new
                                         Address(...));
[...]
Person son = new Person(mother);
[...]
son.moveOut(new Street(...), new City(...));
```

Example 6

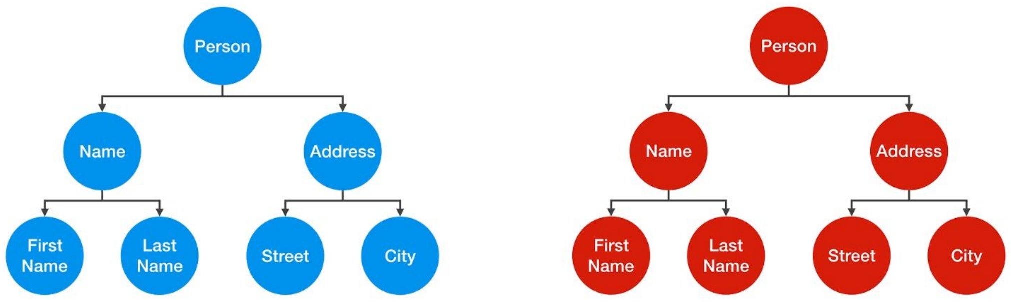
This occurs because our `mother` and `son` objects share the same `Address` object, as you can see illustrated in Example 7. When we change the `Address` in one object, it changes in both!



Example 7

Deep Copy

Unlike the shallow copy, a deep copy is a **fully independent copy of an object**. If we copied our `Person` object, we would copy the entire object structure.



Example 8

A change in the `Address` object of one `Person` wouldn't be reflected in the other object as you can see by the diagram in Example 8. If we take a look at the code in example 9, you can see that we're not only using a copy constructor on our `Person` object, but we are also utilizing copy constructors on the inner objects as well.

```java
public class Person {
    private Name name;
    private Address address;

    public Person(Person otherPerson) {
        this.name = new Name(otherPerson.name);
        this.address = new Address(otherPerson
                                           .address);
    }
    [...]
}
```

Example 9

Using this deep copy, we can retry the mother-son example from Example 6. Now the `son` is able to successfully move out!

However, that's not the end of the story. To create a true deep copy, we need to keep copying all of the `Person` object's nested elements, until there are only primitive types and "Immutables" left. Let's look at the Street class to better illustrate this:

```java
public class Street {
    private String name;
    private int number;

    public Street(Street otherStreet) {
        this.name = otherStreet.name;
        this.number = otherStreet.number;
    }
    [...]
}
```

Example 10

The `Street` object is composed of two instance variables - `String name` and `int number`. `int number` is a primitive value and not an object. It's just a simple value that can't be shared, so by creating a second instance variable, we are automatically creating an independent copy. `String` is an Immutable. In short, an Immutable is an Object, that, once created, can never be changed again. Therefore, you can share it without having to create a deep copy of it.

Conclusion

To conclude, I'd like to talk about some coding techniques we used in our mother-son example. Just because a deep copy will let you change the internal details of an object, such as the `Address` object, it doesn't mean that you should. Doing so **would decrease code quality**, as it would make the `Person` class more fragile to changes - whenever the `Address` class is changed, you will have to (potentially) apply changes to the `Person` class also. For example, if the `Address` class no longer contains a `Street` object, we'd have to change the `moveOut()` method in the `Person` class on top of the changes we already made to the `Address` class.

In Example 6 of this chapter I only chose to use a new `Street` and `City` object to better illustrate the difference between a shallow and a deep copy. Instead, I would recommend that you assign a new `Address` object instead, effectively converting to a ***hybrid*** of a shallow and a deep copy, as you can see in Example 10:

