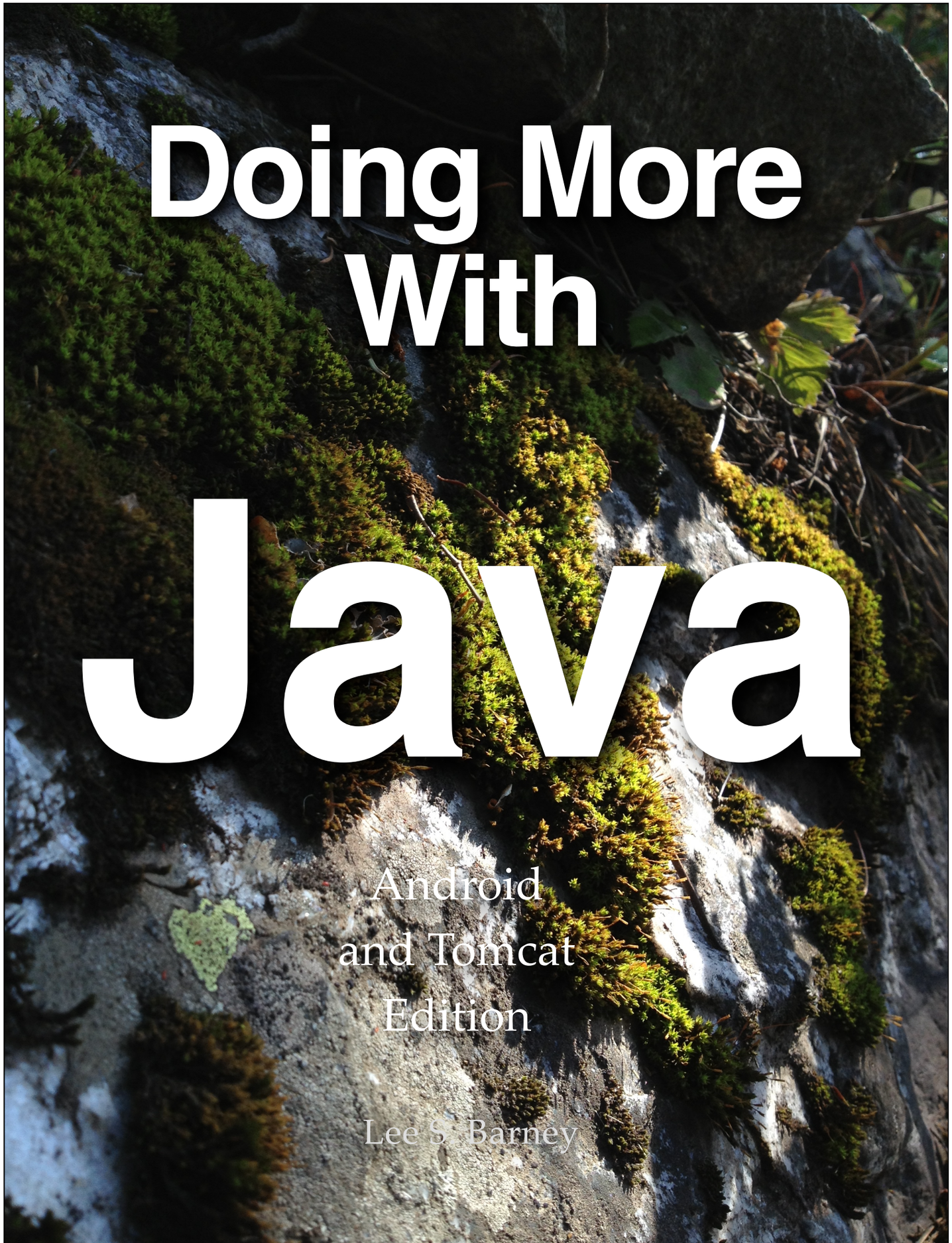

Doing More
With
Java
Android
and Tomcat
Edition
Lee S. Barney

Dedication

This book is dedicated to my wife and children. They make everything worth while. Also to those students who learn because learning is fun.

Other

Information has been obtained by Author from sources believed to be reliable. However, because of the possibility of human or mechanical error by these sources, Author, the editor, or others, Author and the editor do not guarantee the accuracy, adequacy, or completeness of any information included in this work and are not responsible for any errors or omissions or the results obtained from the use of such information.

Java™ is a registered trade mark of Oracle Corp. Android™ and Android Studio™ are registered trade marks of Google Inc. Apache Tomcat™ is a trademark of The Apache Software Foundation. IntelliJ™ is a trademark of JetBrains s.r.o. Genymotion™ is a trademark of Genymobile SAS.

Question and Arrow Left icons used were created by snap2objects and used under license.

Pig image used was created by Martin Berube and used under license.

Sausage image created by Daily Overview, http://www.dailyoverview.com, and used with permission.

Green check icon was created by VisualPharm and used under license.

READ ME. NO, REALLY!

The book is not designed as an exhaustive API or software engineering book, a book of code script-lets to copy, nor a book full of detailed definitions of every possible word since these types of information are readily available on-line. Use the dictionary in your electronic book reader, Google®, and maybe even WikiPedia for definitions that are not explicitly given in the text for unfamiliar words.

The first chapter consists of a series if things you will need to know BEFORE you go on. Other chapters follow a consistent pattern.

Chapter Content Description

Each chapter consists of:

1. A brief description of the material to be covered and why knowledge of this information is important to you.
2. A series of points for consideration as you read and absorb the detailed information. These points are excellent items to discuss with co-workers or team members. Such discussions between peers yields greater depth of understanding and speeds learning.
3. A detailed and well organized description of the material to be learned. This material may have full color images and short videos interspersed within it. Each of these images and videos has been carefully created to clarify difficult or complex concepts.

CHAPTER 1

Stuff You Need to Know

The sections of this chapter present information that you, as a reader, need to understand in order to be successful in learning the remainder of the information presented. Read them and use them as the road to more learning. If you need a more basic introduction to Java consider the Doing Stuff with Java, the initial book in this series.

Required Tools

Get and Install These

1. A laptop or desktop computer
2. The latest version of the Java Developer Kit™ for your computer
3. The latest version of Android Studio™
4. The latest version of IntelliJ™ or some other IDE
5. The latest version of Genymotion™
6. The latest version of the Apache Tomcat™ server

Java and other programming tools can come from many sources

Many tools used to create Java applications are open source and free. You can edit and compile your applications by hand using a command line terminal and a text editor or you can use an Integrated Development Environment (IDE) such as Intellij™.

Regardless of what environment you chose to work in, you will need to download and install the Java Development Kit™ (JDK). You can get it directly from Oracle™, Make sure you get the latest version.

To share with your teammates and to allow potential employers see what you know and can

do, I suggest using git and gitHub (http://www.github.com) as a content management system. If you don't already have a gitHub account go create one.

Learn git. It is a skill that employers of programmers currently want to see in potential employees. Oh... and by the way.... most of them want to see that you know how to use it from the command line. Don't use a GUI. Pro Git is a good resource book and is available for free in PDF, mobi, and ePub formats from the git website.

Example source code for this book can be retrieved from the Doing More with Java gitHub repository either using git or by downloading the zip file.

The editor I used to create the examples is IntelliJ. You can choose to use any IDE or text editor you like.

If it is impossible for you to install a compiler on you own machine, you can use an online compiler like Coding Ground. This is not as good an option as having a compiler on your own machine but it may work for you.

Think First

Steps to Success

1. Think - find out what you don't know and learn it by playing with it (sandboxing).
2. Design - use what you have just learned to layout a solution to the problem.
3. Test - create a series of expectations for behavior that will indicate if you have been successful.
4. Create - write the code!

Think before you act

A quick and easy way to create any type of object has been used throughout history. Why would developing software be different than everything else? It should not.

This historic approach is cyclic. This means that you take a stab at designing and then creating your item and then go back and revise one or more decisions made during design and create it again. Four simple steps are part of the cycle.

Think - What does the customer want? What types of knowledge are required to create the thing the customer wants? Do I already have this

Hero's produce what customers want. Be a hero.

knowledge? If not, where can I find this knowledge? Playing with the new knowledge allows you to see how each portion of the new knowledge behaves and can be used. Example: If you don't know how to use a nail gun you should probably not attempt to use it to build a house. Maybe you should play around with it and a a few types of boards instead of instantly starting to build. There is no replacement for experience. When programmers or software engineers play around with new ideas or libraries, they are creating *sandbox code*.

Design - Decide how the new thing your are creating should behave, look, and interact with other things. This is true of buildings, cars, toasters, baseball bats, and any other physical item you can think of, complicated or simple. This type of planning is also vital to software development. How can you create an application if you have not yet decided what it is going to do, what it will look like, and how it will interact with other pieces of software or hardware?

Test - In this step you decide what standards your product will meet when it is done. For constructing all types of buildings, this step would be the national and local building codes. In software this may consist of User Interaction testing, Unit testing, Component testing, System testing, Installation testing, or just figuring out, at a detailed level, what new data should come out of your application when it is given specific sets of data. You may be thinking, "How can I create a test for something that doesn't exist yet? I don't know how it should work." If this thought or one like it passes through your mind, then you have not sufficiently completed step 1, 2, or maybe both. Go back and do them first.

Create - Begin building. Notice that this step's description starts with begin. Once you have started, it is likely that you will find weaknesses in your knowledge, design, tests, or all of these. When you do, go back and revise your results for steps 1 - 3 immediately. This does not mean 'throw away and start all over again'. It means make modifications and try again.

A common approach taken by many software programmers goes something like this:

1. A customer asks for something.
2. The programmer starts to create it.
3. It doesn't work.
4. The programmer throws it out.
5. The programmer repeats steps 2 and 3 until it works.

6. The programmer produces something the customer didn't want.

I call this the 'oh crap!' or 'think last' approach since nearly every time the developer hits step three 'Oh crap!' or similar words are heard. The 'think last' approach all but guarantees that the software ships late, is over budget, and doesn't have the features the customer wants. It is a proven failure method. Don't use it! Use the historical approach that is known to work. **Think, Design, Test, Create**. Until you use it it may seem that the think first method will take longer than the 'Oh crap!' method but it does not. Guaranteed.

Testing

Test Types

1. Unit tests
2. Integration tests
3. System tests
4. User tests

Testing yields better software

Software testing is a vital step before shipping any software. It helps ensure that the software works as it was designed. Software shipped without being tested causes crashes, general failures, and usually doesn't work in the way the customer wanted.

There are several different types of software testing. The most common includes:

Unit Testing - low level white box testing.

> Unit testing validates the behavior of individual methods and classes. Any method that has no dependency on other portions

of the application can be tested using a good unit testing tool. The industry standard unit testing tool for Java is JUnit. JUnit tests provide a nice way to automate regression tests.

Integration Testing - medium level white box testing.

Integration testing is done on and between independent modules of the application. Some examples of modules in an application could include:

An HTTP communication module for communicating with a web server

A module that communicates with databases

A module that reads and writes files to disk

and many more.

System Testing - high level black box testing

System testing is done on the assembled application. Generally this is done by executing the user interface, if there is one, and attempting to cover every line of code in the application including erroneous uses of the functionality the application provides. Often this is done by sophisticated, automated software or by human software testers.

User Testing - high level testing

User testing is usually done before the user interface is created. A good methodology is to give potential users mockups of the interface design and ask them to accomplish a series of tasks. The user should be given no documentation on how to use the software, and the tester is not allowed to give any verbal, visual, or other types of clues regarding how to accomplish the task.

The idea behind user testing is to watch these users to see where they struggle. Anywhere they struggle is a portion of the user interface that needs to be redesigned. Redesigning based on this process yields a more intuitive user interface.

Unit Testing Hints:

Hint One:

Unit testing often involves modifying an instance of some class and then seeing if it really was modified. This modification may be direct, changing the value of an *int* type attribute, or indirect, such as adding or removing an entity to or from a collection. How then can this change be observed from within the test if the attribute is private?

One known poor solution is to make your tests part of the same package as the class being tested and then change the visibility of the attribute to ***protected***.

Such a change violates the design of the class for the expediency of the test. This is almost always bad. It leads to sloppiness of design and forces the test into the same package as the application source. This is bad design relative to making a clean release. Test code should be kept strictly separate from production code so that it is never accidentally shipped. Putting testing code in the same pack-

age increases the likelihood that the test code will accidentally be shipped. It is a bad idea.

So then is there another option? Yes. It is possible within Java to change the visibility of an attribute of a class at runtime. Imagine a class called ***Person*** with a private ***String*** attribute ***name***. Code Snippet 1 shows how to temporarily set access to this attribute to ***public*** and then test and set its value for a specific instance of the ***Person*** class.

```
Person aPerson = new Person();
Field theNameField = Person.class.getDeclaredField("name");
theNameField.setAccessible(true);
String aName = (String)theNameField.get(aPerson);
// aName could be tested for a specific value here
assertEquals("bob", aName);
// or modified as in the line of code below.
aName = "Ungala";
```

Code Snippet 1

Hint Two:

JUnit test classes can have any number of test methods. It is usual, and highly suggested to have a distinct test class per class being tested. This test class should have one test method for each method of the class being tested.

Hint Three:

Never use methods from a class you create to test another method you have created. When new to unit testing, programmers sometimes think that they should use a getter to test a setter method. They will think that they will do this to save time and reduce the number of lines of code in their tests. This is a BAD idea.

Say you have two methods, addPerson and getAllPersons. You test addPerson by instantiating a Person object, calling addPerson, and then checking to see if the Person object was added to the internal collection manipulated by the addPerson method. If you accomplish this test by calling getAllPersons and no person matching the one just added is found, which method failed, addPerson or getAllPersons? It is logically impossible to know. Use Hint One to

STEPS	RESULTS
1. Enter 'bo#b' into the 'Name' text field.	1. "Special characters such as # are not allowed in names." appears in red beside the 'Name' text field.
2. Enter 'bob' into the 'Name' text field	2. A green checkmark appears next to the 'Name' text field.
3. ..... etc.	3. ..... etc.

test this instead.

System Testing Hint:

If you are doing human driven system level testing, it, when completely created, should be a 'script' that can be executed by a human being. It generally takes the form of what the tester should do and see.

Testers usually are hired to test multiple pieces of software for the company and do not necessarily know your application. The test should be written in such a way that it can be executed by some-

one who knows nothing about how the software works or its purpose.

It is often advisable to create your test with at least two columns. The left column can then consist of a numbered set of extremely specific steps that the tester can follow. The right column can then be used to describe exactly what the tester should see. These descriptions are numbered to match the steps in the left column and have a one-to-one relationship with those steps.

Step 1 has a description of what to do. Result 1 has a description of exactly what the tester should see when step 1 is done. This is true for step 2 and result 2, step 3 and result 3, etc.

A Sample System Level Test Template

PROJECT NAME: <PROJECT NAME>

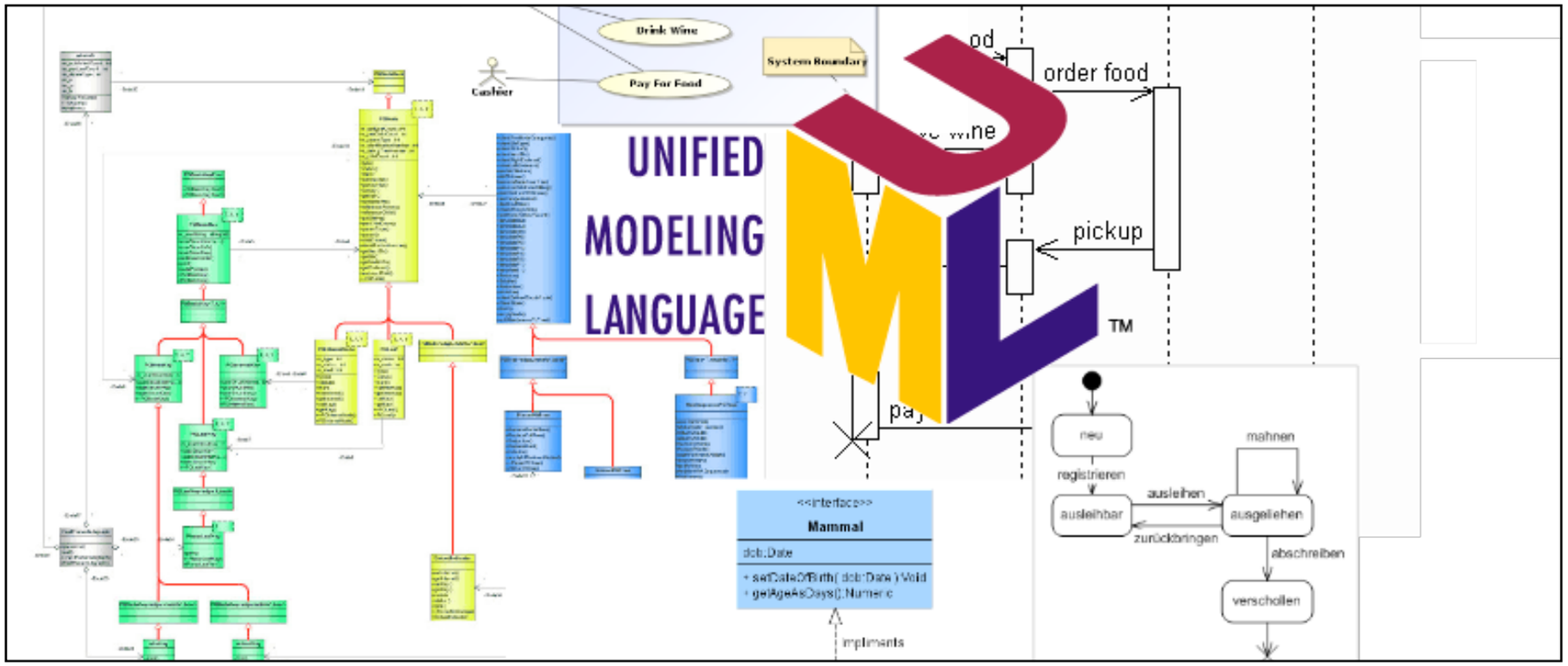
Test Script Name:			
Scenario/Purpose:			
Prerequisites:			
Name of Tester:		**Date:**	
		Time:	

Step	Description	Expected Results	Pass	Fail	N/A	Defect/Comments
1						
2						
3						
4						
5						
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						

UML

Understand

1. UML provides a standardized way to express a design
2. UML documents are created to communicate ideas
3. There are many different types of UML diagrams and documents



UML allows design to be expressed so it can be shared with others.

UML diagrams are dependent on each other. The end result of the creation of one yields information regarding how the next one should be created. Skipping documents or creating them in parallel is a mistake. Work together as a team to create each document and your team productivity will increase. Divide and conquer is not always a good way to be consistent when creating as a work team.

People in the industry create and interpret UML in many ways. Work with your manager and team to understand how you should create documents so you can communicate with them.

Several UML Diagrams And One Document In The Order In Which They Should Be Created.

DOCUMENT TYPE	DESCRIPTION	LEVEL OF DETAIL	TARGET AUDIENCE	SCOPE
Use Case Diagram	What can the user do with the application?	Low	The customer's management team	One diagram per application. One Use Case 'bubble' per user activity
Use Case Document	How does the user accomplish each activity listed in the Use Case Diagram?	Medium	The customer's management team	One document per Use Case 'bubble'. No descriptions of inner workings of the application allowed.
State Diagram	How does the application change internally? What loops, conditionals, and threads are needed?	Low	Internal production teams	One diagram per Use Case Document
Sequence Diagram	What objects, methods, parameters, and local variables are required? When complete the programmer should be able to see every code step that is to be coded including loops, cases, etc. d	High	Internal production teams	One diagram per State Diagram
Class Diagram	What are the objects, methods, attributes and inter-object relationships required within the application	High	Internal production teams	One diagram per Sequence Diagram

Table 1

A sample use Case Document Template

Name:	Use case name
Summary:	A high level description of use case
Version:	1.1 (Current revision number)
Preconditions:	1. Defines all the conditions that must be true (i.e., describes the state of the system) for the trigger to be active.
Triggers:	Describes the event that causes the use case to be initiated
Main Success Scenario:	1. Outline of steps for normal scenario (System displays "", User enters "", etc.) 2. … 3. …
Alternative Success Scenarios:	1. Outline steps for alternative scenario 1 1.1…. 1.2…. 2. Outline steps for alternative scenario 2 2.1…. 2.2….
Postconditions:	1. Describes what the change in state of the system will be after the use case completes
Business Rules:	1. List items to be validated and other business rules that need to be enforced 2. … 3. …
Notes:	1. Additional items that needed to clarified or special considerations
Author:	Authors name
Date:	Current date

Helpful Resources

Steps to Success

1. Install the tools found in the Required Tools section of this chapter.
2. Learn to launch your terminal (Cygwin terminal for Windows™ users).
3. Learn to use vim

Books such as this one are not your only resource. This section has links to resources that you will find helpful as you begin programming in Java.

IntelliJ's YouTube channel

Basic IntelliJ tutorial

A quick video of some basic terminal commands

A git installation tutorial.

Java's API, there are many.

Movie 1.1 Downloading the example files

The examples for this book can be downloaded.

