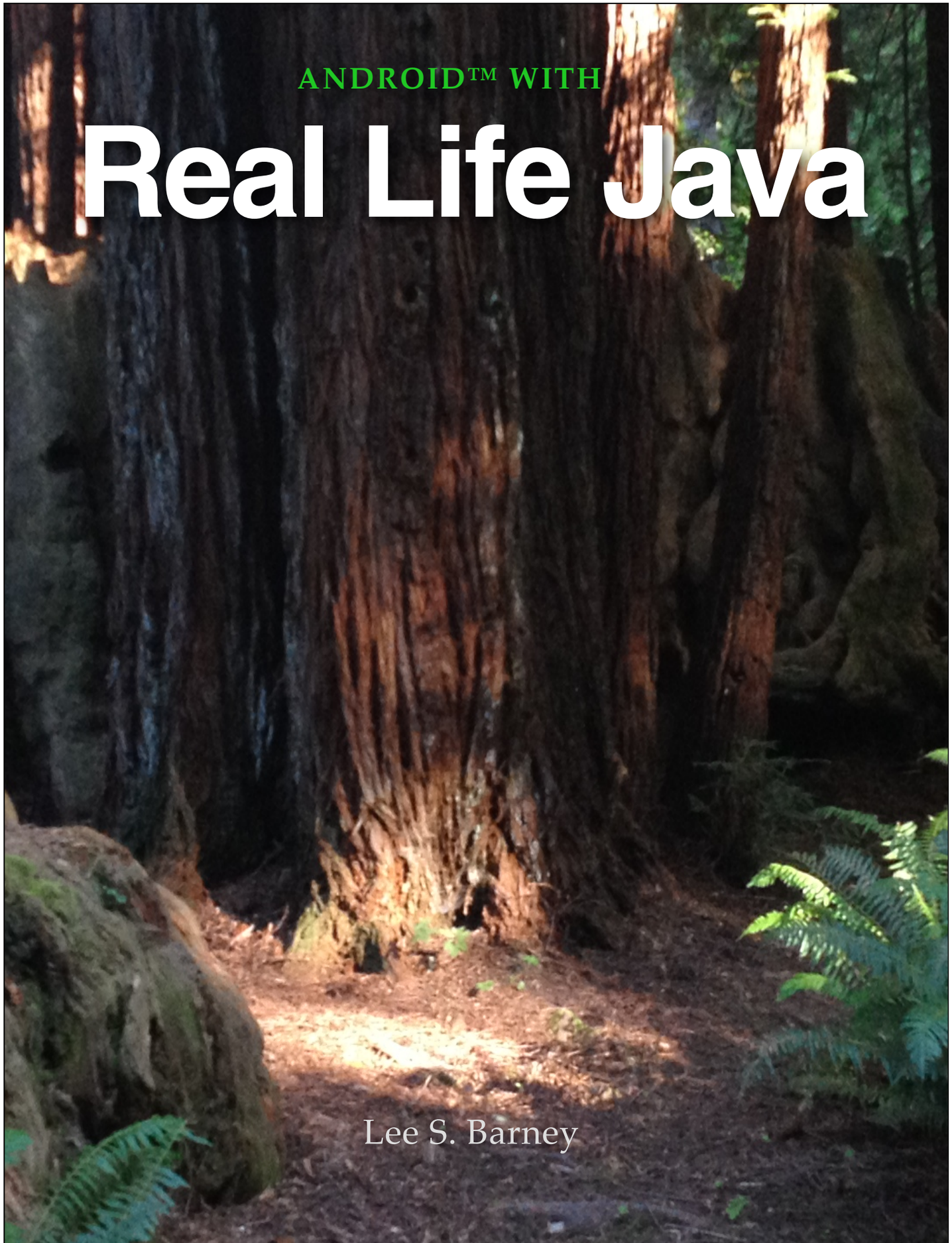