```
Person mother = new Person(new Name(...), new
                                          Address(...));
[...]
Person son = new Person(mother);
[...]
son.moveOut(new Address(...));
```

Example 10

In object-oriented terms, this violates **encapsulation**, and therefore should be avoided. Encapsulation is **one of the most important aspects of Object Oriented programming**. In this case, I had violated encapsulation by accessing the internal details of the `Address` object in our `Person` class. This harms our code because we have now entangled the `Person` class in the `Address` class and if we make changes to the `Address` class down the line, it could harm the `Person` class as I explained above. While you obviously need to interconnect your various classes to have a coding project, whenever you connect two classes, you need to analyze the costs and benefits.

Chapter 30

Immutable Classes

In this chapter, I will be discussing immutable classes in Java. The concept of immutability has always been important in all programming languages, including Java. With the release of Java 8 however, immutable classes have become even more important. This version introduced a new java.time API, which is based entirely on immutable classes.

What is an Immutable Class?

An immutable class is a class whose instances cannot be modified. Information stored in an immutable object is provided when the object is created, and after that it is unchangeable and read-only forever. As we can't modify immutable objects, we need to work around this. For instance, if we had a spaceship class, and we wanted it to fly somewhere, we'd have to return a new object with modified information.

```java
public Spaceship flyTo(Destination destination) {
     return new Spaceship(name, destination);
}
```
Example 1

Advantages of Immutable Classes

At first glance, you'd think that immutable classes aren't very useful, but they do have many advantages.

1. Immutable objects are stable

Firstly, immutable classes greatly reduce the effort needed to implement a stable system. The property of immutable objects that prevents them from being changed is extremely beneficial when creating this kind of system. For example, imagine that we are making a `BankAccount` class for a large bank. After the financial crisis, the bank won't allow its users to have a negative `BankAccount` balance.

Figure 1

So, they institute a new rule and add a validation method to throw an `IllegalArgumentException` whenever a function call results in a negative balance. This type of rule is called an **invariant**.

```
public class BankAccount{
     [...]
     private void validate(long balance) {
          if (balance < 0) {
              throw new IllegalArgumentException(
             "balance must not be negative:"+ balance);
          }
     }
}

```
Example 2

In a typical class, this `validate()` method would be called anytime a user's balance is changed. If the user makes a withdrawal or transfers money out of his account, we would have to call the validate method. However, with an immutable class, we only have to call the validate method once, in the class constructor, as in line 6 in Example 3 below.