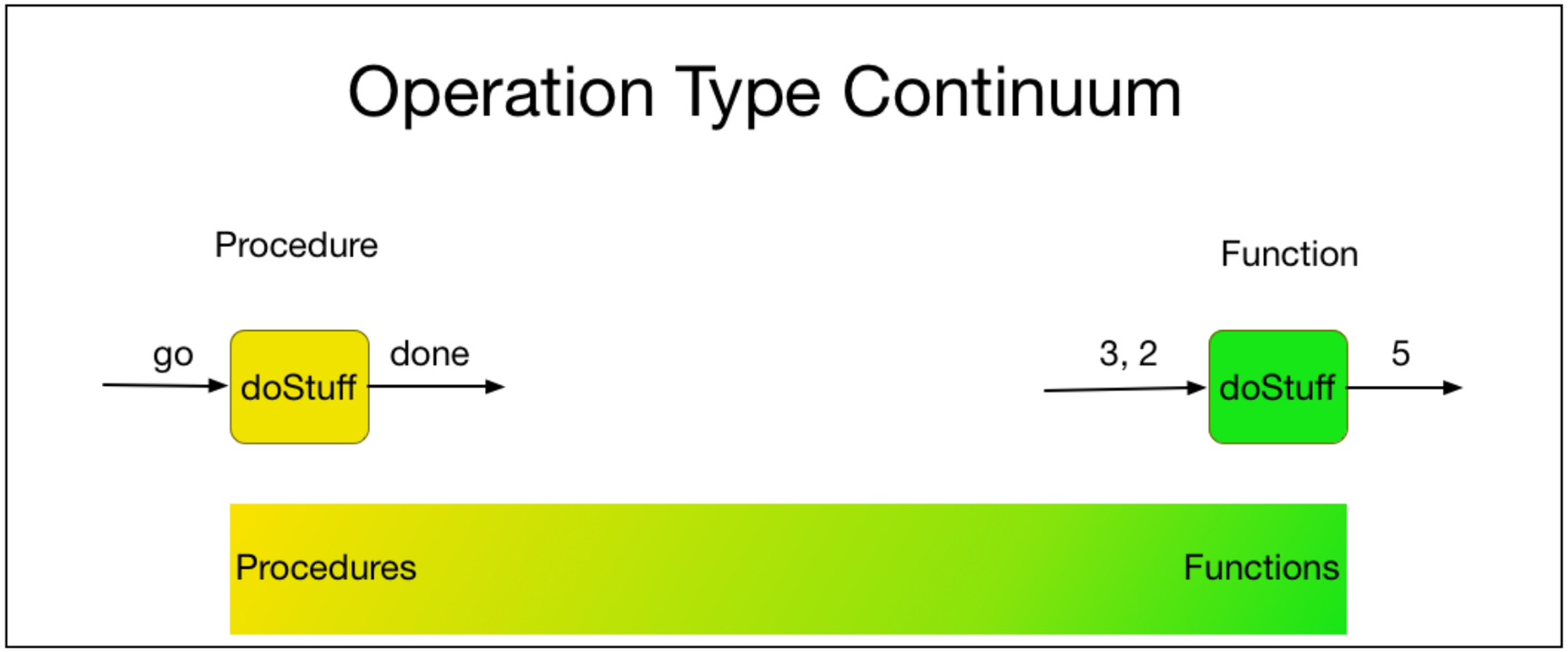
The types of variables that can be used in Java.

The operators available in Java.

Operations and Languages

Points to Ponder

1. What is an operation?
2. What types of operations are there?
3. How can common computer languages be categorized?



Operations can be a pure procedure, a pure function, or somewhere in between.

I've heard some people say they like procedural programming and don't like Object-Oriented programming. I'm not sure what they mean. It seems they're saying they like 'step-by-step' programming.

All programming, regardless of the language or language type, is created and executed 'step-by-step.' All programs are a series of operations--things to be done. Each operation is executed one after the other regardless of the type of the language.

There are two basic types of operations, procedures and functions. Procedures are things to be done that require no data to begin with, do stuff,

and let you know when they are done but don't give you back any computed results. A function requires data to do its job, does stuff, and then gives you a result back when it's done.

A physical example of a procedure is requesting a teenager to clean their room. There is no additional data needed. They clean their room. When they are done is obvious because they come out of their room. At least that is how it would be in an ideal world.

An example of a function would be an oven baking something. If you put in a bread dough mixture and a pan you get bread back out when the bake function is done. If you put cake dough and a pan in you get cake back. Either way, you get back baked goods but the type you get back depends on what you put in.

With procedures and functions available, it is common to classify programming languages as procedural, using only procedures, or functional, using only functions, even though no programming language is purely procedural or functional. Some may be more procedural and others more functional.

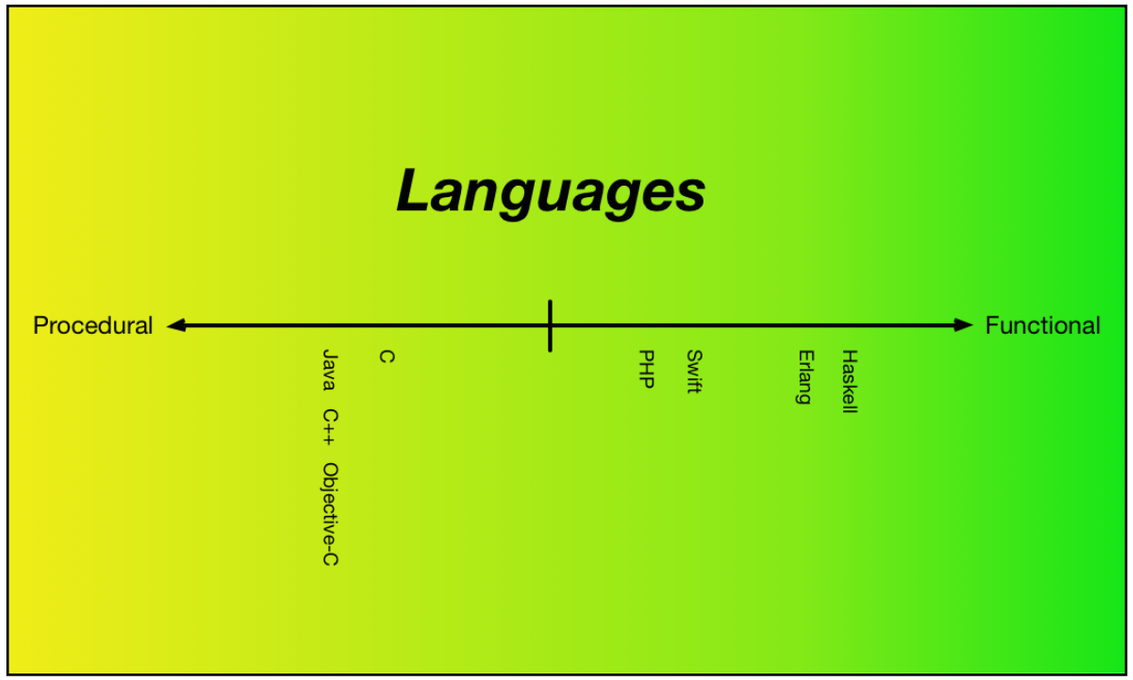


Figure 1. *A classification of some common programming languages.*

Based on the frequency of functions and procedures in the Java API, Java tends to be more procedural than functional. Using the same frequency scheme, C++ and Objective-C are about as procedural as Java. Languages such as Haskell, Erlang, and even PHP tend to be more functional than procedural.

Regardless of which type a language is, and this can vary depending on who you ask, all languages use of both procedures and functions.

CHAPTER 2

Background

In the previous book, Doing Stuff with Java, you discovered some Java basics. Let's build on that and get ready for some really powerful tools.

Constructing Things

Points to Ponder

1. What is a constructor?
2. Why use a constructor?
3. What is the difference between a class and an instance?

Moss grows and changes but it must change from a spore to a plant at some point in time.

Language is an important part of being human. Regardless of what your native language or languages are, you use them to communicate...to share ideas, discuss solutions to problems, and to express how you feel. In order to gain a common understanding, you and the people you are talking with need to understand what the other is saying. To accomplish this a shared set of words, their meanings, and rules for how to put them together to express and comprehend ideas are required...it doesn't do any good to speak Swahili to someone who only understands Chinese. So let's get some common understanding.

Java Language:

One of the nice things about Java is you can use it to represent things you see and know...trees, moss, people, customers, dogs, babies, and anything else you can think of. Then you can have them interact in defined ways. As you read the list of things, trees, etc., a picture or description of them came into your mind. Your picture or description may be a little different than mine, but through communication we can discover the differences and find a common understanding.

Assumptions of what someone else means when they speak leads to problems. Something as simple as what is meant *baby* changes depending on culture. In order to arrive at a common understanding of what baby means we would need to ask questions like, “What do you mean by *a baby*?” The response to this question would be a description of the picture in someones head. The answer wouldn’t be “Annabelle”. While Annabelle is a darn cute baby by my definition, not all babies are Annabelle. You see, *baby* is a description of a group but Annabelle is a specific individual who is a member of that group. In Java groups are called **classes** and versions of classes are called **instances**. Therefore the baby group, in Java terms, is the Baby class and Annabelle is an instance of the Baby class.

Making instances in Java requires using a **constructor**. In English it is very common to say we “construct a building.” When a building is constructed, the pieces of it are accumulated and fitted together. A Java constructor has this same purpose. The pieces of the instance, its attributes, are assembled together. They are assigned to appropriate **attributes**. Once a thing’s constructor is executed it should be all set up and ready to go.

These are the basic pieces of the Java language we need in order to communicate about constructing instances of classes. If the idea of using classes is confusing, please read through Doing Stuff with Java. It takes a more basic approach by referring to classes as groups.

Applying Language:

If you’ve worked through the *Doing Stuff with Java* book you read about the stupidity of computers. They are VERY stupid. Because of this you must be explicit when you use code to describe a class. In *Doing Stuff with Java*, a Customer class was described and some parts of the class were explained. You’ll build your understanding of what was explained there.

keywords **class name**

```
1 public class Customer{
2     String name;
3    int age;
4    double years;
5 }
```

attribute list

Code Snippet 1

Code Snippet 1 gives a minimal description of a Customer. It is all that is required, but is missing a constructor. This lack causes us to have to use Customer like was done in the *Doing Stuff* book.

```
Customer cheapCustomer = new Customer();

cheapCustomer.name = “sally”;

cheapCustomer.age = 45;

cheapCustomer.years = 5.25;
```

Code Snippet 2

Code Snippet 2 follows that naive pattern. An instance of Customer called *cheapCustomer* is made and then its attributes are assigned. While this works, it is clumsy and inefficient. We can make this code a lot better if we add a constructor to the Customer class.

```
1 public class Customer{
2       String name;
3       int age;
4       double years;
5
6       public Customer(String aName,int anAge,double someYears){
7               this.name = aName;
8               this.age = anAge;
9               this.years = someYears;
10      }
11 }
```

Code Sample - Customer.java

[Code Sample Customer.java source as file]

Adding the constructor allows the code in Code Snippet 2 to become one line.

Customer cheapCustomer = new Customer(“sally”,45, 5.25);

Let’s take a look at how a constructor is put together. If you look in the Customer.java code sample line **6** states, “There is a constructor, that any code in the application can use, for Customer. It expects to have a String called aName, an int called anAge, and a double called someYears sent to it.” Lines **7** - **9** then assign the values sent in, called **parameters** in Java, to the instance’s attributes or traits.

“But what” you ask, “is the *this* word on lines **7** - **9**???” Great question. *this* is a special word, a keyword, in Java. It means the specific instance being created. Compare **7** - **9** of the Customer.java sample and code snippet 2. See how cheapCustomer is replaced with *this*? The *this* in the constructor represents the specific customer **as it is being created**. In our example, *this* represents the cheapCustomer as it is being created. You’ll see *this* used in other ways later but make sure you understand how it is working here. Otherwise you’ll just get confused later on.

Great! Now you can speak some of the Java language and can use constructors. It’s time to move on and get some more language and understanding.

Using Instances

Points to Ponder

1. What are methods?
2. What are the two types of methods?

Things, such as trees, not only are but also do. They are active.

In our world, things are more then their attributes. They can also do stuff. A dog can bark. A customer can make a purchase. A baby can smile.

Stuff can also be done to things in our world. A bacteria can be killed. A rock can be broken. A book can be read. Sunshine can be blocked.

In Java, both stuff things do and stuff done to things are called **methods**. What these methods do is included in the class description. Doing Stuff with Java showed you methods without telling you what their Java name was. Let's look at some examples.

In Section 5 of Chapter 2 of *Doing Stuff with Java*, instances of the Customer class were created and then saved for later use by writing them into a file called customers.data. The details of how to do this are explained in the other book. Let's modify that example to use what you learned in Section 1 about constructors.

```
4  public class CustomerCreatorAndWriter {
5   Customer[] customers = new Customer[5];
6    public CustomerCreatorAndWriter(){
7
8   }
9   public void buildCustomers() throws Exception{
10      for(int i = 0; i < 5; i++){
11         Scanner systemInScanner = new Scanner(System.in);
12         System.out.printf("Enter the customer's name\n");
13         String aName = systemInScanner.nextLine();
14         System.out.printf("Enter %s's age\n", aName);
15         int anAge = systemInScanner.nextInt();
16         System.out.printf("How many years has %s been a customer?\n",
                                                                    aName);
17         double someYears = systemInScanner.nextDouble();
18         Customer aCustomer = new Customer(aName,anAge,someYears);
19         this.customers[i] = aCustomer;
20      }
21   }
22   public void saveCustomers() throws Exception{
23      File customersFile = new File("customer.data");
24      PrintWriter customerWriter = new PrintWriter(customersFile);
25      for(int i = 0; i < 5; i++){
26         Customer aCustomerToPrint = customers[i];
27         customerWriter.printf("%d %f %s\n",aCustomerToPrint.age,
                                 aCustomerToPrint.years, aCustomerToPrint.name);
28         customerWriter.flush();
29      }
30   }
31   public static void main(String[] args) throws Exception{
32      CustomerCreatorAndWriter aCreatorWriter = new
                                                  CustomerCreatorAndWriter();
33      aCreatorWriter.buildCustomers();
34      aCreatorWriter.saveCustomers();
35   }
36 }
```

Code Sample - CustomerCreatorAndWriter.java

Code Sample CustomerCreatorAndWriter.java source as file

Details:

The big differences between this updated example and the original, available from the Doing Stuff with Java gitHub repository, are found in the CustomerCreatorAndWriter class's *main* method, lines **6** - **8**, and line **19**.

CustomerCreatorWriter's *main* method, as described in the *Doing Stuff* book, is the point at which the computer begins running your code. In it, an instance of CustomerCreatorAndWriter called aCreatorWriter is constructed on line **34**. It reads, "Construct an instance of CustomerCreatorAndWriter called aCreatorWriter." aCreatorWriter's two other methods, things it can do, are then used to get information from the user to create five Customers and write those five customers to the *customer.data* file. Line **35** reads, "use the aCreatorWriter instance's buildCustomers method." This is followed by line **35**. It says, "use the aCreatorWriter instance's saveCustomers method."

While the old example and this new version do exactly the same thing, this version is more "Java-ish." It uses an instance of CustomerCreatorAndWriter where the old one did not. The other major differences are the addition of a CustomerCreatorAndWriter constructor on lines **6** - **8**, and it's use on line **19**. The constructor expects no parameters and sets no attributes. In Java this constructor falls into a group called "default constructors." These always create an instance of the class but leave the instances attributes with their default values set when the attribute was defined. The CustomerCreatorAndWriter class has only one attribute, an array of Customers called customers. Line **5** sets the customers array's default value to be an array of five customers. This is all that is needed for this example so the constructor doesn't replace the default value with something else. (For more information on arrays take a look at the *Doing Stuff with Java* book.)

Specializing

Points to Ponder

1. How are more specialized versions of classes made?
2. What does inheritance mean and how do you do it in Java?

Some groups have subgroups. Ferns are a subgroup of plants. Raspberries are another.

Many groups have subgroups. The fish group has at least two subgroups, freshwater fish and salt water fish. Trees are a subgroup of plants. Saint Bernards are a subgroup of dogs. Since Java classes are a description of groups it makes sense that there is a way to make subgroups. These are called subclasses in Java.

Another way to describe subgroups is using the word *inheritance*. It is common to say a subgroup *inherits* all the properties and abilities of the main group. Don't panic. It's just another way of saying something is a sub-group of something else. After all, fish all have heads, bodies, fins, and tails. We don't usually say, "A fancy guppy is a

small fish. It has a head, a body, fins, a tail and pretty colors." What we usually say is something like, "A fancy guppy is a small fish with pretty colors." We rely on the person we are talking to to understand all fish have heads, bodies, fins, and tails since Guppy is a subclass (subgroup) of the Fish class (group) . Or in other words, Guppy inherits all the attributes and methods (abilities) of Fish.

We've been using the Customer class in a few examples. Let's make a specialized version of Customer to represent customers who are buying clothing. This is a specialization of Customer. How about calling it ClothingCustomer. This will make it easier to distinguish from other types of customers who may be buying cars, plants, or some thing else.

If we are representing customers buying clothes then we might need to keep track of their inseam, the length of pants they need, but we will still need their name, age, and how long they have been a customer in our pretend situation. You can keep track of all of these by creating a ClothingCustomer class with only an inseam attribute IF you create it as a subclass of Customer. You must specialize it.

```
1 public class ClothingCustomer extends Customer{
2   double inseam;
3
4   public ClothingCustomer(String aName, int anAge, double someYears,
                                                        double inseam){
5     super(aName, anAge, someYears);
6     this.inseam = inseam;
7   }
8 }
```

Code Sample - ClothingCustomer.java

Code Sample ClothingCustomer.java source as file

Details:

Java's special keyword for subclassing, specializing, and inheriting is on line **1** of the ClothingCustomer.java code sample. The keyword is *extends*. I know, I know...we've already got three words to use when discussing this idea. Why do we need another??? Well, a long time ago, back in the late 1980's and early 90's, some old guys with beards decided they liked the word *extends* better than the others. They were creating the Java language so they got to choose. We are stuck with it. Whey you create your own language you can pick whatever word you want and everyone else will be stuck with it. :)

"But wait a minute," you say. "This constructor looks a little strange. I can see all the values the ClothingCustomer constructor expects on line **4**, but what is that strange 'super' thing on line **5**?" Another excellent question on your part! Sub, as in subclass, means under, smaller, or more specialized. The opposite of sub is *super*. Since ClothingCustomer is a subclass of Customer, Customer has to be the super class of ClothingCustomer. This means line **5** reads, "make sure Customer's constructor is used to set the name, age, and years for this customerand set them to be the values in aName, anAge, and someYears." The ClothingCustomer code 'reuses' the Customer code. This is a good thing.

Imagine if you had to create 15 specializations of Customer without being able to use *super*. It would not only be tedious to type the constructor code for assigning the name, age, and years repeatedly, it would also be dangerous. If you made a mistake, and then copy-pasted that mistake into all the versions, you would have to

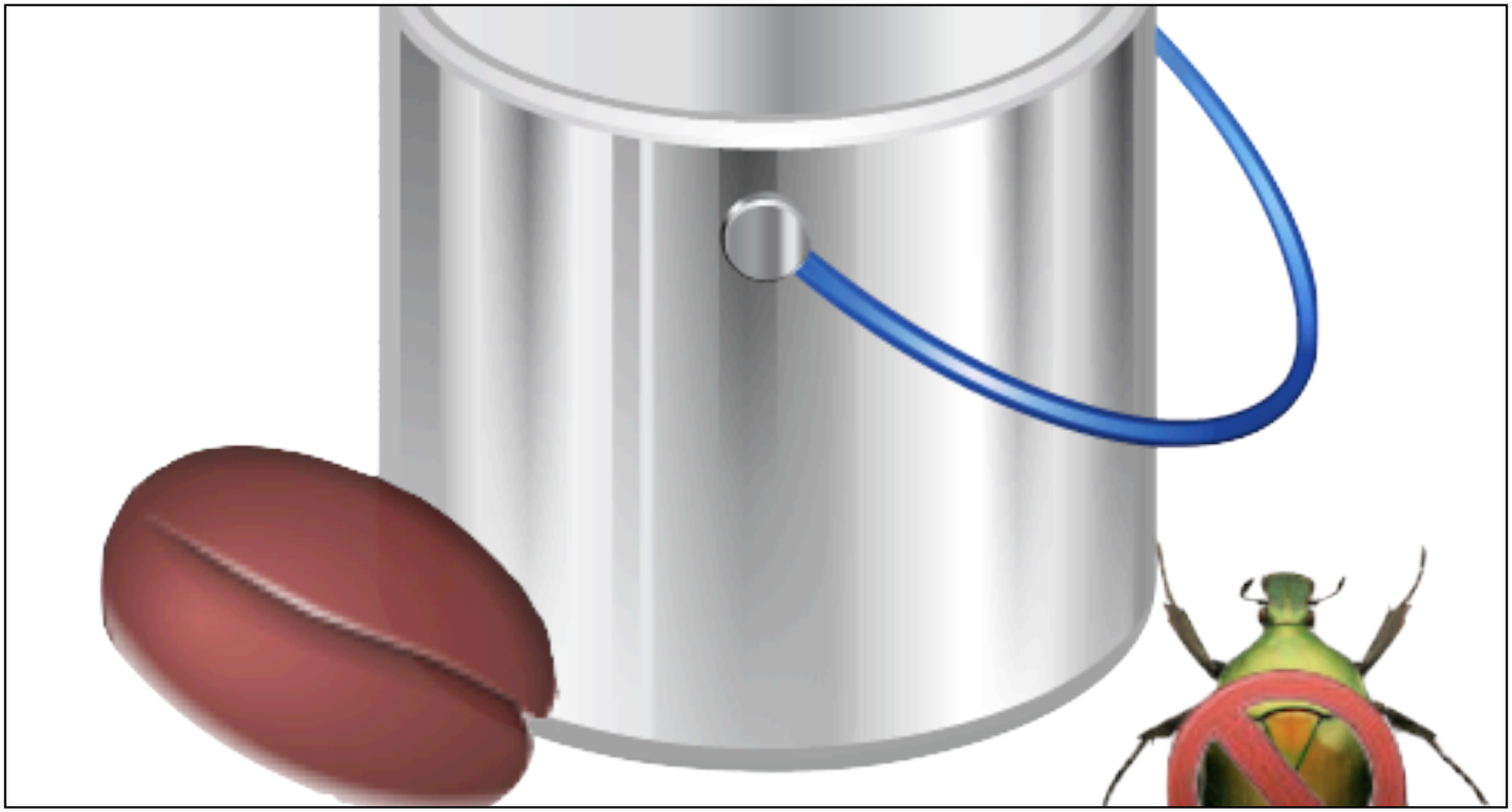
change it in all the versions to fix it. Finding all of the mistakes in this kind of situation is VERY difficult. It is easy to forget one and if you do your fix isn't a fix. It's only a partial fix. This is why we do subclassing. If we make a mistake in the Customer code we only have to fix it once and it is fixed for all subclasses of Customer. If there are 15 subclasses all 15 are fixed. If there are 200, all 200 are fixed. And, they are all fixed by changing the code ONCE. Ah hah...that's a much better solution. It removes the "code duplication" so hated in the technology industry.