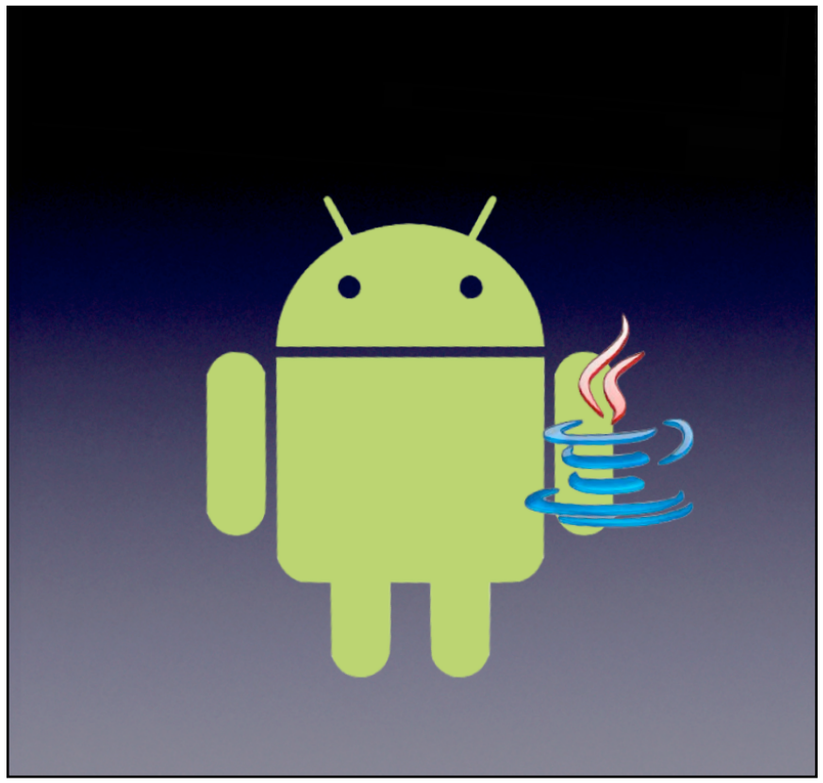
ANDROID™ WITH