```
5  public BankAccount(long balance) {
6       validate(balance);
7       this.balance = balance;
8  }
```
Example 3

For every distinct state, a new object will be created. Whenever a new object is created, its constructor will call the validate method again. This is extremely useful. Once the invariants have been established in the constructor, they will remain true for the entire lifetime of the object.

2. Immutable objects are fault tolerant

Similarly, immutable classes can be used to support a fault tolerant system. Imagine that someone tries to withdraw money from a bank, but between the time that their money is withdrawn from the account and the money is released from the ATM, there is a technical error. In a mutable class, it would be difficult to ensure that the person's money isn't lost forever. In an immutable class, though, you could for instance throw an `IllegalArgumentException`, preventing the account from losing money before the user physically receives it.

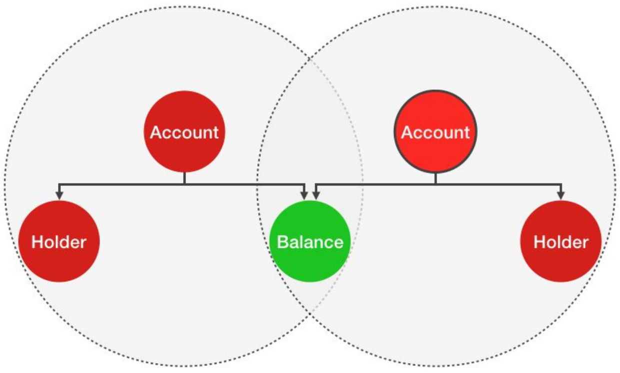
```java
public ImmutableAccount withdraw(long amount) {
     long newBalance = newBalance(amount);
     return new ImmutableAccount(newBalance);
}
private long newBalance(long amount) {
     /* exception during balance calculation */
}
```

Example 4

An immutable object can never get into an inconsistent state, even in the case of an exception. This removes the threat of an unforeseen error destabilizing the entire system. Apart from the cost of the initial validation, this stability comes at no cost.

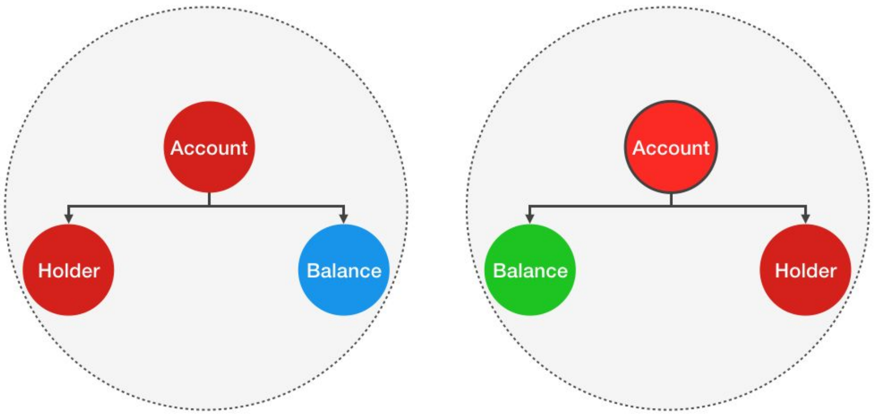
3. Immutable objects can be shared freely

In Example 5 below, you can see two distinct `Account` objects sharing the same `Balance` instance:



Example 5

If `Balance` is an immutable object, then we can change the balance in one of the accounts and the "change" will return a different instance instead. Example 6 below demonstrates this:



Example 6

Therefore, immutable objects can be shared freely. Immutable objects can even be shared freely when using a lock free algorithm in a multithreaded environment, where multiple actions happen in parallel. As a consequence, an immutable class should never provide a copy constructor. More specifically, you should never create any kind of copy of an immutable object.

4. Immutable objects work well as Map keys and Set elements

Finally, immutable objects are also a perfect option to use as keys of a `java.util.Map` or elements of a `java.util.Set`. The reason for this is that when you alter `Map` keys or `Set` elements, a lookup could result in an empty or wrong result returned later on.

Disadvantages of Immutable Classes

The only disadvantage that immutable classes have is a *potential* to cause performance problems. For every new state of an object, you need to return a new different object. But is that really true? Imagine that we want to create an immutable class representing a traffic light. Each time the color of the traffic light changes, we have to return a different object. However, instead of returning a new object each time, we could also store (or ‘cache’) one instance of the traffic light in each color: red, yellow and green (Example 7 below) - and whenever we wanted to change the color of the immutable `TrafficLight` object, we could just switch among the three different objects.

Example 7

This shows that we cannot automatically infer that using immutable objects will lead to poor performance. In fact, the opposite is often the case, especially for multithreaded systems.

Okay, for now, let’s leave it at that. At the end of the chapter, we’ll consider cases where this disadvantage really becomes an issue, and where it is safe to ignore.

How to Create an Immutable Class

Now that I've shown you why immutable classes are valuable and when you should use them, I'm going to show you how to make one. Picking up the `Spaceship` idea from Example 1, let's go through the steps of designing an immutable class. To create an immutable class, you have to follow these four steps:

1. Make all fields private and final.
2. Don't provide any methods that modify the object's state.
3. Ensure that the class can't be extended.
4. Ensure exclusive access to any mutable fields.

1. Make all attributes private and final

The first condition of an immutable class is that all its attributes must be `private` and `final`. `private`, so that the attributes cannot be directly accessed from outside the class. We make the attribute `final`, so that they can also never be reassigned from within the class. Also, this clearly communicates our intent in code to the entire development team: "These fields must not be changed".

Keeping things simple for now, we will start with a class that consists only of immutable attributes. As you can see in Example 8 below, these attributes are `name` and `destination`. The `java.lang.String` class is immutable. `Destination` however is a "custom" class of our application, and for now we will assume that it is immutable as well.