That's all the important parts of specialization. We are done with the background you need to start learning and applying some powerful programming and design topics. Let's do it!

NOTICE: For the rest of this book no code examples will be referenced using line numbers. Hopefully, you are now have enough of a Java background to find the lines in the examples and snippets without being told exactly where they are. Again, don't panic. I'm sure you can do this.

CHAPTER 3

JUnit, Collections, & Beans



There are a few items that are the building blocks of all well designed and executed Java applications. These include collections of objects, a kind of objects called JavaBeans that associate data but have no functional logic, and low-level unit testing of base functionality to remove defects before shipping (JUnit tests).

JavaBeans

Points to Ponder

1. What is the difference between a JavaBean and other Java classes I have created before?
2. What is the purpose of a JavaBean?
3. How do I create a JavaBean?

JavaBeans are independent data objects that all have a similar design.

You already know how to create a JavaBean. It is a **P**lain **O**ld **J**ava **O**bject (**POJO**). It is just like other Java objects you have already made with one exception. The difference is that a JavaBean has no methods that contain behavior. It is a more industry standard design than that used for the Customer or ClothingCustomer classes in Chapter 2.

As an example, imagine that you previously had created a motorcycle object that included a method called ***beepHorn*** that would make a horn sound when called. This ***beepHorn*** method is a behavior method since it includes what normally would be thought of as a motorcycle's behavior. A JavaBean would have no such method.

Also, JavaBeans have private attributes, accessors and mutators for those attributes, and implement Serializable. If your Java object meets this minimum standard and it has no methods implementing behavior it is a JavaBean.

Implements is another Java keyword used to do subclassing (See Chapter 2 for more information on subclassing). Serializable is a part of the standard Java API. When implemented it indicates that instances of the subclass can be converted into a single, linear array of bytes in preparation to sending the array of bytes across a serial network cable or saving the array of bytes to disk. Don't worry too much about it right now. There is a discussion of Serializable in Chapter 4.

An example of a JavaBean can be seen in the PersonBean.java Code Sample.

```
public class PersonBean implements Serializable{
 private String firstName;
 private String lastName;
 private int age;

 public PersonBean() {
 }
 public String getFirstName() {
    return this.firstName;
 }
 public void setFirstName(String firstName) {
    this.firstName = firstName;
 }
 public String getLastName() {
    return this.lastName;
 }
 public void setLastName(String lastName) {
    this.lastName = lastName;
 }
 public int getAge() {
    return this.age;
 }
 public void setAge(int age) {
    this.age = age;
 }
```

Code Sample - PersonBean.java

Code Sample PersonBean.java source as file

Other methods and constructors can be added to a JavaBean as long as they only set attribute values and include no behavior. For example you could add an additional constructor where the values for the ***lastName***, ***firstName***, and ***age*** attributes were passed in as parameters.

It is also common, though not required, to add the ***hashCode*** and ***equals*** methods. If you are using IntelliJ™ to create your application these methods can be inserted for you. To do this right-click somewhere in the source code, select **Generate...** in the popup menu, and then select **Equals and Hash Code.** Make sure you select all of the bean's attributes, called fields in IntelliJ, as you go through the multiple dialog boxes.

The ***equals*** method is used to compare two instances of the JavaBean. An example for the PersonBean class is found in Code Snippet 1.

```
public boolean equals(Object o) {
   if (this == o) return true;
   if (o == null || this.getClass() != o.getClass()) return false;

   PersonBean that = (PersonBean) o;

   if (this.age != that.age) return false;
   if (!this.firstName.equals(that.firstName)) return false;
   return this.lastName.equals(that.lastName);
}
```

Code Snippet 1

Code Snippet 2 contains the PersonBean's ***hashCode*** method. This method is used to get an integer representation of an instance of the bean. This integer is also used for comparing bean instances.

```
public int hashCode() {
    int result = this.firstName.hashCode();
    result = 31 * result + this.lastName.hashCode();
    result = 31 * result + this.age;
    return result;
 }
```

Code Snippet 2

The biggest mistake made by those new to JavaBeans is to overthink them. They truly are as simple as this description. The only thing that makes JavaBeans special is what they don't do (behavior). This is why they are referred to as POJO's (**P**lain **O**ld **J**ava **O**bjects).

Your application should be using JavaBeans, where appropriate, to describe and store data. They are a great way to store data in RAM and they come in very handy when you need to store the data to disk or send it across a network.

Collections & Maps

Points to Ponder

1. Why are there different types of collections?
2. When would I want to use each of the different types of collections?
3. Why use collections instead of arrays?

Collections (groups) and Maps (key - value pairs) allow data to be stored and retrieved

There are two main ways to group objects in Java, collections and maps. Collections come in many types the most common being:

- **ArrayList** - a collection that is accessed like an array and can be sorted
- **TreeSet** - a collection where on object can be added only once and the objects are presorted, and
- **LinkedList** - an unsorted collection where objects can be added any number of times. This class has FIFO (**F**irst **I**n **F**irst **O**ut) behavior us-

ing its ***push*** and ***pop*** methods to add and remove objects to and from the list.

Maps also come in different flavors but the most common being:

- **HashMap -** an unsorted set of key/value pairs. The key is used to get the value from the map after it has been set. Adding and getting are done using the ***put*** and ***get*** HashMap methods. You can also get a list of all of the values from a HashMap but they will be in no defined order, and
- **TreeMap -** a sorted tree of values. The values are sorted by the map ordering of the keys, not the values. Both the ***put*** and ***get*** methods work just like in a HashMap but TreeMap also has several methods you can use to get the keys in order. These methods are ***firstKey***, ***higherKey***, ***lastKey***, and ***lowerKey***. With these methods you can then use HashMap's *get* method with the key they return to get the value associated with each key.

All of these collections are mutable where as primitive arrays are immutable. This means that it is not possible to increase the size of a primitive array after it is created but collections can change their size. This is done for you when you add objects to the collection or remove them.

Code Snippet 1 is an example of using a HashMap. The map contains the managers for some company. It is designed to have a String as the key and a PersonBean as the value. This is done by giving the HashMap **parametric types** (more Java language). In the code example this is done by including ***<String, PersonBean>*** in the variable declaration for ***companyMap***. The HashMap constructor call, ***new HashMap,*** one requires the use of the <> characters. Your HashMaps can use any type of Java object for the key and any type of object for the value.

There is NO requirement that the key be a String. The value associated with the key need not be a JavaBean. They can be anything your application design needs.

```
PersonBean fredPerson = new PersonBean();
fredPerson.setFirstName("Fred");
fredPerson.setLastName("Barney");
fredPerson.setAge(21);

PersonBean annaPerson = new PersonBean();
annaPerson.setFirstName("Anna");
annaPerson.setLastName("Barney");
annaPerson.setAge(23);

HashMap<String, PersonBean> companyMap = new HashMap<>();

companyMap.put("vicePresident",fredPerson);
companyMap.put("president", annaPerson);
```

Code Snippet 1

You do not have to include parametric types if you do not want to. If you do not then you can mix many types of Java objects as the keys and many types for the values. This is different than the native arrays you have used in the past. Those could only contain values of one type.

Code Snippet 2 shows how to add the annaperson and fredPerson JavaBeans to an ArrayList. Notice that there is no key. Only

```
ArrayList<PersonBean> companyList = new ArrayList<PersonBean>();

companyList.add(fredPerson);
companyList.add(annaPerson);
```

Code Snippet 2

the PersonBeans are added.

There are many other types of collections. Examples of them and how to get objects back out of the collections are easily found on the web and so will not be included here.

Movie 3.1 An Analogy

Maps and Lists and Markets; Oh My!

Design, Test Matricies, & Tests

Points to Ponder

1. Why would anyone create a test matrix?
2. How is it possible to create tests for code not yet written?
3. Is creating tests before code is written a waste of time?

The easiest way to get rid of bugs is not create them.

Now that you have a basic understanding of what JavaBeans and Collections are, let's start putting together examples that use them. So that we don't have to waste time let's follow the **Think-Design-Test-Create** approach(See Chapter 1, Section 2). This will help you see how one step in the approach tells you what to do in the next step. Each of these steps should be done with all the members of your team. Do not try to 'divide and conquer' and do them in parallel. It won't work for you.

Steps:

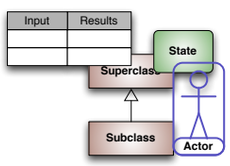
Think:

Evaluate the following customer requirements. Your customer wants a Java class that allows them to store their customer phone numbers and their employees. Both should be retrievable directly by a String that is "<lastName>, <firstName>". The customer phone numbers must also be able to be printed out alphabetically by "<lastName>, <firstName>; ". This String must contain the phone numbers for all of the customers.

Based on these requirements it sounds like there is a need to know about the HashMap and TreeMap classes. Some sandbox code, code where you play with these items and has nothing to do with the solution, is needed. Play with them, adding objects to them, getting objects that have been added, and remove objects from them. You may also need to ask the customer more questions about their requirements if they are unclear.

Design:

Based on the playing with the HashMap and TreeMap classes and the requirements listed by the customer a new kind of object with several methods are needed. Discussions with team members has lead to the UML Class diagram seen in Diagram 1.



The team has decided that there should be a class called Business with the two attributes and six methods seen in the diagram above. Each method has had the method name, parameter list, and return types defined. The team has also decided that there should be a one-to-many relationship between (1) Business object and (many) Person objects. Diagram 1 shows the one-to-many relationship between the new Business class created to fulfill the customer requirements and the Person JavaBean seen earlier in this Chapter.

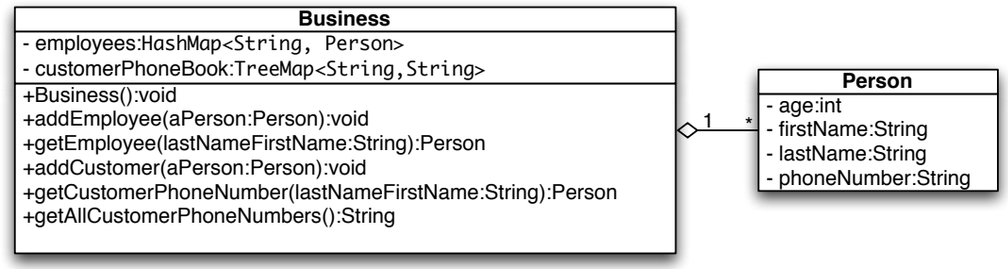


Diagram 1

Diagram 1 as image

The Business class' two attributes are a HashMap to hold the employees and a TreeMap to hold the customer phone numbers. A TreeMap was chosen for the phone numbers because they must be printed out in a sorted fashion and TreeMap automatically sorts objects as they are added. A HashMap was chosen to hold employee Person objects since there is no need for sorting and HashMap uses fewer computational resources than TreeMap does when adding objects since it doesn't need to sort them.

Each method indicated in Diagram 1 was designed to meet the need of the customer as stated in 'needing to store' requirement and 'directly retrieve', based on last and first names, both customer phone numbers and employees. The requirement for a String with all of the customer names and phone numbers is met by the ***getAllCustomerPhoneNumbers*** method.

Test:

Now that we have designed the class or classes we are going to create it is time to design the JUnit tests. We do this by creating a simple test matrix. Simple test matrices are usually created as tables. Each matrix consists of the name of the Class and methods being tested, the input to be used by the method, and the result of the method call given the specified input. See Test Matrix 1 below.

Usually the rows of a test matrix contain at least one 'Happy Path' per method being tested. This is where the method under test is given input that would normally be expect to work without any issues or special handling. Once you have identified the 'Happy Path' it is now time to think terrible thoughts.

When thinking these terrible thoughts you come up with as many what-ifs as you can think of that might cause the method under test to fail. If the method normally handles positive numbers make sure you check for input such as zero, negative values, the largest possible positive number of that type, and the largest possible negative value of that type. Others may be needed depending on what your method is going to do.

If your method is passed objects as parameters you should use nulls as inputs and possibly even objects that haven't had all of their attributes set correctly. Which attributes you leave undefined or give bad values to depends on which attributes are used in the method. Test Matrix 1 shows these principles in action.

Notice that the constructor is not included in the test matrix. It is not included in this example since it does nothing but instantiate the objects attributes without any conditions. It would be very strange for it to have errors.

Business Class
addEmployee

INPUT	RESULTS
4 complete Person objects	employees HashMap has 4 entries
nullPerson object	employees TreeMap has 4 entries
Person object without a first name	employees TreeMap has 4 entries
Person object without a last name	employees TreeMap has 4 entries

Test Matrix 1a

Business Class
getEmployee

INPUT	RESULTS
"Jones, Bob", "Gomez, Jose", Anderson, Sven", or "Anderson, Ingabrit"	each returns a valid Person object
"Body, No"	null returned

Test Matrix 1b

Business Class
addCustomer

INPUT	RESULTS
4 complete Person objects	customer TreeMap has 4 entries
null Person object	employees TreeMap has 4 entries
Person object without a first name	employees TreeMap has 4 entries
Person object without a last name	employees TreeMap has 4 entries

Test Matrix 1c

Business Class
getCustomerPhoneNumber

INPUT	RESULTS
“Jones, Bob” or “Anderson, Sven”	“(208)555-1818” or “(208)888-1598”

Test Matrix 1d

Business Class
getAllCustomerPhoneNumbers

INPUT	RESULTS
void	"Anderson, Ingabrit - (208)888-1599; Anderson, Sven - (208)888-1598; Gomez, Jose - (208)888-1900; Jones, Bob - (208)555-1818; "

Test Matrix 1e

Remember that you need to think of how each method should behave when it is given good data to work with as well as any kind of bad data you can think of. You might think that you can skip this step because you will think of all of these anyway when you are writing the test or, heaven forbid, the code. It will not be so. If you try to think of all of these possibilities while you are creating your unit test, or even worse when you are creating the code, you will forget some. There is even a much higher probability that you will not even see some of the possible bad options if you don’t list them in the test matrix.

You want to create these test matrices because you don’t want to get that nasty phone call at 2:00 AM telling you that the code you wrote is broken and you had better get down to the office right now and fix it. You won’t like getting such calls and your manager won’t like making them. If you get too many of them

you won't be employed for long. Make the test matrices and run them past your team members. They will be able to pick out possibilities you have missed.

Having used Test Matrix 1 to design the tests needed, a JUnit test class now can be created. In this JUnit test there is to be one test method for each method to be tested. You can see in the BusinessTest.java example (*BusinessTest.java as file*) each of the test methods are named "test<methodName>". This makes it easy to create and control the tests. If multiple Business class methods were tested in one test method, the tests could become unwieldy to read and support. Make your life easier. Write one test method per class method.

The JUnit test is large and easier read separately from this book. Please download it.

In addition to the test methods, the JUnit test class has a ***setUp*** method that executes only once when the test is started and before any of the test methods are called. In this example the setup method is used to instantiate the attributes for the BusinessTest class. This means that these BusinessTest attributes can then be reused in all of the other methods of the BusinessTest class.

Notice that the ***addEmployee*** method is not used in the test for the ***getEmployee*** method. All JUnits you create should behave this way. If you use the ***addEmployee*** method to test the ***getEmployee*** method and there is a failure, which method failed? It is not easily knowable and can take quite some time to figure out.

"But" you say,"hasn't the ***addEmployee*** test already completed? Won't that ensure that I can safely use it to test ***getEmployee***?" Sorry but no. There is no guarantee that the ***testAddEmployee*** method ran prior to the ***testGetEmployee*** method.

Write all of your unit tests independent of each other. Never make them dependent on each other. If you do, you are asking for trouble.

The add, get, and list customer method tests seen in Code Sample 6 are very similar to the employee tests. When the two test groups are compared there is one additional test, ***testGetAllCustomerPhoneNumbers*** in the customer group.

This test assures that given some number of customers that the phone numbers are printed out in sorted order by <lastName>, <firstName>.

The JUnit test, as seen in the download, now appears to be complete. It has followed the design decided on when the team put together the UML Class diagram and meets the customers' requirements for behavior. Now it is time to move on to the next step, but remember that you can and should come back and modify the test matrix and the unit test if there is something that has been left out or is discovered later.

Create:

Now that the unit test has been completed it can be used to run the Business class code as you create it. You can tell when your each of the Business class methods is complete because it passes its associated test. If you are creating the code and realize that a condition was left out, an

additional method is needed, or some other change is required you should IMMEDIATELY cycle back to the **Think** and **Design** steps and make any needed changes to your knowledge, the design, the test matrix, and the tests.

Since the tests created in the previous step (Test) tell us how the code should behave it now becomes much easier to write the code. All we have to do is use the input provided for each call and match the expected output.

The Code Example Business.java includes all of the Business class methods. These were created after the test matrix and JUnit tests were completed.

```
package com.real.java.example;

import java.util.HashMap;
import java.util.TreeMap;
import java.util.TreeSet;

public class Business {
   protected HashMap<String, EmployeeBean> employees;
   protected TreeMap<String, String> customerPhoneBook;

   public Business() {
      employees = new HashMap<String, EmployeeBean>();
      customerPhoneBook = new TreeMap<String, String>();
   }

   public void addEmployee(EmployeeBean anEmployee) {
      if (anEmployee != null && anEmployee.getLastName() != null &&
            anEmployee.getFirstName() != null) {
         String key = anEmployee.getLastName() +
               ", " + anEmployee.getFirstName();
         employees.put(key, anEmployee);
      }
   }

   public EmployeeBean getEmployee(String lastAndFirstName) {
      return employees.get(lastAndFirstName);
   }
```

Code Sample - Business.java - a

Code Sample Business.java source as file

```
  public String getCustomerPhoneNumber(String lastAndFirstName) {
    if(lastAndFirstName == null){
       return null;
    }
    return customerPhoneBook.get(lastAndFirstName);
}

public String getAllCustomerPhoneNumbers() {
    StringBuffer phoneNumberBuffer = new StringBuffer();
    TreeSet<String> sortedKeys = new
                                    TreeSet<String>(customerPhoneBook.keySet());
    for(String aKey : sortedKeys){
       String phoneNumber = customerPhoneBook.get(aKey);
       phoneNumberBuffer.append(aKey);
       phoneNumberBuffer.append(" - ");
       phoneNumberBuffer.append(phoneNumber);
       phoneNumberBuffer.append("; ");
    }
    return phoneNumberBuffer.toString();
  }

  public void addCustomerToPhoneBook(CustomerBean aCustomer) {
    if (aCustomer != null && aCustomer.getLastName() != null &&
          aCustomer.getFirstName() != null) {
       String key = aCustomer.getLastName() + ", " + aCustomer.getFirstName();
       customerPhoneBook.put(key, aCustomer.getPhoneNumber());
    }
  }
}
```

Code Sample - Business.java - b

You can see that the HashMap and TreeMap are instantiated in the constructor. Also notice that the ***add*** methods include validation code to make sure that only appropriate Person objects are added to the Maps.

The ***getAllCustomerPhoneNumbers*** method uses a StringBuffer to assemble the phone numbers and their matching names. There is also a loop that retrieves the ordered keys for the ***customerPhoneBook*** TreeMap. Each key is retrieved in the sorted order and then used to get the phone number that is its matching value.

The other get methods use the key to pull the appropriate Person or phone number value without any looping. Looping is unnecessary in those methods since Maps can directly retrieve the related values as you can see in the Business.java code sample.

The ***add*** methods show how to assemble the key from the first and last names. Once they are assembled the keys are used to do the insertion of the key/value pair using the ***put*** method.

"But" you say, "couldn't an ArrayList be used for the employees attribute instead of a HashMap?" Yes it could. If there is very little use of the ***getEmployee*** method then it would be possible to switch to using an ArrayList. Code Snippet 1 shows the ***addEmployee*** and ***getEmployee*** methods changed to use an ArrayList. Notice that the ***addEmployee*** method is simpler than

the version in Code Example Business.java but that the ***getEmployee*** method is more complex.

Notice that if you made these changes they would be hidden from code outside the Business class. This is a standard Object Oriented design approach. There would be no need to change the JUnit test. The test would still require the same behavior since no customer requirements have changed.

```
public void addEmployee(EmployeeBean aPerson){
   if(aPerson != null
         && aPerson.getLastName() != null
         && aPerson.getFirstName() != null){
      employees.add(aPerson);
   }
}
public EmployeeBean getEmployee(String lastAndFirstName){
   EmployeeBean retVal = null;
   if(lastAndFirstName != null{
      for(int i = 0; i < employees.size(); i++){
         EmployeeBean anEmployee = employees.get(i);
         if(lastAndFirstName.equals(anEmployee.lastName
               +", "+anEmployee.firstName) ){
            retVal = anEmployee;
            break;
         }
      }
   }
   return retVal;
}
```

Code Snippet 1

The UML class diagram would need to change slightly if you made these changes as the HashMap would need to be replaced by an ArrayList in the design. Overall, not many changes to diagrams would be needed if this code change were made.