Real Life Java



Lee S. Barney

ANDROID™ WITH REAL LIFE JAVA



Dedication

This book is dedicated to my wife and children. They make everything worth while. Also to the friends I have had throughout my life who helped get me to this point. And finally to Marty Larkin, who helped me learn how to learn by thinking and doing in addition to reading.

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, or others, Author does not guarantee the accuracy, adequacy, or completeness of any information included in this work and is not responsible for any errors or omissions or the results obtained from the use of such information.

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!

This book has been created to take advantage of its electronic format and is designed to work for both technical professionals and students in a classroom. The content of the book uses the Android™ and Java SE toolsets to teach principles of programming and basic introductory software engineering.

The book is not designed as an exhaustive API, UML, or software engineering book, a book of code script-lets to copy, nor a book full of detailed definitions of every possible word since this type of information is 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.

All chapters in the book contain a blend of both programming and engineering principles. By combining both programming and engineering you are exposed to both design and the implementation concepts so you can see the importance of, and relationships between the two.

The first chapter consists of a series of descriptions which indicate and give a brief review of points that are indicative of the information needed to understand the main topics presented in the subsequent chapters. These descriptions are not exhaustive primers.

What Information You Should Already Have

To fully use the information presented in this book you should have mastered the following topics previously:

1. Arrays - creation and use

2. Control structures:

    for loops

    for each loops

    while and do-while loops

    if-else and case

3. Boolean logic:

    &&

    ||

    etc.

4. Comparators:

    ==

    !=

    <

    >

etc.

5. Variables

6. References

7. data types

8. object oriented design:

    constructors

    attributes

    methods

    the meaning of private, protected, and public

    static vs. non-static

    classes vs. objects

9. using predefined interfaces

10. basic Java objects such as String, Integer, StringBuffer, etc.