```java
public class Spaceship {
    private final String name;
    private final Destination destination;

    public ImmutableSpaceship(String name) {
        this.name = name;
        this.destination = Destination.NONE;
    }

    public Spaceship(String name, Destination
                                        destination) {
        this.name = name;
        this.destination = destination;
    }
}

```

Example 8

Note:
Theoretically, you could even make immutable attributes and primitive values public final. However, this would violate the rules of good object oriented design. An object should provide a service / behavior through public methods and hide its internal representation, so that it can be changed at a later time without requiring changes in other classes. This is known as “encapsulation”.

2. Don’t provide any methods that modify the object’s state

The next step should be already somewhat familiar to you, as we already looked at it at the beginning of the chapter. Whenever you have a method that would modify an object’s state, you instead have to return a different object, as lines 22-24 in Example 9 below demonstrate.

```java
public class ImmutableSpaceship {

    private final String name;
    private final Destination destination;

    public ImmutableSpaceship(String name) {
        this.name = name;
        this.destination = Destination.NONE;
    }

    private ImmutableSpaceship(String name, Destination
                                             destination) {
        this.name = name;
        this.destination = destination;
    }

    public Destination currentDestination() {
        return destination;
    }

    public ImmutableSpaceship flyTo(Destination
                                          newDestination) {
        return new ImmutableSpaceship(this.name,
                                           newDestination);
    }

    […]
}

```
Example 9

In Example 10 below, we now create a direct instance of the class.

```java
ImmutableSpaceship immutableSpaceship = new
                                     ImmutableSpaceship();
```
Example 10

This instantiates an immutable *object*. Nevertheless, the *class* is not fully immutable yet. Should someone extend the class, it would still be possible to violate the rules of immutability in the `ImmutableSpaceship` subclass. We say “the *`ImmutableSpaceship` class is effectively immutable*”. As long as no one extends the class, it will be immutable.

3. Ensure that the class can’t be extended

To make our class truly immutable, we also need to protect it from being extended. If our class could be extended, then someone could override its methods, which could modify our object. If you ever get this wrong, you are in good company. Josh Bloch wrote the two classes [BigInteger] and [BigDecimal]. Both were supposed to be immutable, but both can be extended.

To illustrate why this is a problem, let’s look at the `RomulanSpaceship` in Example 11 below.

```
public class RomulanSpaceship extends Spaceship {

    private Destination destination;

    @Override
    public ImmutableSpaceship flyTo(Destination
                                             destination) {
        this.destination = destination;
        return this;
    }
}

```
Example 11

This class maintains its own, non-final `destination` field. The original field of its superclass is `private` and therefore not visible within `RomulanSpaceship`. Fields can generally not be overwritten. It is just a field with the same type and name as that in the superclass. The name of the field is, in principle, arbitrary. However, using the same name makes it even more confusing and difficult for the next developer to detect the bug. The important part is that the overriding `flyTo` method of `RomulanSpaceship` does not return a new `RomulanSpaceship` instance. Instead, it directly changes the internal state of the current object and returns a reference to itself to the caller. Keep in mind that you may not always be able to see the source code, as you can here. Even if you can see it, you may not have the time to look at it in much detail. Now assume that you are using an instance of a `RomulanSpaceship` that is assigned to a reference variable of `ImmutableSpaceship`. A `RomulanSpaceship` *is an* `ImmutableSpaceship`. Any method that requires an `ImmutableSpaceship` instance can also be called with a `RomulanSpaceship` instance. This could result in very hard-to-find bugs in code that relies upon an immutable spaceship instance.

To fix this issue, we must ensure that the class can't be extended. The simplest and most secure way of doing this is to make the class `final`, as you can see in Example 12 below.