Here is why getEmployee got more complex. When attempting to find a value in any ArrayList you must examine all of the entries until you find the one you are looking for. If your list is unsorted the number of checks you have to do to find what you are looking for averages out to be 1/2 the number of entries. In other words if you have 100 items in the list, on average you will have to do 50 comparisons to find what you are looking for. 500 items means 250 comparisons. Primitive arrays behave this same way. This high count in order to find things means you could be wasting large amounts of computer resources and your application may not scale for large numbers of users.

If you are not using the ArrayList for lookups but instead are temporarily adding items to the list to be 'popped' off later then an ArrayList could be just the thing you are looking for. If you are doing searchers HashMap may be a better choice.

Having completed the UML, the test matrix, the JUnit test, and the code you can now feel confident in the code that has been created. You and your team have looked over each step to attempt to make the code as error proof as possible.

CHAPTER 4

Serialization and File IO

Serialization is an easy way to convert objects in memory to other representations. What you choose to do with these representations is up to you. You could store them to a file on disk, send them over a network, display them to a user, or anything else you can imagine.

Serialization and Streams

Points to Ponder

1. Why do serialization when every Java object has a toString method?
2. Why are there different types of serialization?
3. Why use streams? Why not write directly to a file?

This stream moves sediment from the Teton Mountains. Eventually this sediment might enter the Pacific Ocean.

JSON, XML, and binary serialization. Each of these has its place. Which one you choose depends on the situation you find yourself in. Obviously if the data you need to interact with already exists in one of these formats that is the one you would use. But what if you have a free hand?

There are proponents for each of these types of serialization and the proponents are all very convincing. If, however, you want to use the most modern specification that uses the fewest characters the choice is obvious. JSON.

Choosing JSON yields human and machine readable code that minimizes the size of the data to be stored or sent. This can become significant when, instead of writing to a file, you and millions of your applications' users are sending the data to one or more machines across a network.

Working with a web server also limits our choice. Binary serialization doesn't fit naturally with most of the API's available in the languages commonly used in web servers. That alone could mean we might want to choose JSON or XML.

XML tends to be very wordy. An XML representation of data contains lots of characters that aren't the data. These extra characters take valuable network and CPU time on mobile devices, servers, and networks. Each character handled means more battery power consumed, more CPU used, and more network bandwidth consumed.

Since mobile devices are battery constrained this can become a major issue when large amounts of data, either in one big chunk or even worse lots of small ones, are sent to the mobile device and then de-serialized (inflated). This battery drain decreases the likelihood of selecting XML serialization for mobile applications but not necessarily for other types of applications. In fact, XML has been very popular for a long time and still is. For example, Android™ developers use XML to describe their app's user interface. Open source linux projects use it a lot too.

Regardless of which type of serialization is best for your situation it is very common, and a very good idea, to use one type of serialization throughout an entire system. Do not select one type of serialization for data transfer and then another one for data persistence. If you choose to ignore this advice it will complicate the source code of your application. The greater the amount of complication in your application the harder it is to produce, support, and sell.

Also, do not pick a type of serialization for your application or system of applications just because you are familiar with it. It should be a consideration, but not the only determining factor. Be different than most of the human race and keep an open mind. Make the best choice.

JSON Serialization

JavaScript Object Notation (JSON: pronounced like the name Jason in English) came into being around 2001. Its purpose is to allow for a language independent way of transmitting and storing data. It became one of the many such text based data transfer specifications such as XML. JSON was originally was used with JavaScript however implementations of JSON libraries exist in many different languages.

The JSON specification allows for several different types of elements. The two most obvious are arrays and objects. Arrays in JSON are contained within square brackets and the individual elements of the array are separated by commas. And example of the same array declared in both Java and JSON is seen in Figure 1.

```
Java – {1, 2, 75, 13}
JSON – "[1, 2, 75, 13]"
```

Figure 1

JSON objects are similar to HashMaps in that they both have keys and values. A JSON object representation does not include representations of methods. Nor do JSON objects include an indicator of what type the object is. Only attribute names as keys and the attribute values as values for the JSON keys are used. Diagram 1 shows the UML class diagram of a Person class consisting of four attributes:

1. age
2. firstName
3. lastName
4. phoneNumber

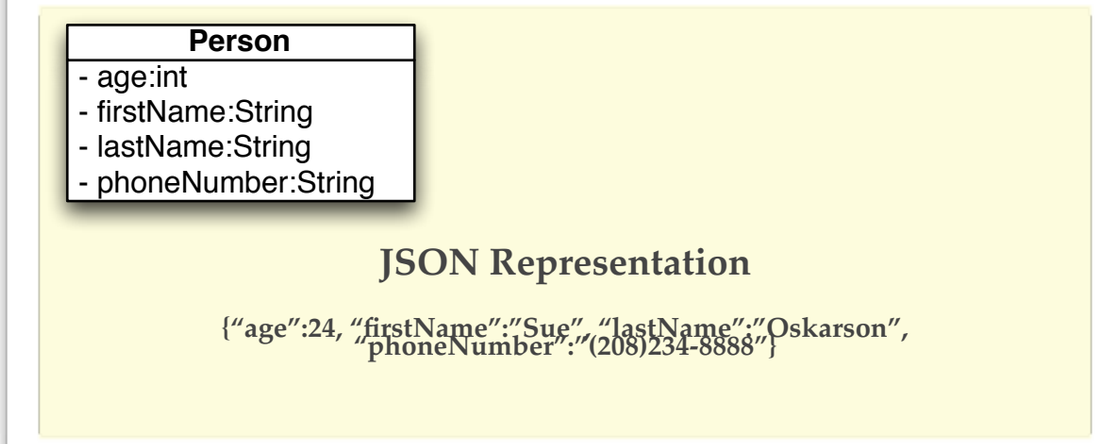


Diagram 1: UML and JSON representations

The JSON String representation of an instance of the ***Person*** class is shown directly below the UML class diagram in Diagram 1. Notice that the JSON keys are the class attribute names and the JSON values are the Java class instances' attribute values. Also notice that JSON objects begin and end with the { and } characters instead of the [and] characters. Remember, [and] are used in JSON to represent arrays.

Each key/value pairing in a JSON string is indicated by a colon character and the pairs are separated by commas. The keys in JSON representations of objects must be strings but the values can be strings, boolean true/false values, numbers as integers or doubles, objects, or arrays.

Parsing and Getting JSON Strings

There are many JSON libraries and several of these are written in Java. Some are more complicated than others yet all yield essentially the same results. The library used in these examples will be

the QCJSON library available at *https://www.github.com/yenrab/qcJSON*.

Code Snippet 1 shows a simple Java class, ***TestObject***. Code Sample 2 shows an Instance of ***TestObject*** being created and then written to the console.

```
public class TestThing implements Serializable {

   private String theString;
   private int theInt;
   private Date theDate;

   public TestObject(String aString, int anInt, Date aDate) {
      theString = aString;
      theInt = anInt;
      theDate = aDate;
   }

}
```

Code Snippet 1

```
   TestObject anInstance = new TestThing("Hello there.", 7,
         new Date(1067899));
   try {
      String jsonString = JSONUtilities.stringify(anInstance);
      System.out.println(jsonString);
   }
   catch (JSONException e) {
      e.printStackTrace();
   }
}
```

Code Snippet 2

The result printed out in the console is a JSON string.

```
{"theString":"Hello there.","theInt":"7", "theDate":"Wed Dec 31
17:17:47 MST 1969"}
```

Converting a JSON string into a Java instance is done in much the same way. Code Snippet 3 shows the string being converted back into a ***TestThing***.

```
try {
   String jsonString = "{\"theString\":\"Hello there.\",\"theInt\":\"7\",
                                 \"theDate\":\"Wed Dec 31 17:17:47 MST 1969\"}";

   HashMap aMap = (HashMap)JSONUtilities.parse(jsonString);
   String aString = (String)aMap.get("theString");
   int anInt = Integer.parseInt((String)aMap.get("theInt"));
   String aDateString = (String)aMap.get("theDate"));
   SimpleDateFormat aFormatter =
                        new SimpleDateFormat("EEE MMM d HH:mm:ss z yyyy");
   Date aDate = aFormatter.parse(aDateString);

   TestThing anotherInstance = new TestThing(aString, anInt,
         aDate);
}
catch (JSONException e) {
   e.printStackTrace();
}
```

Code Snippet 3

To convert a JSON string to an instance you use the ***JSONUtilities.parse*** method passing it the JSON string to be converted. If it is a proper JSON string, the object created will be a ***HashMap*** if the string defines an object. An ***ArrayList*** will be re-

turned if the string describes an array. If the string is not a proper JSON string a JSON Exception will be thrown.

Streams and JSON

Java has two basic types of streams that are used to move information, ***InputStreams*** and ***OutputStreams***. ***InputStreams*** are used to pull data in to your application. You push data out using ***OutputStreams***.

There are several classes that inherit from ***InputStream*** and ***OutputStream***. Two of these, used to read and write data to files, are ***FileInputStream*** and ***FileOutputStream***. For a basic description of these streams take a look in the Doing Stuff with Java book.

Java also allows you to 'wrap' these streams with additional functionality as you need or want. The QCJSON library takes advantage of this by creating two 'wrappers', ***JSONOutputStream*** and ***JSONInputStream***. As seen in Code Snippet 4, if you want to write JSON to a file, instead of using the JSONUtilities parse and stringify methods you use these two JSON streams. You do this by sending a ***FileOutputStream*** object to the constructor of the ***JSONOutputStream***.

```
File aFile = new File("test.json");
try {
   FileOutputStream aFileStream = new FileOutputStream(aFile);
   JSONOutputStream jsonOut =
         new JSONOutputStream(aFileStream);
   jsonOut.writeObject(aSerializableObject);
   jsonOut.close();
}
catch (Exception e) {
   e.printStackTrace();
}
```

Code Snippet 4

Once the ***JSONOutputStream*** has been wrapped around the ***FileOutputStream*** you can then use the ***JSONOutputStream*** method ***writeObject*** to write most any ***Serializable*** Java object to the file.

Getting data back in from a file is done in a very similar fashion. To read an object you wrap a *FileInputStream* in a *JSONInputStream*. Then you call the *JSONInputStreams readObject* method. The result of this call is either a *HashMap* or *ArrayList*. You will get back one of these since *readObject* behaves the same as *parse* and *stringify*.

Diagram 2 is a UML Sequence diagram that can help you see how these streams interact. It shows an existing application of some unknown type and an already existing Serializable object that is to be serialized. It shows how this stream interacts with the FileOutputStream and the Serializable.

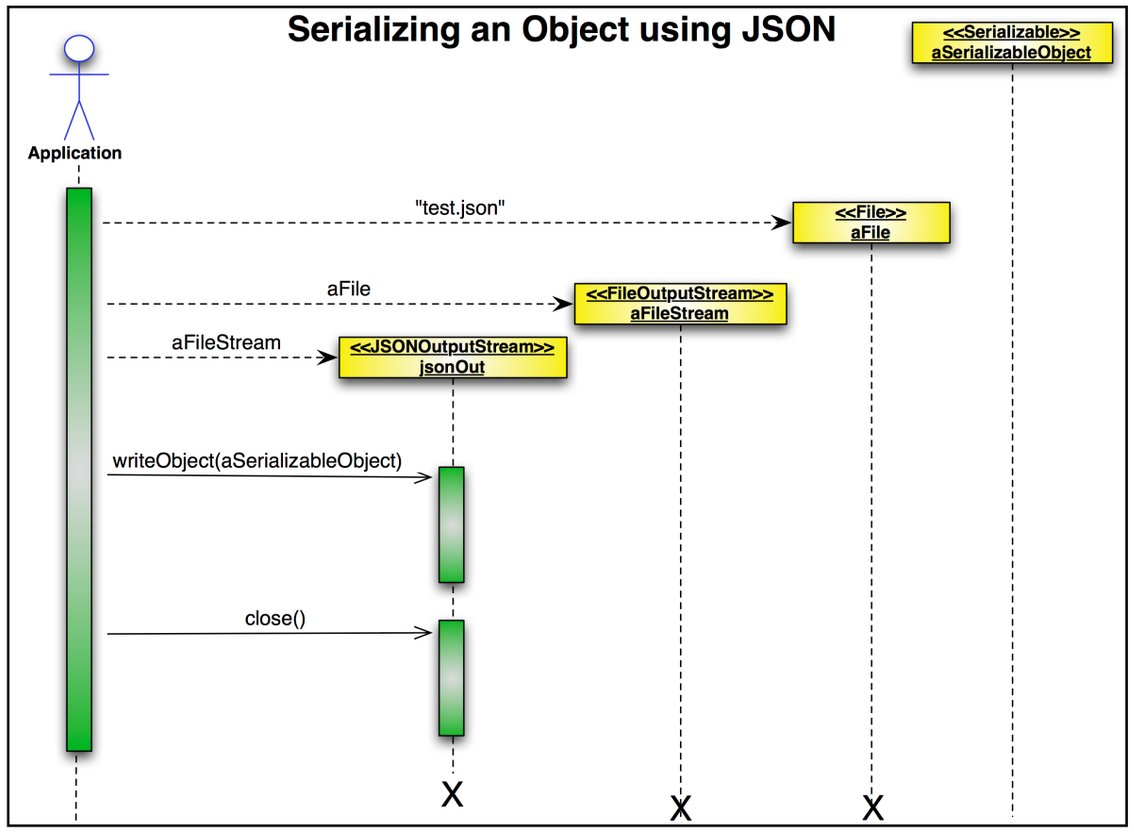


Diagram 2: a partial UML Sequence Diagram showing JSON IO

Diagram 2 as image

As with all UML Sequence diagrams time is shown flowing from top to bottom, elements that already exist, like the App and the aSerializable object instance, are listed across the top of the diagram, and instances of objects that are created during the execution of the code are listed top-to-bottom in the order they are created, and the life lines of each item is shown as a dotted line extending vertically from the description box. Any objects that go out of scope and are garbage collected must have an X on the end of their life line to show that they no longer exist.

Binary Serialization

JSON is far from the only way to serialize Java objects. Another way is part of the standard Java API and uses a Java specific binary specification. This specification is not implemented in other languages. This means that this binary serialization can only be used between or within Java applications.

The standard Java classes provided to accomplish this are the *ObjectOutputStream* and *ObjectInputStream*. Like the JSON output and input streams these are used by wrapping some other stream. And like the JSON streams, their *writeObject* and *readObject* methods are called to send and receive data.

```
File aFile = new File("test.bserial");
try {
   FileInputStream aFileStream = new FileInputStream(aFile);
   ObjectInputStream objectIn =
         new ObjectInputStream(aFileStream);
   Person aPerson = (Person)objectIn.readObject();
   objectIn.close();
}
catch (Exception e) {
   e.printStackTrace();
}
```

Code Snippet 5

Notice that if you use this method of serialization, the object read in can be directly cast to the type that was originally stored using the *writeObject* method. This is one advantage of using the standard binary serialization streams. A problem in selecting them for use is that if you change the class file then any data being read in

that were saved out with the old form of the class will no longer be loadable from the file.

Since the data for the object is being serialized in a binary format it will not be human readable. This is not a drawback unless you need to have it be human readable or if it would make it easier to test if it were. If such is the case you should select one of the other serialization methods, JSON or XML.

XML Serialization

Another popular choice for serialization is to use XML to save and create them. There are many XML tutorials on the web. Some of them are very good. If you need to know what XML is, please look there as this book can only give a very rudimentary explanation.

While there are several different ways of doing XML serialization in the standard Java distribution, the ***java.beans.XMLEncoder*** and ***java.beans.XMLDecoder*** are the ones that most closely match the format of the JSON and Object streams. Because of this it is the example that is shown. If you wish to handle XML more efficiently there are many other library options.

Like the JSON and Object streams the XML encoder and decoder classes have ***writeObject*** and ***readObject*** methods. These behave like the ones we have already seen except they read and write XML. Code Sample 6 shows the ***XMLDecoder*** in action.

```
File aFile = new File("test.xml");
try {
   FileInputStream aFileStream = new FileInputStream(aFile);
   XMLDecoder objectIn = new XMLDecoder(aFileStream);
   Person aPerson = (Person)objectIn.readObject();
   objectIn.close();
}
catch (Exception e) {
   e.printStackTrace();
}
```

Code Snippet 6

Like the *ObjectInputStream* it produces an *Object* that can be cast to the type that was written out. This is only true if the xml file was created using the *XMLEncoder* class. They should always be used together. XML written in other ways will probably fail to be read unless you are very careful.

The *XMLEncoder's writeObject* method will accept any type of Java instance, but this *writeObject* and the *XMLDecoder's readObject* method are intended to be used only with JavaBeans. Any other type of use can easily result in failure. Also, the encoder and decoder do not work with raw network socket streams.

For the examples in this book we'll use JSON streams.

Movie 4.1 An Analogy

Many Become One

CHAPTER 5

Parallel Processing

The power of current CPUs with multiple cores allows our applications to do more than one thing at a time. These applications are not able to do this unless we build in this ability. CPU speeds are not increasing dramatically. Core counts are. To make our applications run faster they must take advantage of multiple cores.

Parallel Options

Points to Ponder

1. Why would one type of threading option be preferable to another? Don't they all accomplish the same thing?
2. Why might I want to have multiple threads in a server?
3. Why might I want to have multiple threads in a client?

Each of these animals live and grow independently yet they share the same environment and space.

In Java parallel processing is done with threads. These threads are similar to, but not the same as, sub-processes within our application's process and memory space.

There are many ways to create and use these threads.

Executor, Thread or Runnable?

Java has three ways of accomplishing parallel processing. As with all options, which you choose depends on the situation. The oldest of these methodologies is to subclass the base java.lang.Thread class.

The newer java.lang.Runnable implementation tends to lead to a more modular design. Being more modular, applications using it tend to be easier to create and maintain in more complicated, meaningful, real life situations. Easier to create and maintain means faster time to completion and less time tracking down and fixing defects. Less time means less cost. Lest cost means a greater likelihood that the application will be completed and then accepted by potential customers.

java.util.concurrent.Executors are the newest way to create multi-threaded applications. They contain 'pools' of threads and use Runnables. If you find yourself repeatedly creating threads when some event like a user click happens, you may be wasting valuable CPU resources. A ThreadPool allows you to initially create a series of threads, use them as needed, and when you are done with the thread put it back in the pool so it can be reused later.

Subclassing Thread

Code sample SillyThread.java is an example of subclassing Thread.

Code Sample SillyThread.java source as file

```
package com.real.java.example;

public class SillyThread extends Thread{
   public void run(){
      for (int i = 0; i < 3; i++) {
         System.out.println("Thread id: "
               +Thread.currentThread().getName());
         try {
            Thread.currentThread().sleep(100);
         }
         catch (Exception e) {
            e.printStackTrace();
         }
      }
   }
   public static void main(String[] args){
      for (int threadCnt = 0; threadCnt < 3; threadCnt++) {
         SillyThread aSillyThread =  new SillyThread();
         aSillyThread.start();
      }
   }
}
```

Code Sample - SillyThread.java

If you choose to subclass Thread, your subclass must override Thread's ***public void run method***. The code in your version of the run method is what will execute when your subclass starts running.

To start a subclass of Thread running in parallel you must instantiate the subclass and call its ***public void start()*** method. You don't see this method in your code. Your subclass inherited it from the

Thread super class. You can see how to use ***start()*** in the ***main*** method of the SillyThread.java code sample.

Using Runnables with Threads

Code Sample SimpleRunnableStarter.java and SimpleRunnable.java contain code that does exactly the same thing as Code Sample 1 but using a custom *Runnable* instead of a custom Thread.

```
package com.real.java.example;

public class SimpleRunnableStarter{

   public static void main(String[] args){
      for (int threadCnt = 0; threadCnt < 3; threadCnt++) {
         SimpleRunnable aSimpleRunnable =
               new SimpleRunnable();
         Thread aSimpleThread = new Thread(aSimpleRunnable);
         aSimpleThread.start();
      }
   }
}
```

Code Sample - SimpleRunnableStarter.java

Code Sample SimpleRunnableStarter.java source as file

These samples contain the source code for two java files; one for the custom Runnable and one for an application that has a main method.

```
package com.real.java.example;

public class SimpleRunnable implements Runnable{

   public void run(){
      for (int i = 0; i < 3; i++) {
         System.out.println("Thread id: "
               +Thread.currentThread().getName());
         try {
            Thread.currentThread().sleep(100);
         }
         catch (Exception e) {
            e.printStackTrace();
         }
      }
   }
}
```

Code Sample - SimpleRunnable.java

Code Sample SimpleRunnable.java source as file

Back in Chapter 2 you learned about subclassing. In Chapter 3 you were first introduced to the keyword *implements* and what interfaces are. It's now time to face the music and get an understanding of what an interface is and how to use it. Don't freak out. An interface is just a very simple type of class. It is so simple that it has no attributes and it doesn't have things it does either. "Well." you say, "Then what good is it? Those were the two things classes could do." Ah...very good question and concern. Let's answer those.

An interface allows you to create a class that can only be used to do subclassing. When you do subclassing you can override methods like you see in the SillyThread.java sample code. There

Thread's *run* method was overridden. When you implement some sort of interface like Runnable, you HAVE TO override any methods it has. But how can it have methods if it doesn't do anything? Take a look at the Runnable.java code snippet.

```
package java.lang;

public interface Runnable {
   public void run();
}
```

Code Snippet - Runnable.java

There isn't much to it is there? The whole thing reads, "There is an interface called Runnable that is part of the java.lang package. Anything implementing it HAS TO override a *run* method that expects nothing to be passed to it and returns nothing when it is done." That's all there is to it. Essentially it has a method declaration, public void main(), but that method CAN'T have any code in it because it is an interface's method. That's different than normal class's behavior. That's why this code says "public interface Runnable" instead of "public class Runnable." Got it? Don't over think this. Like JavaBeans they are supposed to be very simple to create and use.