11. using Eclipse™ or some other IDE for development of code

12. event driven programming and event handling

13. basic UML Class diagrams

14. basic understanding of using JUnit tests

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 video that is a non-programming analogy of the information presented. These videos are not screen captures where you watch someone write, compile, and run something. They are designed to help you relate the new information to something that you probably already know.
4. A detailed and 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.
5. A series of self-check questions to help you evaluate your understanding of the material to be presented.

CHAPTER 1

Stuff You Need to Know

These prologues contain information that you should understand in order to be able to be successful in learning the information presented in this book. Read them and use them as a reference while reading the book.

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 a reference.

Required Tools

Install These

1. The latest versions of the Android Studio™ (Google's Android IDE) and Eclipse™.
2. A UML modeling tool

Android™ tools can come from different sources

The tools used to create Android™ applications are open source and free. You may choose to edit and compile your applications by hand using a command line terminal and a text editor. You may choose to use Android Studio™, or you can choose to use some other IDE. Since the mobile space constantly changes watch for new development environments from which to choose.

Regardless of which environment you chose, you will need to download it and follow its installation and setup instructions. Android Studio™ can, at the time of this book was published, be found at http://developer.android.com.

Think First

Steps to Success

1. Think - figure out what you don't know and learn it.
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 - make it

Think before you act

Throughout history, a common approach has been used to create any type of object in the quickest and easiest way. Why would developing software be any different? It should not.

This historic approach is cyclic. This means that you take a stab creating your item and then go back and revise one or more decisions made using the approach and then continue on.

This historic approach is composed of four simple steps.

Think - What does the customer want? What types of knowledge are required to create the thing the customer wants? Do I already have this knowledge? If not, where can I find this knowledge? Playing with the new knowledge in a simple sandbox code set allows you to see how each portion of the new knowledge behaves and can be used. Example: If you don't know how a 2X4 wooden board behaves you should not attempt to build a house. Maybe you should play around with a few of them instead of instantly starting to build. There is no replacement for experience.

Design - Decide how the thing 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, complex or simple. 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 buildings this step would be the building codes. In software this would consist of User Interaction testing, Unit testing, Component testing, System testing, and Installation testing. 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 steps 1 or 2 or maybe even both. Go back and do them first.

Create - Begin building the item, be it something physical or conceptual like software. Notice that this step 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 as needed. This does not mean 'throw away and start all over again'. It means make modifications and try again.

The standard approach taken by most software programmers goes 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 repeat at step 2 until it works.
6. The company ships 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 words like them, are heard. The 'think last' approach all but guarantees that the software ships late, over budget, and doesn't have the features the customer wants. It is a proven failure method. Don't use it! Use the method that is known to work. **Think, Design, Test, Create**. Until you use it it may seem that it 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 Sample 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 = aPerson.getClass().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 Sample 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						

User Requirements

Questions to Ask

1. What type of application does the customer want?
2. What is it supposed to do?
3. What parts must it have?
4. What must the user be able to do with it?
5. What systems and tools will be used to produce the application?

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

Rarely, if ever, do customers supply an initial list of requirements sufficient to create the application they want. It is important to use their initial request as the basis for a discussion between your team and them. Gradually you will be able to get sufficient information to begin the development process. This means you will regularly be contacting the customer throughout the development process to sanity check your understanding of what the app is supposed to be, look like, and act like.

When you create an app for your own company, a team you are working with, or yourself the same thing will happen. Your and your team's

understanding of the app will shift as you develop the app. With changing understanding will come honing and refinement, and occasionally completely rethinking, the purpose and design of the app you are creating.

Application Requirements

An example of customer requirements is helpful. This example is more complete than most requirements documents I've seen but is still incomplete. Often, all customers start us off with is an idea of something that is needed.

The Example as First Received

Major Medical Inc., a large health care provider, has decided that due to ongoing federal HIPA regulations and productivity issues they need to move away from the laptop based home healthcare application they are currently using and move to a mobile platform. They have approached Mighty Corp, your employer, to produce an application for their home health nurses that will meet their use and regulatory needs. The application has been named MMHome.