```java
public final class Spaceship
```
Example 12

Note:
There are also other alternatives. For instance, you could also make all constructors private and provide a static factory method to create an instance of the class. This is a bit more dangerous, though, as it documents your intention less clearly. Another developer could add another public constructor to the class later on, and in this way violate the immutability of the class.

Now the source code of `ImmutableSpaceship` looks like this:

```java
public final class ImmutableSpaceship {

    private final String name;
    private final Destination destination;

    public ImmutableSpaceship(String name) {
        this.name = name;
        this.destination = Destination.NONE;
    }

    private ImmutableSpaceship(String name, Destination
                                              destination) {
        this.name = name;
        this.destination = destination;
    }

    public Destination currentDestination() {
        return destination;
    }

    public ImmutableSpaceship flyTo(Destination
                                        newDestination) {
        return new ImmutableSpaceship(this.name,
                                        newDestination);
    }

    […]
}

```

Example 13

On the assumption that `destination` is immutable, this class is immutable now, and we are done.

Please remember that designing your class as immutable *today* is no guarantee that it will also be immutable *tomorrow*. Another developer who is unaware of your intentions may later make adjustments to the class that will break its immutability. This is one of a few good reasons to add a comment to your class, like this one in Example 14:

```
/*
 * This class is immutable.
 */
public final class Spaceship
```

Example 14

When I spoke about the advantages of immutable classes earlier, I explained that immutable objects can be shared freely without having to create a deep copy. For this reason, it is particularly easy to build an immutable class based on immutable attributes. Besides the immutable `String` attribute, our class contains a `Destination` attribute. Until now, we have defined the "custom" class `Destination` as immutable also. From now on, we'll assume that the `Destination` attribute is mutable. In this case, we'll have to follow a fourth rule, as we'll see in the section that follows.

4. Ensure exclusive access to mutable fields

Anyone who holds access to a mutable field can alter it, thereby mutating the otherwise immutable object. To prevent someone from gaining access, we should never obtain or return a direct reference to a mutable object. Instead, we must create a deep copy of our mutable object and work with that instead. Let's see what this means by going back to our `ImmutableSpaceship`, as shown in Example 15 below:

```
 3 public final class ImmutableSpaceship {
 4
 5     private final String name;
 6     private final Destination destination;
 7
 8     public ImmutableSpaceship(String name) {
 9         this.name = name;
10         this.destination = new Destination("NONE");
11     }
12
13     private ImmutableSpaceship(String name, Destination
                                             destination) {
14         this.name = name;
15         this.destination = destination;
16     }
17
18     public Destination currentDestination() {
19         return destination;
20     }
21
22     public ImmutableSpaceship flyTo(Destination
                                          newDestination) {
23         return new ImmutableSpaceship(this.name,
                                           newDestination);
24     }
25
       [...]
51 }
52
```
Example 15

The source code is almost identical to the source code of Example 13. In fact, the only difference is line 10, where we explicitly call a `Destination` constructor. This is to clearly indicate that we are now assuming that `Destination` is a mutable class. Besides, a mutable public static attribute would be a serious design flaw.

As a consequence, we have to ensure that no one can obtain a direct reference to an internal `Destination` attribute. To achieve this, we start by checking all `public` methods and constructors for any incoming or outgoing `Destination` references.

The public constructor (lines 8-11) does not receive any `Destination` reference. The `Destination` object it creates is safe, as it cannot be accessed from outside. So the `public` constructor is good as it is.

In the `currentDestination()` method (lines 18-20), however, we directly return the `Destination` reference. On the assumption that `Destination` is mutable, this is a problem. To fix it, we return a deep copy of the internal `Destination` object, as you can see in Example 16 below:

```
18  public Destination currentDestination() {
19        return new Destination(destination);
20  }
```

Example 16

The last `public` method we have to check is `flyTo` (line 21-23) in Example 17 below:

```
12      private ImmutableSpaceship(String name, Destination
                                             destination){
13          this.name = name;
14          this.destination = destination;
15      }
16
        [...]
21      public ImmutableSpaceship flyTo(Destination other) {
22          return new ImmutableSpaceship(this.name, other);
23      }
        [...]
```

Example 17

It receives a `Destination` reference and directly forwards it to a `private` constructor (lines 12-14). The `flyTo` method will return a new object. Unfortunately, this new object will share the same reference to the destination object given as a parameter to the `flyTo` method - a mutable reference just leaked in! You might be tempted to fix the problem by creating a deep copy of the destination parameter right in the `flyTo` method. However, this would not truly fix the problem! Any other `public` method, now or in future, would suffer the same consequences. As Clean Coders, we must not allow this! The true source of the problem is the `private` constructor that allows to create a new instance with a given reference. To solve the problem, we have to create a deep copy of the incoming destination reference right there, in the constructor, as you can see in Example 18 below:

```
12  private ImmutableSpaceship(String name, Destination
                                              destination) {
13        this.name = name;
14        this.destination = new Destination(destination);
15  }
```
Example 18

Now the final source code of `ImmutableSpaceship` looks like this:

```java
/*
 * This class is immutable and thread-safe.
 */
public final class ImmutableSpaceship {

    private final String name;
    private final Destination destination;

    public ImmutableSpaceship(String name) {
        this.name = name;
        this.destination = new Destination("NONE");
     }

    private ImmutableSpaceship(String name, Destination
                                             destination) {
        this.name = name;
       this.destination = new Destination(destination);
    }

    public Destination currentDestination() {
        return new Destination(destination);
    }

    public ImmutableSpaceship flyTo(Destination
                                          newDestination) {
        return new ImmutableSpaceship(this.name,
                                           newDestination);
    }

    [...]
}
```

Example 19

Assuming that `Destination` is immutable in Example 13 and mutable in Example 19, keep in mind that *both* examples are valid immutable classes.

When to use Immutable Classes

Okay, now that we know *how* to create an immutable class, I want to discuss **when** we should use them. In his book Effective Java, Josh Bloch gives us a clue when he says:

"Classes should be immutable unless there's a very good reason to make them mutable".

This is fully correct, but how do we know when we have a good reason to make a class mutable? It's a hotly debated topic, and there is no golden rule. Remember that an immutable class requires a separate object for each distinct state. Therefore, using immutable objects tends to increase the number of objects created. This may become taxing if the objects used are too complex. In other words, small classes that contain few attributes are better suited as immutable classes than complex classes that contain a large number of attributes. We cannot automatically deduce from the number of attributes a class has whether it should be immutable or not, though. That'd be premature. There are many other deciding factors.

For example, it depends on the specific problem at hand (the problem domain), its software design, the hardware it is running on, as well as the scenario in which it is being used. A system may be used by only one person or by a thousand people in parallel. The program could be running on your private laptop or on a high performance server cluster. There will be parameters that you can and cannot influence. You can't influence your problem or the environment, but you can influence the way you go about designing a solution to the problem. The best you can do, is follow this guideline:

Utilize immutable classes as much as possible. Start by making every class immutable and facilitate their immutability by creating small classes with few attributes and methods. Simplicity and clean code are essential.

If you have clean code, that facilitates immutability; and if you have immutable classes, your code is cleaner. In the majority of cases, this approach will lead to a system that exceeds all requirements.

If testing reveals that you haven’t achieved satisfactory performance, relax the immutability rules gradually, as a last resort. As much as necessary, but as little as possible.

I hope you now have a better understanding of immutable classes, and can see just how useful they are. Remember to keep your code simple and clean, by using immutable classes to the greatest possible extent.