Using Executors, Thread Pools, and Runnables

The SimpleThreadPool.java code sample uses the same SillyRunnable class but in a different way. Here we create and use an *Executor* instance that contains a *CachedTreadPool*. This means that each time the *execute* method of the *Executor* instance is called, it runs a *SillyRunnable* in an available thread from the pool. A thread is available and in the pool if it is not currently running some other Runnable.

If the *Executor* instance finds that are no available threads it will create a new one and then use it to run the Runnable.

Since we used the *newCachedThreadPool* method, if a thread hasn't been used for 60 seconds, it will be removed from the pool and deleted.

```
package com.real.java.example;

import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

public class SimpleThreadPool {

   public static void main(String[] args) {
      Executor anExecutor = Executors.newCachedThreadPool();
      for(int threadCnt = 0; threadCnt < 3; threadCnt++){
         SimpleRunnable aSillyRunnable =
               new SimpleRunnable();
         anExecutor.execute(aSillyRunnable);
      }
      try {
         Thread.sleep(5000);
         System.out.println("Done Sleeping");
         for(int threadCnt = 0; threadCnt < 3; threadCnt++){
            SimpleRunnable aSimpleRunnable =
                  new SimpleRunnable();
            anExecutor.execute(aSimpleRunnable);
         }
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
}
```

Code Sample - SimpleThreadPool.java

[Code Sample SimpleThreadPool.java as file]

By dynamically adding and removing threads from the pool Executor strikes a delicate balance. If threads were added but never removed RAM would be wasted should the number of threads peak and then drop. If threads had to be created every time an event happened, CPU cycles would be wasted creating and destroying the threads. Use the Executor and its pool if there are a lot of times you need to use threads for short periods.

Android™ Specific Threading

All of these threading methods work in Android™ but may be problematic if at the conclusion of the thread the User Interface (UI) must be updated. Updates to the UI can not be made in a background worker thread. They must be made from the main, default, UI thread. It is also true that communication with a server should never be done from within the main UI thread. This is true of all modern languages that are event driven such as Objective-C, C++, and others.

The Android™ version of Java includes a method that can be called from a worker thread that causes code to execute on the UI thread. It has been found that this was too hard for most beginning programmers so they created a new class called *AsyncTask*.

AsyncTask is designed to be used as a parent class to a custom task you create for your application. It has three methods that you would override to implement your functionality.

- ***onPreExecute*** - This method will be run in the UI thread before a worker thread executes in the background.
- ***doInBackground*** - This method is run in the background on a thread.
- ***onPostExecute*** - This method is run in the UI thread after the worker thread completes execution of the ***doInBackground*** method.

Be aware that AsyncTask with its default behavior will not allow two background threads to run in parallel. To see how to change this and other methods available for overriding see the AsyncTask API.

AsyncTask also imposes design restrictions. To be used as its creators intended you should create your subclass of AsyncTask as an inner-class within the Android™ Activity which uses it. If your application is normally or highly complex other options are a better choice.

If this is the case, and since the UI cannot be updated from a background thread, what can be done? The Android™ API provides a way to resolve this problem. The android.os.Handler class is used to pass a Runnable to the main UI thread. This Runnable can then update the UI without issue. When used in conjunction with UI elements wrapped in standard Java WeakReferences, Android's Handler class allows the UI to be updated. Take a look at ***Chapter 6*** for more information about these classes and how to use them.

Applications Need Multiple Threads

A problem with the serialization code snippets in Chapter 4 is the *writeObject* and *readObject* JSONStream calls lock up the user interface until they complete. This is a BAD idea. If you attempt to do anything significant, your user will think your application has frozen. You've experienced how frustrating it can be when an app stops working for a little while and then 'magically' starts working again. You didn't like it. Don't do to others what you don't like done to yourself.

When doing any serious computation, it is much better to do it in a separate, background thread. We are getting ready to write code that will talk to a server. When doing so it is vital that all that communication be done on a separate threads.

Multiple Users == Multithreaded Server

All modern production servers are able to handle multiple clients at the same time. The Apache Web Server would be useless if it could only process one request from one client at a time. The same is true of Tomcat, PHP, ASP, JSP, Ruby, Python, Perl, or other servers. If a request had to wait for all other requests to complete, web servers would be useless. The time delay between a request and getting a response would be huge. Imagine if there were 200,000 people all connecting to your server. Nothing would ever get done because your users would give up. They'd go find someone else's code that was designed better.

All modern servers, regardless of the language the are written in, use parallel processing. In Java servers this means one thread per connected client. That thread can then handle the requests from the client while other threads handle requests from other clients.

CHAPTER 6

Clients and Servers

The mobile computing world we now live in requires that devices be able to communicate with each other and with other larger computing resources. This can be done in many ways. Currently the most common way uses the HTTP protocol. Let's gradually go from ideas all the way to Android™ speaking HTTP to a production quality server. Hold on to your hat. It's going to be a bumpy ride.

Sockets, Ports, and Streams

Points to Ponder

1. Why create a client and a server, isn't peer-to-peer better?
2. Why use sockets instead of writing to files?
3. Why does the 'three-way-handshake' exist?
4. Why can one server handle multiple clients if they all connect to the same port?

Clients and servers use streams similar to file streams to communicate over sockets and ports.

Let's get something straight right up front. A server isn't hardware or a virtual machine. It's a piece of software. Servers allow remote users to connect via a client. The users then can have the client make requests of the server. The client usually shows the user the result of their request. A client you are familiar with is your web browser. With these points understood, let's get to it.

Somehow people have begun to think that there is a major difference between how peer-to-peer and client-server applications work. You may have used both of these kinds of applications. You browser is the client you use to communi-

cate with HTTP servers. Chat and video conferencing are often implemented as peer-to-peer. Both of these types of applications have at their root the same basic parts--sockets and ports.

Peer-to-Peer or Client-Server?:

In client-server applications the server is used as the storage location for shared data. All clients communicate with this server and never communicate directly with another client.

Peer-to-peer applications seem to be different but are significantly the same. As mentioned earlier, chat applications are often written following a peer-to-peer design. You would think that there would be no server involved. Such thoughts are usually false. In fact your chat app most likely connects to a main server in order to find out who else is currently available. It then is able to connect directly to the peer chat application on the other end.

OK. So this isn't a big deal. After this initial connection there is no communication with servers... right? Wrong. A Peer-to-peer application on your device functions as both a client and a server. A client since it has to connect to the remote peer. And a server since it accepts connections from remote peers. So peer-to-peer applications are just specialized adaptations of the client-server design.

So which design, peer-to-peer or client-server, is the best to use? That depends completely on your preferences, your customers' needs, and the situation in which your application will be running.

Networking Software Options:

There are many options for network data communications with Android™ devices. These range from the *HttpURLConnection*, which will be used in this book, to the low-level, standard Java *Socket* class that the HttpURLConnection and all other options depend on. Each solution has its pluses and minuses. Each its advantages and disadvantages.

If your application needs to communicate with a web application or service, the natural choice when writing Java is to use the HttpURLConnection or its many competitors, some of which are also included in the Android™ SDK. If your application needs to speak directly to another Android™ device or traditional server software the standard Java Socket class may be the better choice, though this is rarely needed.

Regardless of which type of communication is being done, you need to understand how sockets work. If you do not you will make engineering and coding mistakes. By making these mistakes your app will consume too many network and computing resources. This means that your customers will not be satisfied and, given an option, will abandon your application.

A very common design mistake when creating client-server applications is to send many small pieces of data rather than fewer, larger ones. For a long time it has been true that if you send data across the network in chunks of approximately 2k in size or multiples there of, you will minimize waste. Thankfully the libraries used for networking in modern languages will handle this for you unless you force them to send smaller amounts. This mistake is usually made by 'flushing' the data on your own rather than letting the libraries flush when full. Don't over flush.

Ports:

When creating a client-server set of applications you must decide which port will be used for communication. This port must be known to both the client and the server. If it is not, then the client and server will not be able to work together. Which port or ports you choose is up to you. You must decide which one is most likely not in use by some other application. Another consideration is which ports are open on the network within and across which the application will run. Which port is best to use depends on your preferences and the specific installation situation. Choose wisely.

Port, since it is an English word, had a meaning long before computers ever appeared on the scene. They are places, be they for ships, airplanes, or other types of transportation, where items enter and leave. Those who developed the TCP/IP networking abilities of computers adopted this word because it describes exactly what a software port is in a computer. It is an identifiable location where data moves in and out of the computer.

These ports are not matched one-to-one with the physical locations where something might be physically connected to the machine. They are software and managed by the the operating system. That is why there are vastly many more ports available on a machine than the number of physical connections.

There are tens of thousands of ports available on modern computers. Many of these are reserved for specific purposes and some, such as the http port 80, are used for specific tasks by convention.

Java has no port class. Instead it uses integers to represent ports. This matches up well with how operating systems expose them as numeric values.

Sockets:

Sockets, on the other hand, are a standard type of class in Java. Software sockets, regardless of the computing language, do the same thing as other types of sockets. They wrap themselves around something. Socket wrenches wrap themselves around bolts. Electrical sockets wrap themselves around electrical plugs, etc., etc.

Software sockets wrap themselves around software ports. This is why code that you write creates and uses sockets. Your code does not communicate directly with ports. Software sockets have a great deal of functionality built into them so you don't have to re-invent it. Code Snippet 1 shows a socket that is part of a client attempting to connect to a server using port 9292.

```
Socket toServer = new Socket("10.0.2.2", 9292);
```

Code Snippet 1

Code Snippet 2 shows how to the server waits for incoming connections on port 9292.

```
ServerSocket aListeningSocket = new ServerSocket(9292);
Socket toClient = aListeningSocket.accept();
```

Code Snippet 2

These two snippets show code that reflects the result of the three way handshake managed by the TCP protocol. The port number used to start the communication is not the one that is finally used for data transfer. If it were, only one connection at a time could be made to the server. That would not be very useful. Which port is actually used as the client and server communicate with each is hidden from the programmer. We just don't need to know.

Don't worry about completely understanding the snippets in this section. You won't be writing this type of code. Just remember that the server you'll be using, Apache Tomcat™, already has this code in it. Someone already wrote it for you and they did it with threads like you learned about in Chapter 5.

Streams and Sockets:

In Chapter 4 you learned how to use streams to do IO with files. Streams are also used to send and receive data across a network. Again, the Tomcat server already has this kind of code inside it. You won't have to write this part of the server yourself. You do need to understand how it works. There are lots of examples of how to do this online and there are snippets here as well. Feel free to experiment with them.

Diagram 1 is a visual interpretation of how Sockets, ports, and streams interact with each other.

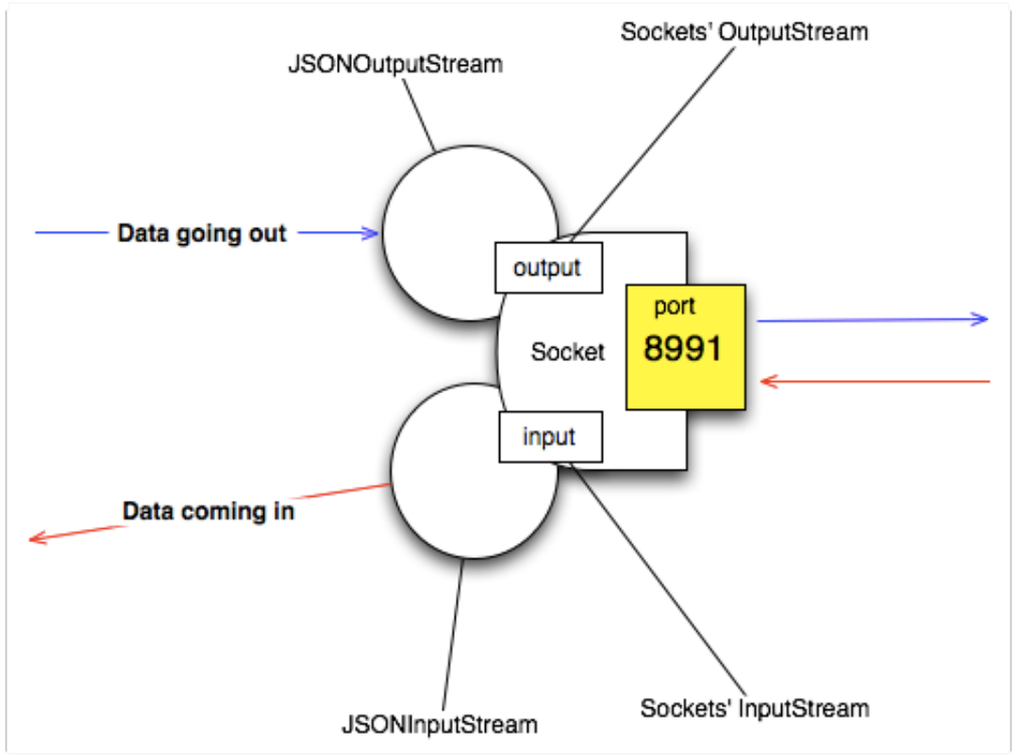


Diagram 1: Interactions of ports, sockets, and streams

Diagram 1 as image

After a Java Socket is instantiated, it contains within itself an InputStream and an OutputStream. Both of these streams are set up and ready to pull data in through a port as well as send data out. Rarely are these raw input and output streams used directly.

Instead, as seen in Diagram 1, they are usually wrapped with some other type of stream. Since JSON is such an easy way to communicate in an Android™ application, Diagram 1 shows the socket's raw streams being wrapped with a JSONInputStream and a JSONOutputStream as described in Chapter 4. The JSON streams

are then used by the application that created them to do all of the communication with the remote server.

Diagram 1 shows data being sent to the JSONOutputStream instance using its *writeObject* method. The data is passed by the JSONOutputStream instance to the Socket's *outputStream* which sends the data out through the port. Data is pulled into the application using the JSONInputStream instance's *readObject* method.

The *readObject* method reads data from the Sockets' input stream. The Socket's inputStream pulls the data through the port.

These relationships hold true for both the Client and the Server. Some example snippets may help you understand. In both the client and server snippets you use JSONInputStreams, JSONOutputStreams, Sockets, and ports. Code Snippet 3 shows JSON streams being wrapped around the input and output streams of the *toServer* socket shown in Code Snippet 1.

```
Socket toServer = new Socket("10.0.2.2", 9292);
//setup the JSON streams.
JSONInputStream inFromServer =
                                new JSONInputStream(toServer.getInputStream());
JSONOutputStream outToServer =
                                new JSONOutputStream(toServer.getOutputStream());
```

Code Snippet 3

Code Snippet 4 shows the corresponding JSON streams being wrapped around the socket in the server application.

```
Socket clientSocket = aListeningSocket.accept();
//setup the JSON streams
JSONInputStream inFromClient =
                                new JSONInputStream(clientSocket.getInputStream());
JSONOutputStream outToClient =
                                new JSONOutputStream(clientSocket.getOutputStream());
```

Code Snippet 4

While you won't be instantiating Sockets and ServerSockets in your code, I've put together a Java socket, port, and stream pair of examples you will see in the next section of this book. HttpURLConnection and Tomcat both give you access to input and output stream around which you will need to wrap JSON streams. That's why you won't have to write the socket code yourself. The next section of the book is for background understanding purposes. You will rarely need to write this kind of Java code, but if you don't understand it, you won't understand what HttpURLConnection and Tomcat are doing "under the hood" and how data is sent back and forth. I strongly suggest getting the code from Section 2 of this chapter compiled and working so you can see JSON streams in action across a network.

A Client-Server Pair

For many flowering plants a separate pollinator plant is required for the most robust offspring. Most robust software also comes in two parts--a client for user interactions and a remote server to store and serve up data.

Many examples of Java clients and servers are found on the web that use BufferedReaders and PrintWriters to send strings back and forth. These examples are usually of poor quality since no one really uses BufferedReaders and PrintWriters for significant client/server communication. Both of those classes are too limited in their ability and force the programmer to create a great deal of custom string parsing. A simple client-server example using JSONInputStreams and JSONOutputStreams is called for.

Heads up! This chapter uses a bunch of stuff you learned in previous chapters. I'll point them out to you when you get to them but please go back and refresh what you learned previously when you start to get confused.

There are two main types of client-server pairs. The first is called *connect many use once*. A web server and your browser are this type of pair. The way this type of client-server pairs works is they con-

nect each time you ask the client to do something. They then do what you asked, and then disconnect. Each time you click a link in your browser, your browser opens a socked on port 80 and connects to the web server. It then sends a request to the server, waits for a response and closes the socket. HttpURLConnection and Tomcat use this *connect many use once* approach.

There are advantages and disadvantages to writing your code to be *connect may use once*. When using this approach you use CPU cycles to create a Socket each time data needs to be sent or retrieved from the server. This causes the CPU use to grow unreasonably if a single client is connecting to the server a lot. Thankfully, no matter how fast you click in your browser you will not be able to connect a lot in a short amount of time. Humans like you are just too slow. If you were writing code or scripts to automate connections, then you might run into trouble. Computers are much faster than we are.

The advantage to using *connect many use once* is when the request and response are completed, the Socket is closed and the port it was using is freed. Since ports are a limited resource on the server, closing them as soon as possible frees them up for other clients to connect and use. The threads in the server's thread pool are also freed up for reuse on the server when it is written to be *connect many use once*.

The other approach to creating client-server pairs is *connect once use many*. An example of this type of client-server pair that you may have used is the Oracle[tm] command line interface (CLI). For this type of client, you log in once and then make many requests. After completing all of your requests you log out. When you use Oracle's CLI and login, a socket connection is made to the server. This socket, and its associated port, are then used for all the request you make. The socket isn't closed and the port freed up for reuse until you logout.

Connect once use many's advantages and weaknesses are the inverse of those for *connect many use once*. Since your are only creating a socket once, CPU resources to do so are only used that one time. A disadvantage is it can be easy to use up all of the server's available ports and threads in its thread pool if a lot of clients need to connect simultaneously.

"But wait a minute." You say. "I login to servers, make many requests, and then log out of servers all the time using my web browser. I thought you said browsers and web servers were *connect once use many*!!" Yep. I did. They are. Those web sites are written to "fake up" being *connect once use many*. They do it using session cookies to track who you are and if you are logged in. Session cookies are outside the scope of this book. If you need more information on them there's a lot of information on the web.

The Client:

The ConnectManyClient.java code sample uses things you learned in Chapter 4 and in Section 1 of this chapter. If you start to get confused take a look back there to get a refresher. ConnectManyClient is also an implementation of the *connect many use once* approach. If you take a good look at it you will find an infinite while loop. Each time the code goes around the loop it connects to the server using a socket, sets up the JSON streams, sends a request using the *writeObject* method, gets the server's response using the *readObject* method, and then allows the socket to auto-close when it goes out

of scope (when the code starts the loop over again). The ConnectManyClient connects many times and uses each connection only once.

```
public class ConnectManyClient {
  public static void main(String[] args) {
    ConnectManyClient theClient = new ConnectManyClient();
    theClient.go();
  }
  private void go() {
    try {
      while(true) {
        Scanner systemInScanner = new Scanner(System.in);
        System.out.printf("Enter the message to send to the server.\n");
        String messageForServer = systemInScanner.nextLine();
        //connect to the server
        Socket toServer = new Socket("localhost", 9393);
        JSONInputStream inFromServer = new
                                        JSONInputStream(toServer.getInputStream());
        JSONOutputStream outToServer = new
                                    JSONOutputStream(toServer.getOutputStream());
        HashMap<String, Object> request = new HashMap<>();
        request.put("command", "Speak");
        request.put("message", messageForServer);
        outToServer.writeObject(request);
        System.out.println("waiting for response");
        HashMap<String, Object> response =
                                (HashMap<String, Object>) inFromServer.readObject();
        if (response.get("command").equals("Done")) {
          System.out.println("Sent request: " + request +
                                                  "and  got response  " + response);
        } else {
          System.out.println("Oops. got " + response);
        }
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Code Sample - ConnectManyClient.java

Sample ConnectManyClient.java source as file

The Server:

The ConnectManyServer.java code sample is in some ways the inverse of its client. It also has an infinite while loop but instead of writing and then reading using JSON streams, it reads then writes. This inversion is required since the client writes then reads. Stop and think about this for a while. It takes some pondering.

The ConnectManyServer, unlike the client, is threaded so it can handle connections and requests from multiple clients at the same time. It uses the Executor class as a thread pool that you learned about in Chapter 4. If the Executor is confusing you, go back and refresh.

If you take another good look at this code you'll see that each time a connection is accepted a thread in the thread pool is used to execute an anonymous Runnable. It is the Runnable that does all of the communication.

While the Runnable is busy, the server goes back around the loop and waits for another connection. Make sure you understand that the server NEVER exits unless there is some horrible problem and it is not able to start up the ServerSocket. It would be really bad if a server you were responsible for exited after each request or if something minor went wrong. You'd have to restart it ALL the time. That sounds like a boring job.