Major Medical has broken development up into three versions. Funding for development of the second preliminary version depends on the successful delivery of the first preliminary version. After successful completion of the second preliminary version a contract will be created to reflect all the data and other needs of the application.

Application Requirements (from Major Medical Inc.)

First Preliminary Version:

Data entry and viewing must be secured using strong username and password combinations.

All data entered must be timestamped with the time they are initially set.

The data must consist of visits to patients. The date, time, and location of each visit and standard, basic medical information for each patient visit must be stored. This data is to include:

systolic and diastolic blood pressures,

heart rate,

temperature,

blood sugar levels,

weight,

respiration rate,

the time of the start of the visit, and

the duration of the visit.

All data must be stored in a searchable fashion.

Nurses must be able to view past visit data for specific patients as well as statistical and trend information regarding specific patients.

All data must be stored on the device.

It must be easy to use since most users have medical but not computer skills and there is no time for even moderate training.

To cut cost, Major Medical Inc. has requested that open source tools be used where reasonable and feasible.

Second Preliminary Version Additions:

Data must be stored on a database and accessed via a custom Java server application.

All data must be encrypted during network transport.

Data must be stored in a searchable fashion in the database.

Any updates must have a time stamp separate from the data creation time stamp.

Data entries must be stored securely on the device only if there is no internet connectivity and must update the remote server and database without user interaction when connectivity is available.

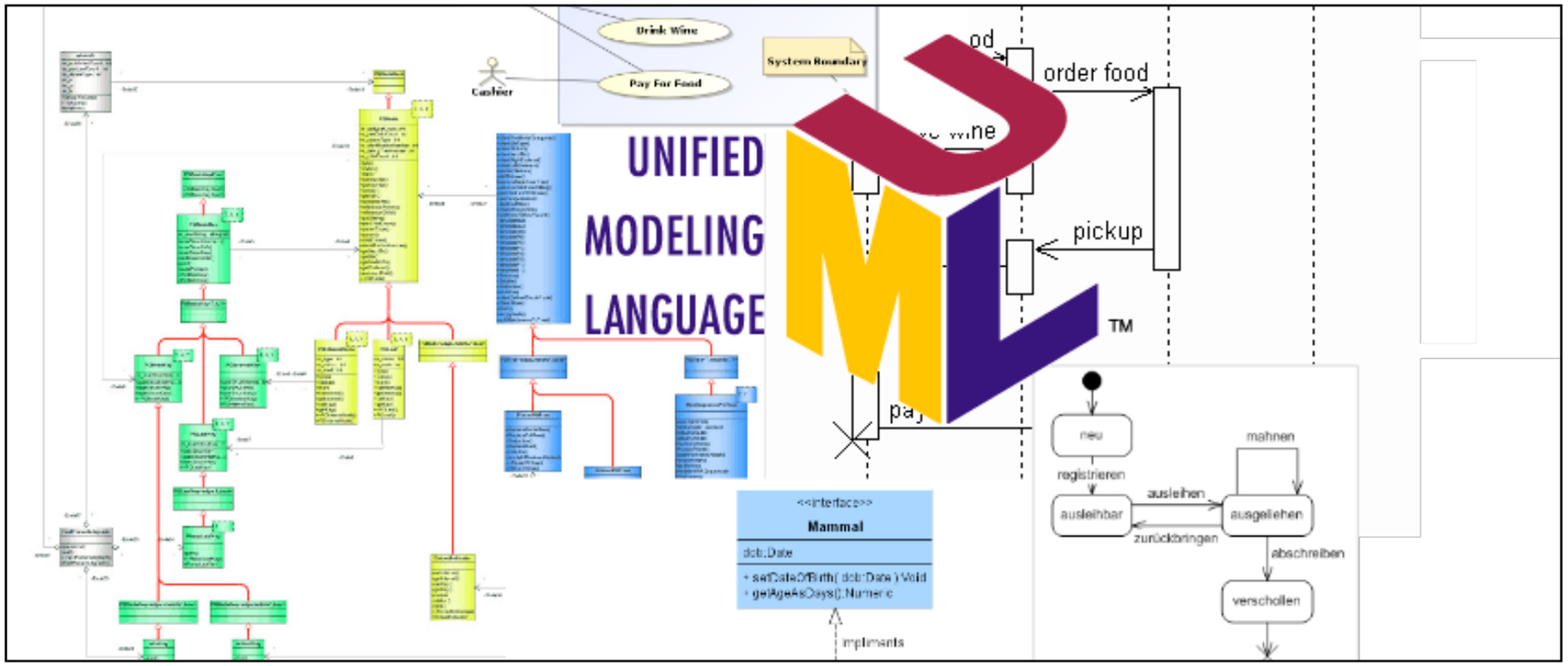
Only the mobile application may change or modify the data via the server.