```
public class ConnectManyServer {
   private Executor theExecutor = Executors.newCachedThreadPool();

   public static void main(String[] args) {
      ConnectManyServer theServer = new ConnectManyServer();
      theServer.start();
   }
   private void start(){
      try {
         ServerSocket aListeningSocket = new ServerSocket(9393);
         while(true){
            System.out.println("Waiting for client connection request.");
           final  Socket clientSocket = aListeningSocket.accept();
           this.theExecutor.execute(new Runnable() {
              @Override
              public void run() {
                 try {
                    JSONInputStream inFromClient = new
JSONInputStream(clientSocket.getInputStream());
                    JSONOutputStream outToClient = new
JSONOutputStream(clientSocket.getOutputStream());
                    System.out.println("Waiting for a message from the client.");
                    HashMap aRequest = (HashMap) inFromClient.readObject();
                    System.out.println("Just got:" + aRequest + " from client");
                    aRequest.put("command", "Done");
                    outToClient.writeObject(aRequest);
                 } catch (Exception e) {
                    e.printStackTrace();
                 }
              }
           });
         }
      } catch (Exception e) {
         e.printStackTrace();
         System.exit(1);
      }
   }
}
```

Code Sample - ConnectManyServer.java

Sample ConnectManyServer.java source as file

Model-View-Control:

If you've spent any time in the software business, taken programming classes, or tried to learn to create apps using online information you've run across the Model-View-Control (MVC) pattern. Everybody has a different idea of exactly what MVC is and what code falls into which part. Is it Model, View, or Control code??? Ask that question in a design meeting and you almost always get multiple answers from experienced software engineers. That's because the answer depends on opinion, not fact. So in this book I'm going to suggest a definition for us to use.

Let's define Model first. It seems to be the easiest for people to agree on. How about "The model is the data, a representation of the data, and code that hides how the data is accessed and stored."

The definition of View code we'll use is "Any code that is used to communicate with the user OR ANY OTHER COMPUTING SYSTEM requesting or sending data (this is IO code not part of the model)." Lots of people new to software development seem to get the idea that View code only consists of what the user sees. OK. What is a user? Does a user have to be human? Nope.

That leaves us needing a definition for Control. Here is one I like. "Control code is the *smarts* of the application. If a decision is being made, it's Control code. If data is being manipulated, stats calculated, strings assembled, etc., its Control code."

If we use these definitions, which we will, the code in the ConnectManyServer code sample is View code. What?? How could that be??? Well...the client is the user and the server sample code is only doing IO, Input-Output and according to our definition that makes the code in the example View code.

The rest of this chapter focuses on View code. Chapter 7 deals with Control code and Chapter 8 with Model code. We'll gradually put it all together.

Now....if you understand how the client and server pair in this section work together, you're ready to see how to use the *connect many use once* Tomcat server and HTTPURLConnection client pair.

Tomcat and Java HTTP

Apache Tomcat™ is a robust, commonly used web application server.

With a basic understanding of the way servers accept connections and do threading gained from the previous section of this chapter, let's use a server others have already built. Apache Tomcat™ manages all the connections, thread pools, and Runnables for us. It also parses all of the HTTP headers, body, and content so we don't have to write all that code. Thank goodness! HTTP parsing would be way too much to cover in a book like this and you probably don't want to know how to do it anyway.

Instead, this section will introduce you to a new class, part of Enterprise Edition of Java (Java EE) used by Tomcat and other enterprise ready Java servers. The client example in this section is a regular Java class. Be patient. Spend some time with this section and it will be much easier to figure out what's going on in the next section, connecting to and running the server code using an Android™ client.

Basic Tomcat:

If you haven't downloaded and installed Tomcat. It's time do do it now. There are many tutorials and videos on the web. Follow them. Oh, and make sure you have the latest version of Tomcat.

Since others have done most of the hard work for us when they wrote Tomcat, what we really need to know is how to insert our code into that existing server. You do this using things called **servlets**.

Servlets are little pieces of code you write that run inside the Tomcat server. Each time a request is made, Tomcat is responsible for finding the appropriate servlet to execute based on the URL it is sent. It also parses all the rest of the HTTP information it gets from the browser into ArrayLists, HashMaps, and builds any other appropriate objects to make it easier for your code to focus on handling what ever request was made. Just like a server at a good restaurant takes care of all kinds of things for you so you can focus on enjoying your meal, Tomcat does all of this for you.

Part of the HTTP information parsed by Tomcat is the request type. If you've done much HTML or JavaScript coding you probably are aware that there are different types of requests that can be made. The two you are most familiar with are GET and POST requests. POST requests are supposed to be used to handle form submission and GET requests are used to request information. Going to other pages as a result of selecting links would be an example. There are other types of requests (PUT, DELETE, etc.) but GET and POST is all we'll need for the purposes of this book.

The HTTPServlet Class:

In order to create a servlet that can run inside of Tomcat, you must subclass the standard Java EE HttpServlet class. This book will focus on three of the HttpServlet class's methods. There are many others but let's save those for later.

HttpServlet's *doGet* and *doPost* methods are called for you depending on if the HTTP request is GET or POST. One of these is called each time the client requests or sends data. The other method we're interested in is *init*. It doesn't get called each time the client makes a request. It is called the first time the servlet needs to be used. This means you can put code there you only want to have executed once. We'll take advantage of that in Chapter 7. But for now, let's focus on *doGet* and *doPost*.

The Servlet:

The JSONEchoServlet.java code sample has exactly the same behavior as the server in the previous section so the code should be very familiar to you. The same JSONStreams are used to read and write the same data as in that example. One difference between the previous sample and the JSONEchoServlet sample is how to get ahold of the input and output streams used by the JSON streams.

```
@WebServlet(name = "JSONEchoService", urlPatterns = {"/json"})
public class JSONEchoServlet extends HttpServlet {
   protected void doPost(HttpServletRequest request,
         HttpServletResponse response) throws ServletException, IOException {
      JSONInputStream inFromClient =
                              new JSONInputStream(request.getInputStream());
      JSONOutputStream outToClient =
                              new JSONOutputStream(response.getOutputStream());
      try{
         System.out.println("Waiting for a message from the client.");
         HashMap aRequest = (HashMap) inFromClient.readObject();
         System.out.println("Just got:" + aRequest + " from client");
         aRequest.put("command", "Done");
         outToClient.writeObject(aRequest);
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
   protected void doGet(HttpServletRequest request,
         HttpServletResponse response) throws ServletException, IOException {
      this.doPost(request,response);
   }
}
```

Code Sample - JSONEchoServlet.java

Sample JSONEchoServlet.java source as file

As you can see, Tomcat has created them for us. It added the input stream to an instance of HttpServletRequest class called *request* and the output stream to an instance of the HTTPServletResponse class called *response*. Wow. No more dealing with Sockets. The way this works is, some client has made a request so Tomcat gives you a request. And since you'll probably need to respond to that request, Tomcat gives you a response. That makes your code much simpler. Especially since Tomcat also handles all of the looping, threads, and thread pools for you. Thank you Tomcat developers and designers!

The only other major difference is your code is being executed as part of the *doPost* method instead of the start method you wrote for that example. All in all, you've written less code in this sample to get the same behavior you had in the other sample. Less code is good code, so we are headed in the right direction.

"But wait a minute. What's that @WebServlet stuff?" Hmmm...you noticed that did you? That is called an annotation. Annotations were added to Java quite some time ago. They let you decorate an instance of a class with extra information. Way back in the old days, in order to get Tomcat to know which servlet to run for a specific URL, you would have had to edit an XML file. That's not too much fun and it can make it hard to fix bugs. You would have had to make changes in two files instead of one. Because of that, the engineers in charge of Java EE created annotations for you to use instead of the XML file.

@WebServlet is one of the annotations they created. Since all annotations always apply to the next line of regular Java code, the annotation line of code reads, "Associate the name 'JSONEchoService' and the urlPattern '/json' with the JSONEchoServlet." By including this annotation, any time a client uses a URL that looks something like 'http://localhost/json,' Tomcat will do its thing, create the request and response instances, and send them to the JSONEchoServlet's *doGet* or *doPost* method and off you go.

The Client:

Since you are using HTTP to talk with the JSONEchoServlet sample, the Socket code you learned about before won't work. It doesn't speak HTTP. You'll need to use an instance of a class that does. The standard HttpURLConnection class speaks HTTP so let's use that one.

The JSONEchoClient is nearly the same as the ConnectManyClient. I've underlined the differences for you in the JSONEchoClient.java code sample. Like the JSONEchoServlet.java sample, the input and output streams aren't coming directly from an instance of the Socket class. The HttpURLConnection handles the creation of the Socket instance for you. It also has *getInputStream* and *getOutputStream* methods that will get you these from the socket it created. So in that way, HttpURLConnection works like Tomcat. However, it won't create threads and thread pools for you. Be sure to remember this. In the next section you'll need to create threads and pools yourself.

One thing to be careful about. Notice that JSONEchoClient doesn't create an instance of JSONInputStream until AFTER *writeObject* has been used to send data to the server. This is a limitation of HttpURLConnection. If you try to use HttpURLConnection's *getInputStream* method before you send data using it's output stream your code will fail. Just a head's up. Be careful.

```
public class JSONEchoClient {
  public static void main(String[] args){
    JSONEchoClient theClient = new JSONEchoClient();
    theClient.go();
  }
  private void go() {
    while(true){
      try {
        Scanner systemInScanner = new Scanner(System.in);
        System.out.printf("Enter the message to send to the server.\n");
        String messageForServer = systemInScanner.nextLine();

        URL url = new URL("http://localhost:8080/json");
        HttpURLConnection connection =
                                (HttpURLConnection) url.openConnection();
        connection.setDoOutput(true);//allows POST
        JSONOutputStream outToServer =
                    new JSONOutputStream(connection.getOutputStream());

        HashMap<String, Object> request = new HashMap<>();
        request.put("command", "Speak");
        request.put("message", messageForServer);
        outToServer.writeObject(request);
        JSONInputStream inFromServer =
                              new JSONInputStream(connection.getInputStream());
        HashMap<String, Object> response =
                              (HashMap<String, Object>) inFromServer.readObject();
        if (response.get("command").equals("Done")) {
          System.out.println("Sent request: " + request + "and  got response  "
                                                                  + response);
        } else {
          System.out.println("Oops. got " + response);
        }
      }
      catch (Exception e){
        e.printStackTrace();
      }
    }
```

Code Sample - JSONEchoClient.java

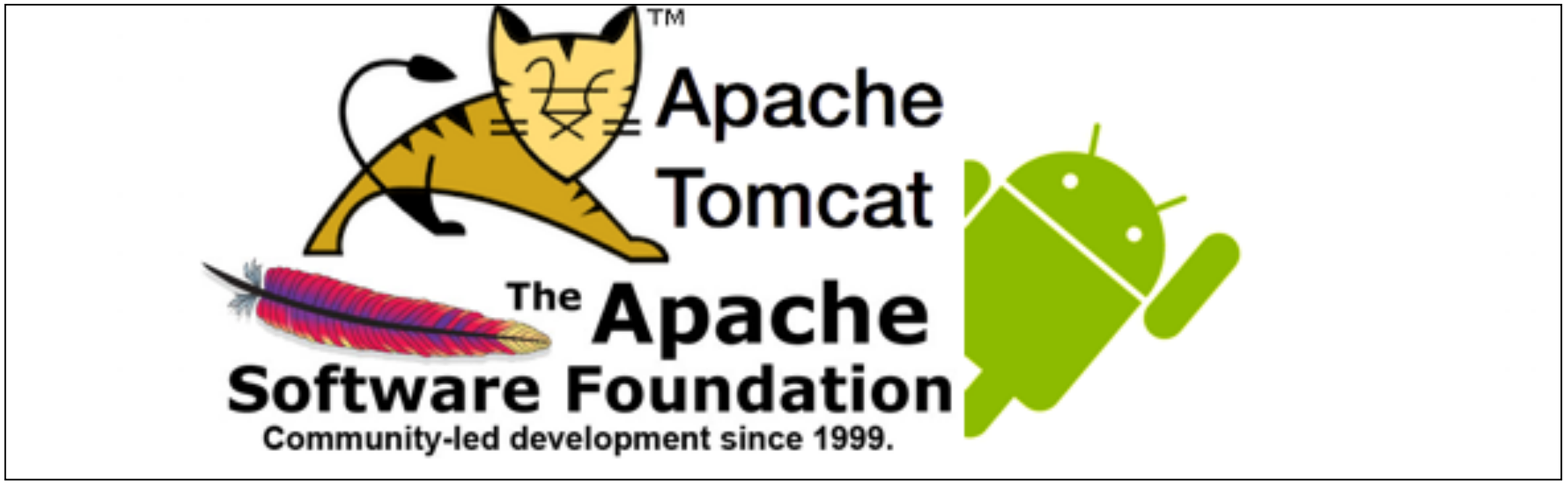
Sample JSONEchoClient.java source as file

Spend some time with this code. Get it working. Play with it. Make changes. Stepwise debug it using your IDE. Stop and spend the time. I will be well worth the effort.

Before reading this book any further spend the time examining this code. ***STOP HERE. PLEASE.*** Don't make your life harder than it has to be.

Now that we've got a beginning understanding of servlets and HttpURLConnection lets build on that. Let's make an Android client that uses and instance of HttpURLConnection to talk with a JSONEchoServlet.

Tomcat and Android HTTP



Apache Tomcat™ works very well with an Android HTTP client.

Android is a window based operating system. No...I did not say Windows™ based. I said window based. Windows™, OS X™, iOS™, and Linux™ are all window based operating systems. When you write code for modern window based systems you should always use a background thread to do any serious computation. If you don't, your app will temporarily lock up until your code completes. You've experienced this in applications before. You didn't enjoy it. Don't do to your apps users what you don't like being done to you. Hang on. This is going to get a little messy. But what ever you do....Don't panic. Take some time. Play with what you see and learn here. Talk with someone else about it. It takes a while to sink in.

In Android and these other window based systems, only the main thread, known as the User Interface (UI) thread, can update the user interface. This keeps the UI from getting in situations where

it is being modified in two different ways at the same time. What would your poor, stupid computer do if a UI label was being told to display 'hello' by one thread, and at the SAME TIME was being told to display 'goodbye' by another? To avoid this type of problem and the nasty code you would have to write to resolve these types of conflicts, the creators of these systems and the languages you use to write apps for them have a restriction. Your code can only update the UI from the UI thread.

"Uh Oh." You say. "Back in Section 3 the client was making server requests in the UI thread! Server requests are notoriously slow. Isn't that bad??" Yep, it is. Now you'll be putting the making of the request and handling the response in a background thread.

"But if the information in the response needs to be displayed in the UI and I can't do that from a background thread, what am I to do?" Good question. Android has a way for you to pull this off.

Android UI Elements and Queues:

In Java and in some other languages, threads are associated with a queue. A queue is similar to an ArrayList, but enforces First In-First Out (FIFO) behavior. That's a mouth full, but a simple example should clear up what it means. If you were going to the movies in the United States and a bunch of other people showed up too, you would get in a line and wait your turn. The first person that got in line would be helped first, then the second, then the third, until they finally got to you. This is FIFO behavior. If you were in England you wouldn't get in a line. You'd get in a queue. Line in the United States and queue in England mean the same thing.

Every Java Thread has a Queue associated with it. The Queue has a series of things to do; they're Runnables. The Thread is supposed to accomplish each of them when it gets a chance. You've experienced the end result of queue's being associated with threads before. Remember the last time you clicked really fast on something and the application, yes your operating system is an application, couldn't keep up but it eventually processed all your clicks? What happened was the UI element you clicked ended up putting some code to execute in the UI thread's queue for later processing.

In Android, each instance of a View (all Android UI elements are subclasses of View) remembers the queue of the Thread in which it was created. What you want to do, is to pass the UI element you want updated to the background thread's Runnable. That way your Runnable can use it to tell the UI Thread to update the element. "Great! Let's do it!" Hold on a minute. There are still some things you don't know about Android.

Android, Threads,and Memory Leaks:

If you wanted to update a label in your UI to say 'hello' from a background thread, you are going to have to pass a reference to the label to your Runnable. If you don't you can't write the update code because the code you will write won't have a label to update. It's as plain as that. No label, no update.

But Java is a reference counting system. This means that every time nothing refers to an instance, it gets garbage collected and eventually deleted. So, if you pass something to a method or constructor, the reference count for what ever you passed goes up by 1 meaning it can't be garbage collected yet.

So? Who cares? It will all work out in the end right? No. Not even close.

In Android, each time you are using an app and someone calls you on the phone, the app you were using is serialized. It does this so when you are done using the phone the app can presented to you and put you right back where you were. Once it is serialized the user interface elements and the rest of the parts of the app are garbage collected. When you are done with your phone call, a brand new bunch of instances are created when the serialized version of the app you were using is started up. The app DOESN'T just go to sleep. It is dead and then reborn.

So what is the problem then? If you pass a reference to ANYTHING in the UI to a runnable, the UI will never be garbage collected because the reference count to a part of it will never be zero. Even though it isn't garbage collected, a complete, brand new UI will be made and used when you start using the app again. The app just leaked all of the UI's memory and you won't get it back until you kill, not close, the app. If this happens a couple of times, the Android™ operating system will kill your app while the user is using it. It will 'crash.' Not good.

But now you're stuck again. You have to pass a label reference to a the background thread's Runnable so you can update what it says. But you can't pass a reference without creating memory leaks that cause your app to crash. It look like there's no way to update the UI. The solution to this conundrum is Java's WeakReference class.

Java Weak References:

Java WeakReferences are designed to get around this kind of problem. They are a strange class. When you send something to the constructor for a WeakReference, Java doesn't increase the reference count of the thing you sent. The WeakReference instance will attempt to remember the thing sent, but if the thing gets garbage collected, the WeakReference will forget it. Kind of nasty, but, oh well. It's what's available to work with. Other languages don't use this complicated of an API to accomplish updating the UI. This is an Android™ problem.

An example may help. If you haven't installed Android StudioTM and got a default app running in GenyMotion™, do it now before you go on. ***STOP HERE. PLEASE.***

Now that you've got at least the default Android™ app compiling and running, let's move on.

Code Snippet 1 is the sendMessage method of the MainActivity.java sample. It is a custom method created for this sample that is triggered each time a button in the sample's UI is clicked.

```
public void sendMessage(View v) {
   this.requestCount++;
   EditText textInput = (EditText)this.findViewById(R.id.message);
   String theMessage = textInput.getText().toString();
   TextView responseView = (TextView)this.findViewById(R.id.response);

   WeakReference responseViewReference = new WeakReference(responseView);
   ServerCommunicationRunnable communicationRunnable =
         new ServerCommunicationRunnable(this.requestCount,theMessage,
                                         responseViewReference);
   Thread serverCommunicationThread = new Thread(communicationRunnable);
   serverCommunicationThread.start();
}
```

Code Snippet 1

Sample MainActivity.java source as file

If you look near the beginning of the snippet, you'll see how to get a reference to the *response* multi-line text display, an instance of Android's TextView class. It is placed in the *responseView* reference. In the line of code right before this, an instance of Handler is instantiated and placed in the uiThreadStack reference. uiThreadStack is being made in the UI Thread and is going to be used later in the run method of a Runnable. Feel free to refer back to the Handler discussion as the sample is described.

The Runnable:

Code Snippet 1 uses a class that implements Runnable called *ServerCommunicationRunnable* (If you need to refresh you understanding of using Runnables, what they are, and how to use them, take a look at Chapter 5). ServerCommunicationRunnable is a little different than what you saw in Chapter 5. It has a constructor (see Chapter 2 Section 1 for more information on constructors) that is used to remember the message, the number of messages sent, a WeakReference to the multi-line text view, and a Handler that represents the stack of the UI thread.

```
public class ServerCommunicationRunnable implements Runnable {

   private int numMessagesSent;
   private String messageForServer;
   private WeakReference responseViewReference;

   public ServerCommunicationRunnable(int currentNumMessagesSent,
                                      String aMessageForServer,
                                      WeakReference aResponseViewReference) {
      this.numMessagesSent = currentNumMessagesSent;
      this.messageForServer = aMessageForServer;
      this.responseViewReference = aResponseViewReference;
   }
```

Code Snippet 2

Sample ServerCommunicationRunnable.java source as file

ServerCommunicationRunnable's attributes are going to be used in its run method and are in code snippets 3 and 4. The code here does almost exactly the same thing as the code in the JSONEchoClient in the previous section of this chapter. It puts together a HashMap containing a command and a message. It writes that HashMap to the outgoing stream and then reads a response in from the server. That is all the same.

The difference is at the end of Code Snippet 3. There the WeakReference and the Handler are used to update the user interface. If you are going to be writing Android™ apps and write them well, you will need to be very familiar with using WeakReferences and Handlers like this. Go over the code with someone else. Explain it to them. It will help you understand what's happening.

```
public void run() {
   try {
      URL url = new URL("http://10.0.2.2:8080/json");
      HttpURLConnection connection =
                                    (HttpURLConnection) url.openConnection();
      connection.setDoOutput(true);//allows POST
      JSONOutputStream outToServer =
                        new JSONOutputStream(connection.getOutputStream());

      HashMap<String, Object> request = new HashMap<>();
      request.put("command", "Speak");
      request.put("message", messageForServer);

      outToServer.writeObject(request);

      JSONInputStream inFromServer =
                                 new JSONInputStream(connection.getInputStream());
      HashMap<String, Object> response =
                                 (HashMap<String, Object>) inFromServer.readObject();
      final TextView responseView = (TextView) responseViewReference.get();
      responseView.post(new Runnable() {
         @Override
         public void run() {
            if (responseView != null) {
               responseView.setText("Sent " +
                        ServerCommunicationRunnable.this.messageForServer" +
                        ". Message number " +
                         ServerCommunicationRunnable.this.numMessagesSent);
            }
         }
      });
```

Code Snippet 3

Code Snippet 4 uses the same WeakReference to show the user an error message if something goes wrong. If this were a shipping app instead of a development example, you would never append the

error's localized message to what you tell your user. They won't understand it anyway. Your error messages to them should be helpful. They should always suggest ways they can resolve the issue, what ever it might be.

```
catch (Exception e) {
   e.printStackTrace();
   final String theErrorMessage = e.getLocalizedMessage();
   final TextView responseView = (TextView) responseViewReference.get();
   responseView.post(new Runnable() {
      @Override
      public void run() {
         if (responseView != null) {
            responseView.setText("Error: Unable to communicate with server. "
                                                                  + theErrorMessage);
         }
      }
   });
}
```

Code Snippet 4

This Android code is complicated but it is a good way to design and write your code. Do not give in to temptation and use Android's AsynchTask class. It can cause all kinds of bad code designs. It looks like it will make your life simpler, but if you are doing anything significant, it will make your life miserable. Stick with Threads, Runnables, and WeakReferences. You'll be happier in the long run.

In Chapter's 7 and 8, you'll end up modifying this Android™ code and the servlet code from Section 3 to be a full blown app. The user interface will be nothing to brag about, but you will be able to have your server efficiently process multiple kinds of requests. Even those that store data in a database and get data back out so you can display it to the user. In other words, with the understanding you gained perusing Chapter's 1 - 6 you will be ready to make the next major leap.

CHAPTER 7

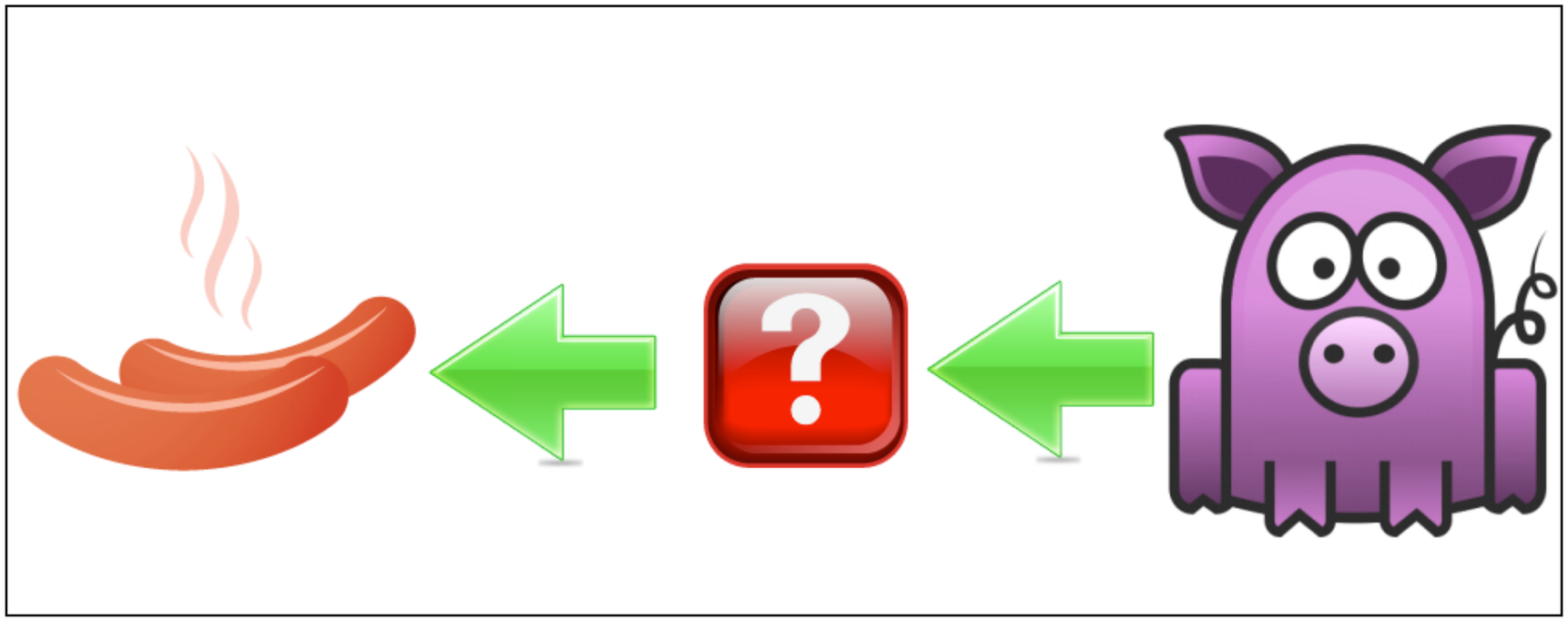
Small, Efficient, Flexible

For most software designs the more flexible the design the less efficient it is computationally. A case in point is Ruby on Rails; highly flexible and very inefficient with regard to CPU use. Another truism is that the smaller the code base, the less flexible it is. Both of these truisms are false. Code can be very small and highly efficient. It just takes a little abstract thinking.

Application Controller Pattern

Points to Ponder

1. Why might the false flexibility/efficiency truism have become believed?
2. Why is abstract thought important for solving true problems? (True problems are questions for which there is no known answer.)
3. Why would you ever create your own Interface instead of using a prebuilt one?



Command: make sausage. Input: pigs. Result: sausage. How did it happen? You don't want to know.

It is better, as a human being, to learn from the experience of others than to learn only from our own experience. We do this all the time when we search for resources on the internet. Often when we are looking for help but we don't want specific code. What we need is an approach, or way, to help us to design a specific solution.

Learning from others:

In a work environment we might go to a more experienced engineer or developer to see if they have done something similar to what we are attempting. If they have, then we harvest their experience for our own use and gradually become

an expert ourselves. This harvesting of knowledge has been done since before recorded history.

Books, and now internet resources, are an efficient way to store and distribute this type of information. The question has changed from 'how can I get information' to 'how do I know if the information is good or bad'? With books this was the role of content editors. With the internet there is no independent validator of the information. Because of this, bad and even malicious information can spread quickly and be widely adopted.

There has been some effort in the software engineering field to arbitrate and test software design approaches. The result is that some commonly held opinions and design approaches have been shown to be true and good and others false and bad. One of the end results of this effort is a set of patterns for good software design and anti-patterns showing bad software design.

Large If-Else Statements:

Large if-else or switch statements are the bane of major applications yet small statements are a joy. Invariably as the complexity of these types of statements grow, dependencies in the order of the checks occurs. In spite of this, required order efficiency requirements might indicate that the checks be in a different order.

For example, you may want to do a string comparison last and some integer comparisons first in a series of if-else statements since string comparisons use many more computing resources than integer comparisons. Yet you may have to do the string comparison first so that your code doesn't falsely pass one of the integer checks. Now you have a set of conflicting goals.

In this sort of a situation you need good engineering and a good design to solve your problem. The application controller pattern is a design used to solve this comparison problem. It is not without cost. There is the possibility that it could use more resources than an if-else statement in the same situation. You may need to do some runtime checks to see which solution would work better in for your need.

The Application Controller Pattern:

Diagram 1 is an a UML class diagram for a naive application controller pattern implementation. While method and class names are shown, they are not actually defined parts of the pattern and can be modified to be anything you would like.

The diagram shows that some class in your application will call the *handleRequest* method of the *ApplicationController* passing it a command, usually a string, and some optional parameters. The relationship between your application class and the ApplicationController is defined as *delegates* since once the call has been made your application will do no more independent computation. It has delegated the handling of the situation to the ApplicationController.

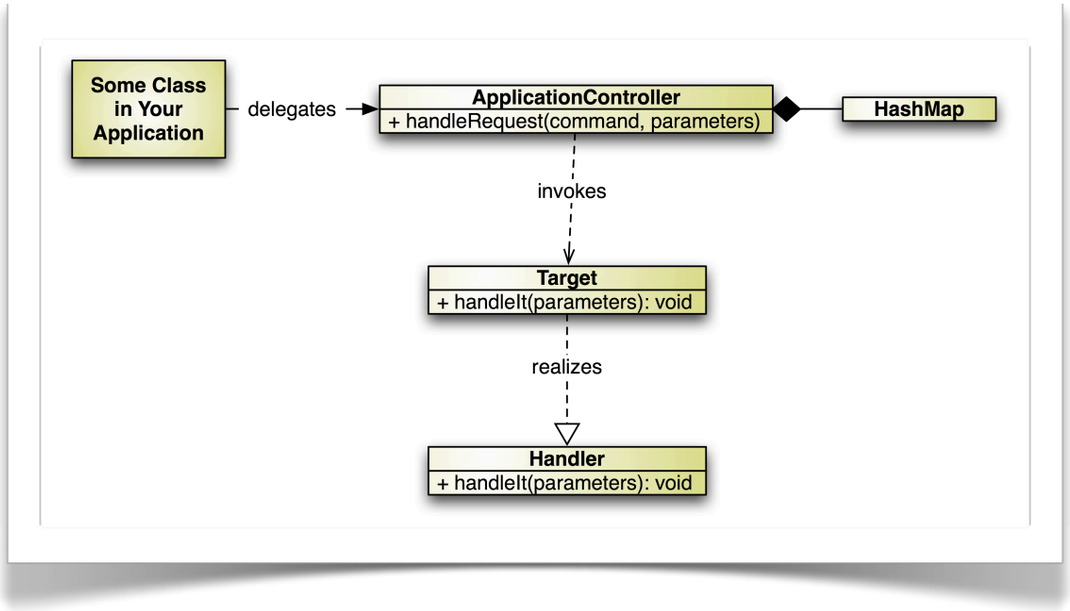


Diagram 1: the application controller pattern as a class diagram

Diagram 1 as image

The *Target* class in the diagram is a placeholder for any number of classes that implement, realize in UML speak, the *Handler* interface. This interface is one of your own creation. It is not included in any release of Java. Since all of the targets you create will implement this interface they will each have a unique implementation of a *handleIt* method.

The *HashMap* contains a series of keys and values. The keys are command strings and the values are instances of the various target classes. Each of the target classes has, in its *handleIt* method, all of the code to handle the situation indicated by the command. Each command represents only one situation that needs to be handled.

A design such as this is highly modular and makes the addition of new behavior and the support of existing behavior much easier than many other designs. This design also solves the if-else problem described earlier.

Since a user logging in is a commonly understood problem let's use it as an example. Diagram 2 is a UML sequence diagram illustrating using the Application controller pattern to handle this situation. The diagram assumes that the application is already running but the user has not logged in.

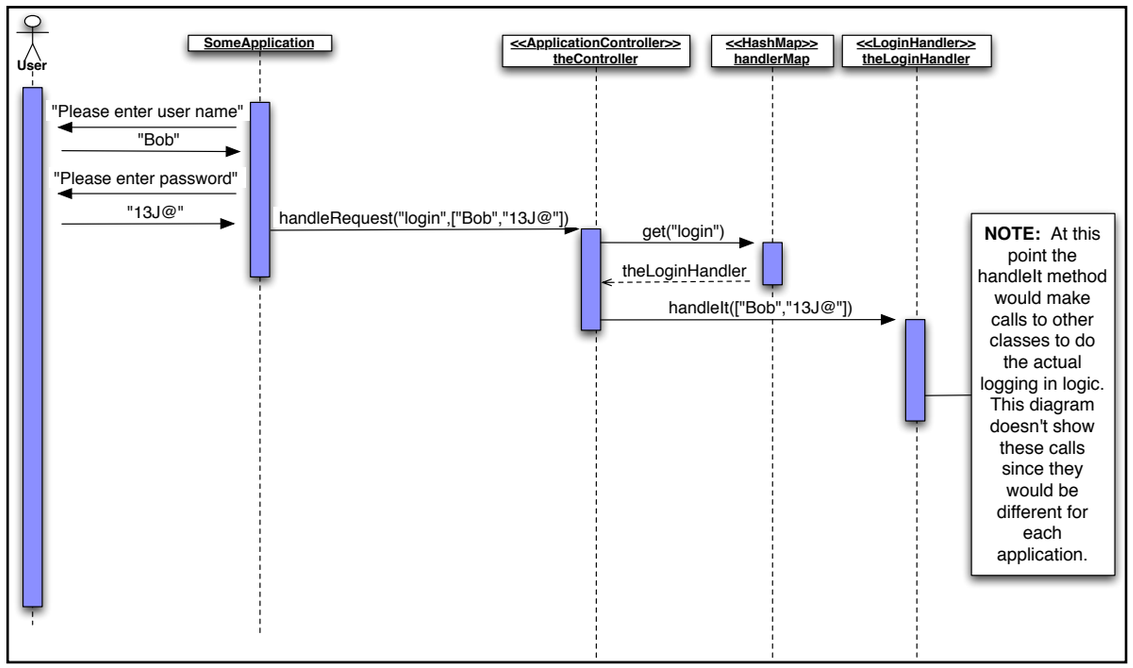


Diagram 2: a partial sequence diagram of an application controller

Diagram 2 as image

You can see the flow of the method calls in this diagram. It shows that the Handler instance specific to the *login* command is found in the HashMap and the Hander's handleIt method is called. If you were creating an online bank you may have a Handler instance for "transfer funds," another for "makeDeposit," and many others. If

your application was a calculator you could have a Handler instance for "add" or "subtract" or some other function.

Regardless of how many Handler instances your application has and what situations they are intended to handle the code within the ***handleIt*** method of each of your handlers will be unique to the situation it is to handle. These handlers should be independent of each other and do all of the computing as well as all interaction with the user interface, databases, web services, or any other resource that is needed to resolve the request.

Movie 7.1 An Analogy

Living in the Rocks

The Code:

There really isn't very much code needed to implement the naive version of the Application Controller Pattern, one class and one interface is all there is.

You've used the standard Java Runnable interface several times as you've gone through this book. Now its time to create the Handler interface described by the UML diagrams.

```
package com.doing.more.java.example.appcontrol;

import java.util.HashMap;

public interface Handler {
   public void handleIt(HashMap<String, Object> data);
}
```

Code Sample - Handler.java

Sample Handler.java source as file

There isn't much to this interface, but then again interfaces tend to be simpler than classes. They don't have code for their methods. If this is confusing you, please refresh you understanding of what you learned in Chapter 5 and how the Runnable interface was used in Chapter 6.

The class you'll need to create is found in the ApplicationController.java code sample. There isn't much code there either. One method, *mapCommand* is used to associate command Strings with instances of classes that implement the Handler interface.

```
package com.doing.more.java.example.appcontrol;

import java.util.HashMap;

public class ApplicationController {
   private HashMap<String,Handler> handlerMap = new HashMap();

   public void handleRequest(String command, HashMap<String,Object> data){
      Handler aCommandHandler = handlerMap.get(command);
      if (aCommandHandler != null){
         aCommandHandler.handleIt(data);
      }
   }

   public void mapCommand(String aCommand, Handler acHandler){
      handlerMap.put(aCommand,acHandler);
   }
}
```

Code Sample - ApplicationController.java

Sample ApplicationController.java source as file

The other ApplicationController method, *handleIt*, finds the Handler for any given command String and calls that Handler's *handleIt* method. Other than that, all there is is one attribute, the HashMap that uses the command Strings as keys and Handlers for each of the commands as the keys' associated value.

Using It:

For some of us, the ApplicationController and Handler code can look like magic. It's not. One of the easiest ways to figure out how it works is to write a simple app, run it in debug mode, and stepwise debug the app. It really helps to see the code in action. Take the time to do this. Watch every line of code execute. It will help you see how this abstract code can end up doing what you want it to do even though it looks like it couldn't.

So let's modify the JSONEchoServlet from the last chapter to use the application controller pattern. It is an overly simple example, but that is what we need right now. In the next chapter you'll see how to use the application controller to build a server that has multiple commands.

On the Server:

So let's start with the Handler. JSONEchoClient from the last chapter sent a HashMap that had "Speak" as the value for the "command" key. Let's build on that and implement a Handler called *SpeakHandler*. In the SpeakHandler's *handleIt* method will be code from the JSONEchoServlet's *doPost* method.

```
public class SpeakHandler implements Handler {
   @Override
   public void handleIt(HashMap<String, Object> dataMap) {
      try{
         JSONOutputStream outToClient =
                                    (JSONOutputStream)dataMap.remove("toClient");
         System.out.println("Just got:" + dataMap + " from client");
         dataMap.put("command", "Done");
         outToClient.writeObject(dataMap);
         System.out.println("just sent "+dataMap);
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Code Sample - SpeakHandler.java

Sample SpeakHandler.java source as file

Notice the code uses HashMap's *remove* method instead of *get*. The only reason *remove* was chosen was so *dataMap* could be written out to the client, the 'echo' behavior. JSONOutputStreams require that anything being written out implement Serializable. JSONOutputStreams, and other types of streams, are not Serializable so 'toClient' had to be removed so dataMap could be echoed back.

Since SpeakHandler is taking care of all the logic, JSONEchoServlet can be simpler. Gone is the code for dealing with the request. The servlet is now focused on generically dealing with requests from clients.

```
@WebServlet(name = "JSONEchoService", urlPatterns = {"/json"})
public class JSONEchoServlet extends HttpServlet {
   private ApplicationController theAppController = new ApplicationControl-
ler();

   public void init(){
      theAppController.mapCommand("Speak", new SpeakHandler());
   }
   protected void doPost(HttpServletRequest request, HttpServletResponse re-
sponse) throws ServletException, IOException {

      try {
         JSONInputStream inFromClient =
                                    new JSONInputStream(request.getInputStream());
         JSONOutputStream outToClient =
                              new JSONOutputStream(response.getOutputStream());

         HashMap<String, Object> dataMap =
                                                (HashMap) inFromClient.readObject();
         dataMap.put("toClient", outToClient);

         String aCommand = (String) dataMap.get("command");
         theAppController.handleRequest(aCommand, dataMap);

      } catch (Exception e) {
         e.printStackTrace();
      }
   }
```

Code Snippet 1

Sample JSONEchoServlet.java source as file

MVC Again:

Back in Section 2 of Chapter 6 a definition of what Model-View-Control is and what kind of code falls into each of the three parts was stated. Following those definitions, JSONEchoServlet is View code and SpeakHandler is Control code. JSONEchoServlet is View code because it busy doing IO with the client and SpeakHandler is Control code because it is making decisions. Granted, the isn't much to do in the way of decisions in this silly little sample. That's what's coming in Chapter 8.

CHAPTER 8

Hibernate and the Model

When you use a relational database within an app, the results of a query have to be converted from database records to things the app can understand each time a query is done. And then the app's data types and structures have to be changed to SQL for each insertion or update. This kind of code is bug prone and when done poorly, slows data transfer. Hibernate takes care of these problems for you.

Hibernate != Sleep

Lorem Ipsum

1. Why might you choose to use Hibernate in your application?
2. Why might Hibernate not be a viable choice for your application?
3. Why do many businesses choose to use Hibernate in their Java applications?

Hibernate is an enterprise quality tool used for database interaction in Java applications.

This section is an introduction to Hibernate. Some of the code examples are not what you would create for a real app. That will be in the next section. Also, a whole bunch of relational database words are going to be used in this chapter. Rather than try to use this book to learn databases, search the internet when you come across a word you don't understand. There are huge numbers of explanations and examples out there.

Hibernate consists of several very large open source Java libraries. These libraries, when included in your application, allow you as an engineer or programmer to deal with Java objects rather than database tables and records.

Each of your application model classes that you declare to be Hibernate entities represents a specific table in the database. It can have the same name as the table but usually does not since the rules for naming Java classes and database tables differ dramatically. The attributes of each of these objects matches the count and types of the fields in the database.

The idea behind Hibernate is that you can change the value of the attributes of the Java instances created from classes and the change is stored in the database. When you create a new object it can be stored in the database. Objects can also be deleted from the database though in most businesses deletion is rarely done.

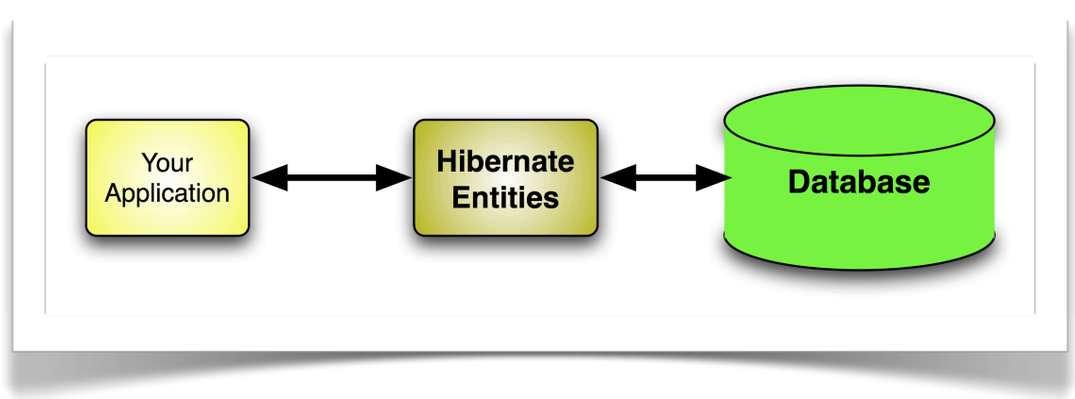


Diagram 1

Hibernate uses normal class based relationships rather than SQL joins for one-to-one, one-to-many, and many-to-many relationships. All to-many relationships are modeled as a class attribute of type ***Set***. Each attribute has a an accessor and a mutator method and look much like the JavaBeans from Chapter 3. When the getter is called the Hibernate libraries take over and execute the correct prepared statement query against the database.

The Sample's Database Design:

Diagram 2 shows the tables in a database and their relationships that will be used in the example that follows. While this chapter will not fully explain database design it will explain this diagram.

Database script as file

In the database there are three tables, *app_user*, *phone_number*, and a special kind of table called *user_number*. There is nothing special about these table names. They are used for the example only.

These tables have fields unique to the need. The app_user table has *id*, *uname*, *pword*, *manager_level*, *active*, and *session* columns. The *phone_number* table has *id* and *phone* columns. The special *user_number* table has *user_id* and *phone_id* columns.

The *id* columns in the *app_user* and *phone_number* tables are referred to as **primary keys** indicating that each record, or row, in the table with have a unique value in that field. The other fields have no special attributes.