Hibernate is to be used for all database interaction and MySQL is to be used as the database for this application.

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.

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 and creating as a work team.

When working in industry UML is created and interpreted in many ways. Work with your manager and team to understand how you should create documents to 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

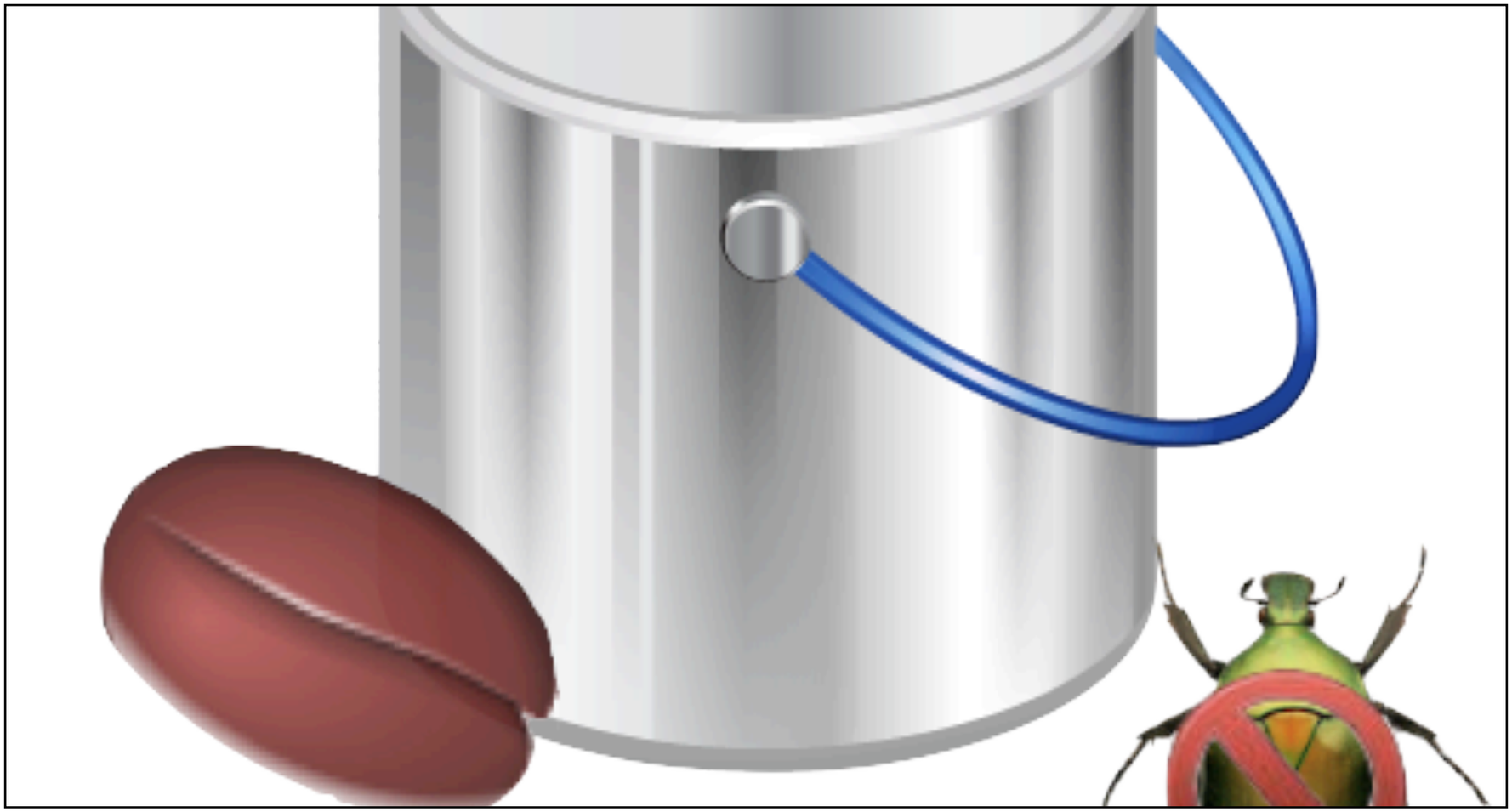
Table 1

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

CHAPTER 2

JUnit, Collections, & Beans



There are a few items that are the building blocks of all well designed and executed Java applications. These include collections, not arrays, of objects, objects called JavaBeans that associate data but have not 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**) just like other Java objects you have already made but with one exception. The difference is that a JavaBean has no methods that contain logic.

For 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 logic method since it includes what normally would be thought of as behavior.

A JavaBean would have no such method. JavaBeans only have private attributes, and accessors and mutators for those attributes. If your Java object meets this minimum standard and it has no methods implementing behavior it is a J

JavaBean. An example of this minimal type of implementation can be seen in Code Sample 1.

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

Code Sample 1

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 Eclipse™ to create your application these methods can be inserted for you. To do this right-click on the source code, select **Source** in the popup menu, and then select **Generate hashCode() and equals()....**

The ***equals*** method is used to compare two instances of the JavaBean. An example for the PersonBean class is found in Code Sample 2.

```
public boolean equals(Object obj) {
          if (this == obj)
                return true;
          if (obj == null || getClass() != obj.getClass())
                return false;
          PersonBean other = (PersonBean) obj;
          if (age != other.age)
                return false;
          if (firstName == null) {
                if (other.firstName != null)
                      return false;
          } else if (!firstName.equals(other.firstName))
                return false;
          if (lastName == null && other.lastName != null) {
                return false;
          } else if (!lastName.equals(other.lastName))
                return false;
          return true;
    }
```