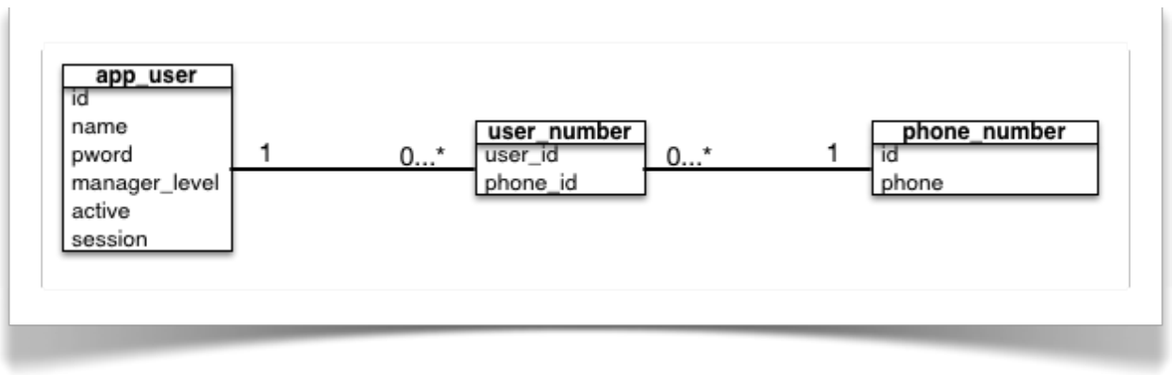


Diagram 2

Diagram 2 as image

If we stop and think for a minute, each user can have many phone numbers. You may have a mobile number, an office phone number, and an internet phone number. You may also share one or more of these numbers with someone else. Thus you have many phone numbers and each of your phone numbers can belong to many people. In database design this is referred to as a *many-to-many relationship*.

Diagram 2 shows how this relationship is implemented in a relational database. In order to do this correctly, a special third table, *user_number*, is needed. This table is referred to as a transition, junction, or join table. Its purpose is to associate the unique id's in the app_user table with the appropriate id's in the phone_number table and thus build the many-to-many relationship. This database is now ready to be represented by Hibernate entities.

Creating a Hibernate Entity:

Hibernate entities are very similar to and were modeled after JavaBeans so they have private attributes and no logic. A getter and setter method for each attribute is also included. So far it sounds just like a JavaBean and like JavaBeans they are Model code.

However, for a class to be a Hibernate entity, the class file must be modified using special Hibernate annotations. All annotations, including Hibernate ones, are declared using the @ symbol and act on the next line of Java code. You may have noticed the @override annotation previously when you were writing code in the code samples.

Several annotations can be listed prior to a single line of Java code. If they are then all of those annotations will all act on the single line. Code Snippet 1 shows how Plain Old Java Object (POJO) can be changed into a Hibernate entity.

```
@Entity
@Table(name = "app_user")
public class User {
```

Code Snippet 1

Sample User.java source file

Snippet 1 is an example of several annotations acting on one line of code. It shows how to use the *@Entity* annotation to declare the *User* class to be a Hibernate entity. The *@Table* annotation also acts on the next line of code. It associates the *User* class with the app_user table. This means that instances of User represent individual rows in the app_user table.

Attributes in a Hibernate entity represent columns in the table they are associated with. Code Snippet 2 shows the User class updated with these types of attributes. The names of the attributes, in this example, are the same as the names of the fields in the table. This is not a requirement. If you want the attributes to be named differently than the table field names there are plenty of examples on the web of how to do it. Take a look there.

```
@Entity
@Table(name = "app_user")
public class User {

   @Id
   @GeneratedValue
   private Integer id;
   private String uname;
   private String pword;
   private int active;
   private int manager_level;
   private String session;
```

Code Snippet 2

A pair of annotations are declared right before the private *id* attribute. Since the *id* attribute is the first line of code after the annotations, as with the @Entity and @Table annotations, they both act on the *id* attribute.

The *@Id* annotation indicates that the *id* attribute is the primary key in the table. The *@GeneratedValue* annotation declares that the *id* attribute will get set to a value generated by the database. *uname*, *pword*, *active*, *manager_level*, and *session* are also columns in the app_user database table.

There is yet one more attribute of the *User* class. As ***stated earlier***, each user can have multiple phone numbers and each phone number can belong to multiple users. In Java we represent this as a *Set* of *PhoneNumber* instances. You can see the declaration of this attribute in Code Snippet 3.

```
@Entity
@Table(name = "app_user")
public class User {

   @Id
   @GeneratedValue
   private Integer id;
   private String uname;
   private String pword;
   private int active;
   private int manager_level;
   private String session;

   @ManyToMany(cascade=CascadeType.ALL)
   @JoinTable(
         name="user_number",
         joinColumns = { @JoinColumn( name="user_id") },
         inverseJoinColumns = @JoinColumn( name="phone_id")
         )
```

Code Snippet 3

The annotations before the ***phoneNumbers*** line of code tells the application how to access the phone numbers associated with a specific user. The *@ManyToMany* annotation declares that the relationship between a *User* and a *PhoneNumber*, which will be described a little later, is many-to-many. Each *PhoneNumber* can belong to multiple *Users* and each *User* can have multiple *PhoneNumbers*.

The *@JoinTable* annotation tells the application that it should use the *user_number* table to find any phone numbers. It also describes the columns found in the *user_number* table. These columns are mentioned in the annotations as joinColumns, *user_id*, meaning

that the *user_id* field in the rows must match the *id* of a specific User entity *id*.

The *inverseJoinColumns* declaration says that what is found in the *phone_id* field of the *user_number* table will match the *id* of owned phone numbers in the *phone_number* table.

Again, this is not meant to be a database tutorial. If you have no database design experience you are probably confused. Hang in there and hopefully this will all come together when you run the example code.

Movie 8.1 An Analogy

Aunts as templates and children as proxies

Having now declared attributes and setup their annotations correctly, the getters and setters are created just as you would for a JavaBean. To see these download the User class source.

Sample User.java source as file

The *PhoneNumber* class is much simpler and found in the PhoneNumber.java code sample.

```
@Entity
@Table(name = "phone_number")
public class PhoneNumber {

   @Id
   @GeneratedValue
   private Integer id;
   private String phone;
   public String getPhone() {
      return phone;
   }
   public void setPhone(String phone) {
      this.phone = phone;
   }
}
```

Code Sample - PhoneNumber.java

The *PhoneNumber* class needs only to be annotated to be a Hibernate entity and have its attributes declared and annotated.

There is no class declared for the special *user_phone* table. It is not needed. That was taken care of in the User class's annotations.

Sample PhoneNumber.java source as file

The two classes, *User* and *PhoneNumber*, can now be used as proxies for the database tables and instances of them can be created and used as proxies for individual rows in the tables.

Using Hibernate Entities:

Now it is finally time to make some instances, store them in the database, modify them, and delete them from the database. An example application called HibernateRunner has been created and is discussed here. Again, HibernateRunner is not how you would design code for your app.

Sample HibernateRunner.java source as file

Code Snippet 5 shows two instances of the User class being created and then saved into the database. Notice that a transaction is begun before and committed after the Users are created and saved. A Hibernate Session's *save* method doesn't store entities in a completed fashion. It notifies the database to do a provisional save. The *commit* method of the transaction forces the database to permanently save them.

Database transactions are used in case something goes wrong. If, for some reason, a save failed we would want to 'rollback' any changes made so far so the data in the database doesn't get messed up. This is the responsibility of the *Transaction* instance.

```
/*
 * show how to add records to the database
 */
private void addNewUsers() {
   Session session = theHibernateUtility.getCurrentSession();
   /*
    * all database interactions in Hibernate are required to be inside a transaction.
    */
   Transaction transaction = session.beginTransaction();
   /*
    * create some User instances.
    */
   User aNameUser = new User();
   aNameUser.setUname("aName");
   aNameUser.setPword("aPass");

   User leeUser = new User();
   leeUser.setUname("lee");
   leeUser.setPword("barney");

   /*
    * save each instance as a record in the database
    */
   session.save(aNameUser);
   session.save(leeUser);
   transaction.commit();
   /*
    * prove that the User instances were added to the database and that
    * the instances were each updated with a database generated id.
    */
   System.out.println("aUser generated ID is: " + aNameUser.getId());
   System.out.println("anotherUser generated ID is: " + leeUser.getId());
}
```

Code Snippet 5

The *theHibernateUtility* instance that you see at the beginning of the *addnewUsers* method is an instance of a helper class that can be

downloaded as the *HibernateConfig.java source file.* It will not be discussed in detail in this book but it is where you will find items such as the connection URL string for the database, etc. Take a look at it. You may be able to modify and use it in applications you make.

Having now seen how to create and add Hibernate entities to the database, you need to know how to query them. Code Snippet 6 contains the *HibernateRunner* class' *showAllUsers* method.

In this method you can see a *Query* instance being created. Notice that the string that is passed to the *createQuery* method is NOT SQL. It looks something like SQL but is in fact Hibernate Query Language (HQL). HQL is class based. This is why you see *U%_{ser* declared rather than the *app_user* table you would use in a SQL query. The HQL query string says, "Get me all of the User objects and order them by their id".

Once the query has been created it is executed when its *list* method is called. This list method returns a List of User objects that is then iterated over to print out what was found. This is done for this example only. Your app will have different behavior.

```
/*
 * show how to get a collection of type List containing all of the records in the app_user table
 */
private void showAllUsers() {
   Session session = theHibernateUtility.getCurrentSession();
   Transaction transaction = session.beginTransaction();
   /*
    * execute a HQL query against the database.  HQL is NOT SQL.  It is object based.
    */
   Query allUsersQuery = session.createQuery("select u from User as u order by u.id");
   /*
    * get a list of User instances based on what was found in the database tables.
    */
   users = allUsersQuery.list();
   System.out.println("num users: "+users.size());
   /*
    * iterate over each User instance returned by the query and found in the list.
    */
   Iterator<User> iter = users.iterator();
   while(iter.hasNext()) {
      User element = iter.next();
      System.out.println(element.toString());
      System.out.println("num of phone numbers: "+element.getPhoneNumbers().size());
   }
   transaction.commit();
}
```

Code Snippet 6

Modifying data in the database is done in much the same way as getting data from the database. In Code Snippet 7 only one object is to be changed so the HQL query string includes a clause that states only the user with the *uname* attribute *lee* is to be fetched.

```
/*
 * show how to modify a database record
 */
private void modifyUser() {

   Session session = theHibernateUtility.getCurrentSession();
   Transaction transaction = session.beginTransaction();
   /*
    * get a single User instance from the database.
    */
   Query singleUserQuery = session.createQuery("select u from User as u where
u.uname='lee'");
   User leeUser = (User)singleUserQuery.uniqueResult();
   /*
    * change the user name for the Java instance
    */
   leeUser.setUname("Joshua");
   /*
    * call the session merge method for the User instance in question.  This tells the data-
base that the instance is ready to be permanently stored.
    */
   session.merge(leeUser);

   /*
    * call the transaction commit method.  This tells the database that the changes are ready
to be permanently stored.
    */
   transaction.commit();
   /*
    * permanently store the changes into the database tables.
    */
   showAllUsers();
}
```

Code Snippet 7

Because only one User should match this requirement, the Query class' *uniqueResult* method is used instead of its *list* method. This guarantees that only one User object is returned.

The User instance, *leeUser*, is then modified by changing its userName attribute. Instead of calling *save* like we did when new Users were created the Session's *merge* method is used. This causes the existing row in the database table to be updated rather than creating a new row.

As of yet no PhoneNumber instances have been created nor has a PhoneNumber been shared by more than one User. This is the purpose of the code in Code Snippet 8.

In this snippet two unique users are retrieved from the database and a new PhoneNumber is instantiated. The PhoneNumber is then added to Sets belonging to each of the users. Once this is done the PhoneNumber is saved, the User changes merged, and the transaction is committed.

```
private void addSharedPhoneNumber() {
   Session session = theHibernateUtility.getCurrentSession();
   Transaction transaction = session.beginTransaction();

   Query joshuaQuery = session.createQuery("select u from User as u where
u.uname='Joshua'");
   User joshuaUser = (User)joshuaQuery.uniqueResult();
   Query aNameQuery = session.createQuery("select u from User as u where
u.uname='aName'");
   User aNameUser = (User)aNameQuery.uniqueResult();
   /*
    * create a PhoneNumber instance
    */
   PhoneNumber sharedPhoneNumber = new PhoneNumber();
   sharedPhoneNumber.setPhone("(546)222-9898");
  /*
   * add the shared phone number to the joshuaUser
   */

   Set<PhoneNumber> joshuaPhoneNumbers = joshuaUser.getPhoneNumbers();
   joshuaPhoneNumbers.add(sharedPhoneNumber);
  /*
   * set the single phone number to be used by more than one User
   */
   Set<PhoneNumber> aNamePhoneNumbers = aNameUser.getPhoneNumbers();
   aNamePhoneNumbers.add(sharedPhoneNumber);
  /*
   * inform the database that the phone number should be ready for permanent storage.
   */
   session.save(sharedPhoneNumber);
  /*
   * inform the database that the modified User instances should be ready for permanent storage.
   */
   session.merge(joshuaUser);
   session.merge(aNameUser);
  /*
   * permanently store the changes into the database tables.
   */
   transaction.commit();

   showAllUsers();
}
```

Code Snippet 8

Nearly all done now. Hang in there. All that is left is to see how to delete records from the database. Code Snippet 9 shows how to do this.

```
private void deleteAddedUsers() {
   Session session = theHibernateUtility.getCurrentSession();
   Transaction transaction = session.beginTransaction();

   int numUsers = users.size();
   System.out.println("user count: " + numUsers);
   for(int i = 0; i < numUsers; i++){
      System.out.println("deleting user "+users.get(i).getUname());
      session.delete(users.get(i));
   }
   transaction.commit();
   /*
    * at this point the records have been removed from the database but still exist in our
class list attribute.
    * Do not store lists retrieved from the database since they will be out of synch with the
database table from which they come.
    * This example shows that you should not store retrieved lists.
    */
   System.out.println(users);
   users.clear();
   /*
    * now the Java instances are also gone and the database is back to its original state so
the example application can be run again.
    */
   System.out.println(users);
}
```

Code Snippet 9

In order to delete a Hibernate entity from the database you must have a reference to it. You may get this reference by querying the database as we have seen in the other code snippets or, as in this

example, get it from an ArrayList where it had previously been stored. Either way, the session is retrieved, the transaction started and then the session's *delete* method is called. Once ***transaction.commit()*** is executed the row that matches the entity is removed permanently from the database table.

Hibernate entities may seem daunting at this point. Take some time here. Stop, ponder, and think about what you've seen. Play with them. When you do so you will begin to see a repeating pattern that saves you a great deal of time and effort. You were going to create the data as JavaBeans anyway so add the annotations, modify the ***HibernateUtilSingleton*** slightly, and then start using them. Overall you will write less code and it will be more modular code. Oh, and by the way, Hibernate doesn't work in Android™ at the time of the writing of this book.

Now let's put everything together in the server. Let's make it fully *MVC*.

All Together Now

Points to Ponder

1. How do the pieces go together to make a whole?
2. How else can I apply these concepts, principles, and platforms?

All species in a biome contribute to its health. Remove a species and the rest won't be as healthy.

You've seen all of the pieces now let's take a look at how to create a servlet that uses the Application Controller pattern and Hibernate. The sample consists of a small portion of a JSON service for an online store. To keep it simple, only some of the store will be created in code. You'll see handlers for registering, logging in, and logging out, but not for other things like searching for a product, adding a product to a cart, checking out, and paying. The sample store app will also use the User and PhoneNumber classes along with the database from the previous section.

In line with keeping it simple, the model will only have some of its parts and not all of those

parts will be used by the sample Handlers. Feel free to add in the missing parts if you want to. I think after you see how the sample parts work it would be great if you started something of your own from scratch. Just try to be calm. You can do it.

The Model:

In the last section you found out how to use Hibernate methods and instances to manipulate a database. Now it's time to finish up 'the model.' Models, in accordance with the definition we are using, consist of more than just the Hibernate entities representing the table. You should try to hide Hibernate from the rest of your code. The better you can do this, the easier it is to make dramatic changes.

Instead of putting hibernate code in the handlers for your app, the control code, you could put them in a separate class behind a method. Your handler could then call something like *updateUser* instead of doing Hibernate things like starting and getting sessions, starting transactions, and merging. If you take this design approach and later you have to switch to some other data storage tool, like firebase, text files, or something else, you ONLY have to change the code in updateUser method.

If you don't hide the specifics of how data storage is implemented you will have to change every handler that updates a user. That is bad. You will have a lot of work to do and it will be hard to make sure ALL places where the code needed to change got changed. It is almost always better to create one or more model classes.

Code snippet 1 has the first lines of the StoreModel.java sample in it.

```
public class StoreModel {
   private HibernateConfig theHibernateConfiguration;
   public StoreModel(){
     this.theHibernateConfiguration = new HibernateConfig();
   }
   public void addUser(User aUser){
      Session theSession = this.theHibernateConfiguration.getCurrentSession();
      Transaction transaction = theSession.beginTransaction();
      theSession.save(aUser);
      transaction.commit();
   }
public void updateUser(User aUser){
   Session theSession = this.theHibernateConfiguration.getCurrentSession();
   Transaction transaction = theSession.beginTransaction();
   theSession.merge(aUser);
   transaction.commit();
}
```

Code Snippet 1

Sample StoreModel.java source as file

By hiding the Hibernate specific code behind the *addUser* and *updateUser* methods, it becomes much easier to write the Handler to register a new user.

The Handlers:

When registering, in the RegistrationHandler.java code sample, a check must be done to see if the user name and password the user would like to use are already taken. The StoreModel's getUser method is used. If it returns null then our control code 'knows' the user name and password pair are available. It then creates a random Universally Unique IDentifier (UUID) to use as a HTTP session id. After this, RegistrationHandler creates a new User, sets the appropriate values, and uses the StoreModel's *addUser* method. No-

tice that the RegistrationHandler has no idea what is going on inside of *getUser* or *addUser*. It shouldn't!

```
public class RegistrationHandler implements Handler {
   @Override
   public void handleIt(HashMap<String, Object> dataMap) {
      String userName = (String)dataMap.get("uname");
      String password = (String)dataMap.get("pword");
      StoreModel theModel = (StoreModel)dataMap.get("model");
      User foundUser = theModel.getUser(userName,password);
      HashMap<String,Object>responseMap = new HashMap<>();
      String sessionID = "";
      if(foundUser == null){
         UUID sessionUUID = UUID.randomUUID();
         sessionID = sessionUUID.toString();
         User aUser = new User();
         aUser.setSession(sessionID);
         aUser.setUname(userName);
         aUser.setPword(password);
         theModel.addUser(aUser);
         responseMap.put("id",sessionID);
      }
      responseMap.put("id",sessionID);
      JSONOutputStream outToClient =
(JSONOutputStream)dataMap.get("toClient");
      try {
         outToClient.writeObject(responseMap);
      }
      catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Code Sample - RegistrationHandler.java

Sample RegistrationHandler.java source as file

The LoginHandler and LogoutHandler look much the same as RegistrationHandler. They contain no Hibernate code.

In the StoreServlet.java sample, I've commented out some Handlers that you may want to implement yourself. It would be good practice and you shouldn't have to add anything to the StoreModel.java sample to make them work. There are a few difference between StoreServelet and the JSONEchoServlet.java sample you learned about in Chapter 6.

The Servlet:

The differences are in Code Snippet 2. There is a new attribute, *theModel*. It is the model class hiding all the Hibernate code. You can see it being used just before theAppController's handleRequest method is used. In order for the model instance to be used in each of the different Handlers, I added it to the dataMap like the JSONOutputStream was added in Chapter 7. Now when RegistrationHandler needs to call StoreModel's *getUser* method, there is no need to write "StoreModel theModel = new StoreModel()."

```
@WebServlet(name = "StoreServlet")
public class StoreServlet extends HttpServlet {
   private ApplicationController theAppController =
                                                new ApplicationController();
   private StoreModel theModel = new StoreModel();

   public void init(){
      theAppController.mapCommand("register", new RegistrationHandler());
      theAppController.mapCommand("login", new LoginHandler());
      theAppController.mapCommand("logout", new LogoutHandler());
      //theAppController.mapCommand("addMyNumber", new AddPhoneHandler());
      //theAppController.mapCommand("myNumbers", new GetPhoneNumbersHan-
dler());
      //theAppController.mapCommand("upgrade", new ManagementChangeHandler());
   }
   protected void doPost(HttpServletRequest request, HttpServletResponse re-
sponse) throws ServletException, IOException {

      try {
         JSONInputStream inFromClient =
                              new JSONInputStream(request.getInputStream());
         JSONOutputStream outToClient =
                              new JSONOutputStream(response.getOutputStream());
         HashMap<String, Object> dataMap =
                              (HashMap) inFromClient.readObject();
         dataMap.put("model",this.theModel);
         dataMap.put("toClient", outToClient);
         String aCommand = (String) dataMap.get("command");
         theAppController.handleRequest(aCommand, dataMap);

      } catch (Exception e) {
         e.printStackTrace();
      }
   }
```

Code Snippet 2

[*Sample StoreServlet.java source as file*]

You really don't want to make a new instance of the StoreModel class each time a request comes in. Making all of the database connections and getting Hibernate all set up over and over again would be bad. It would use up huge amounts of CPU cycles. In other words, your application would be slow and wouldn't scale. Instantiating it once and reusing it for all requests is a much better option.

That's it! All the parts are now working together. Congratulations. You have learned a lot of stuff. Good stuff. Things that are really nice to know. You've come a long way, but we've only scratched the surface of what you can do with Java, Android, and Hibernate.

Where Next?

A good start is only a start. If you don't keep going did you really start or are you right where you began?

There are still many things about you don't know. A few of these are listed here with links to help in your discovery process.

A more complete tutorial on Hibernate. It uses the older XML version of how to configure Hibernate to work with a database rather than the approach taken in the HibernateConfig.java code sample.

An Android tutorial focusing on user interface elements and Android specific services.

More about how to use JUnit.

Information on DevOps automation for Java apps.

CHAPTER 9

Appendices