Code Sample 2

Code Sample 3 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() {
          final int prime = 31;
          int result = 1;
          result = prime * result + age;
          result = prime * result + ((firstName == null) ? 0 :
                                     firstName.hashCode());
          return = prime * result + ((lastName == null) ? 0 :
                                     lastName.hashCode());
}
```

Code Sample 3

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 much better way to store related data than arrays or some other sort of collection.

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 theobjects are sorted, 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 using 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 the keys, not the values. Both the ***put*** and ***get*** methods work just like in a HashMap but also TreeMap also has several methods you can use to get the keys in order. These methods are ***firstKey***, ***higherKey***, ***lastKey***, and ***lowerKey***. With these keys you can then use the *get* method 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 Sample 4 is an example of using a HashMap. This 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. In the code example this is done by including ***<String, PersonBean>*** in both the variable declaration, ***companyMap***, and the HashMap constructor call, ***new HashMap***. 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 leePerson = new PersonBean();
leePerson.setFirstName("Fred");
leePerson.setLastName("Barney");
leePerson.setAge(21);

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

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

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

Code Sample 4

Parametric types are not required. If you do not include parametric types in your declaration you can then mix many types of Java objects as the keys and many types for the value. This is different than the native arrays you have used in the past. They can only contain values of one type.

Code Sample 5 shows how to add these same JavaBeans to an ArrayList. Notice that there is no key. Only 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.

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

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

Code Sample 5

Movie 2.1 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'. It won't work for you.

Steps:

Think:

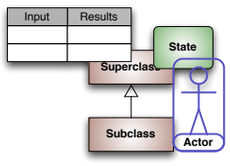
Evaluate the following customer requirements.

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.

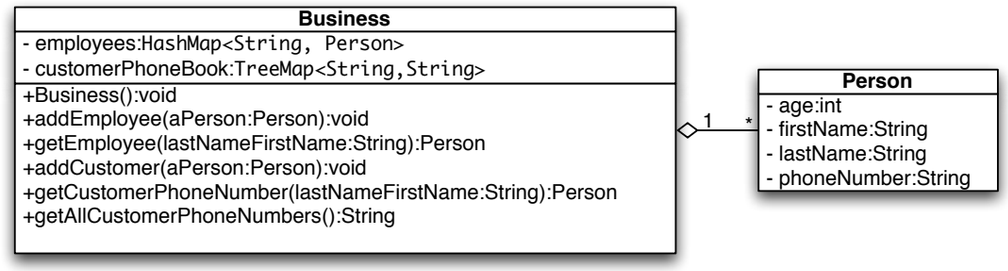


Diagram 1

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.

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.

Test:

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 Code Sample 6 (Download Code Sample 6 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 by when 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 in Code Sample 7 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;

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

    public Business() {
          employees = new HashMap<String,Person>();
          customerPhoneBook = new TreeMap<String,String>();
    }
    public void addEmployee(Person aPerson){
          if(aPerson != null && aPerson.getLastName() != null &&
                        aPerson.getFirstName() != null){
               String key = aPerson.getLastName()+
                        ", "+aPerson.getFirstName();
               employees.put(key, aPerson);
          }
    }
    public Person getEmployee(String lastAndFirstName){
          return employees.get(lastAndFirstName);
    }
```

Code Sample 7a

```
    public void addCustomer(Person aCustomer){
          if(aCustomer != null && aCustomer.getLastName() != null
                        && aCustomer.getFirstName() != null){
               String key = aCustomer.getLastName()+", "
                             +aCustomer.getFirstName();
               customerPhoneBook.put(key,
                                  aCustomer.getPhoneNumber());
          }
    }
    public String getCustomerPhoneNumber(String lastAndFirstName){
          if(lastAndFirstName == null){
               return null;
          }
          return customerPhoneBook.get(lastAndFirstName);
    }
    public String getAllCustomerPhoneNumbers(){
          StringBuffer phoneNumberBuffer = new StringBuffer();
          for(String aKey = customerPhoneBook.firstKey();
                     aKey != null;
                     aKey = customerPhoneBook.higherKey(aKey)){

               String phoneNumber = customerPhoneBook.get(aKey);
               phoneNumberBuffer.append(aKey);
               phoneNumberBuffer.append(" - ");
               phoneNumberBuffer.append(phoneNumber);
               phoneNumberBuffer.append("; ");
          }
          return phoneNumberBuffer.toString();
    }

}
```

Code Sample 7b

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

Code Sample 8

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 Code Sample 7.

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 Sample 8 shows the ***addEmployee*** and ***getEmployee*** methods changed to use an ArrayList. Notice that the ***addEmployee*** method is simpler than the version in Code Sample 7 but that the ***getEmployee*** method is more complex.

Since these changes being made are hidden from code outside the Business, a standard Object Oriented design approach, there is no need to change the JUnit test. The test still requires the same behavior since no customer requirements have changed. The UML class diagram would have to change slightly as the HashMap would need to be replaced by an ArrayList in the design. Overall, not many changes to diagrams are needed if this code change is made.

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 the same way. This high count means

that 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.

Review

Review 2.1 JUnit, Collections, & Beans

Question 1 of 6

All JavaBeans contain

- **A.** Attributes
- **B.** Behavior
- **C.** Both A & B
- **D.** Neither A or B

Check Answer

Suggested Further Exploration

- ☑ Collection Iterators
- ☑ The Model-View-Control pattern

Self Check

- ☑ When should you use the HashMap, TreeMap, and ArrayList classes?
- ☑ What is the purpose of a test matrix?
- ☑ What is the purpose of a unit test?
- ☑ What are some advantages of using the **Think-Design-Test-Create** approach to making software?
- ☑ What are the types of activities you do in the **Think** step?
- ☑ In addition to the 'Happy Path' what types of entries are needed in a test matrix?

CHAPTER 3

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?

Rivers and streams move water, silt, and minerals to the sea.

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 generated output. 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 Android™ also restricts our choice. At the time of the writing of this book the default binary serialization of objects for the Java language don't work in Android™. That is a heck of a limitation. This may push us away from selecting binary serialization if we were creating an Android™ app. If not it is no limitation.

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. Each character handled means more battery power 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. Android™ developers use XML to describe the user interface for their app and open source linux projects use it a lot.

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

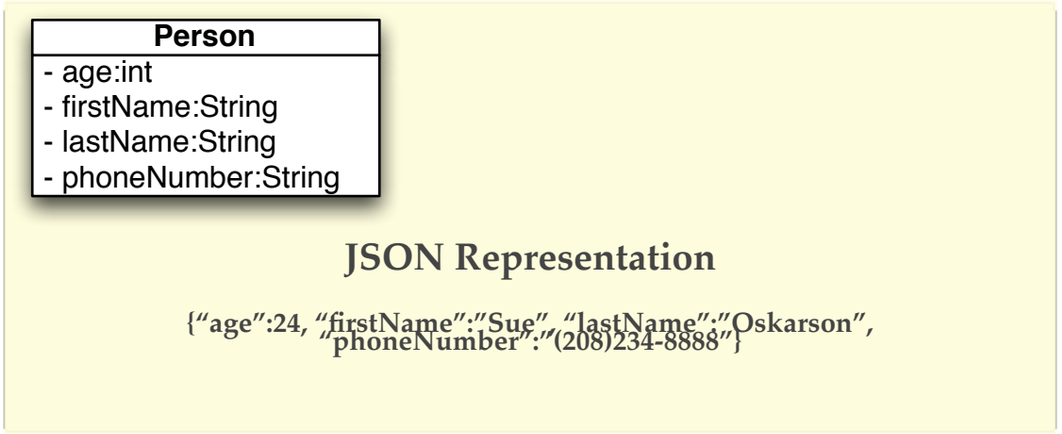


Diagram 1: UML and JSON representations

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. [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 object representations must be strings but the values can be strings, boolean true/false values, numbers as integers or doubles, JSON objects, or JSON arrays.

Parsing and Getting JSON Strings

There are many JSON libraries and several 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://github.com/yenrab/qcJSON.

Code Sample 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 TestObject 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 Sample 1

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

Code Sample 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 JavaObject is done in much the same way. Code Sample 3 shows this same string being converted back into a ***TestObject***.

```
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);

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

Code Sample 3

To convert a JSON string to an object 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 and an ***ArrayList*** if the string describes a JSON 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 and 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*** but there are many more.

Java also allows you to 'wrap' these any stream 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 Sample 4, if you want to write JSON to a file you pass 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 Sample 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 write the object to the file. Doing so pushes a JSON string representing the object into the the requested file.

Reading a JSON string from a file is done in a very similar fashion. To read an object you wrap a ***FileInputStream*** in a ***JSONInputStream***. Having done so you call the ***JSONInputStreams readObject*** method. The result of this call is either a ***HashMap*** or ***ArrayList*** much like the ***parse*** method described earlier.

Diagram 2 is a UML Sequence diagram that can help you see how these objects interact. It shows an existing application of some type and an already existing Serializable object that is to be used to create a JSON string stored using the JSONOutputStream. It shows how this stream interacts with the FileOutputStream and the Serializable.

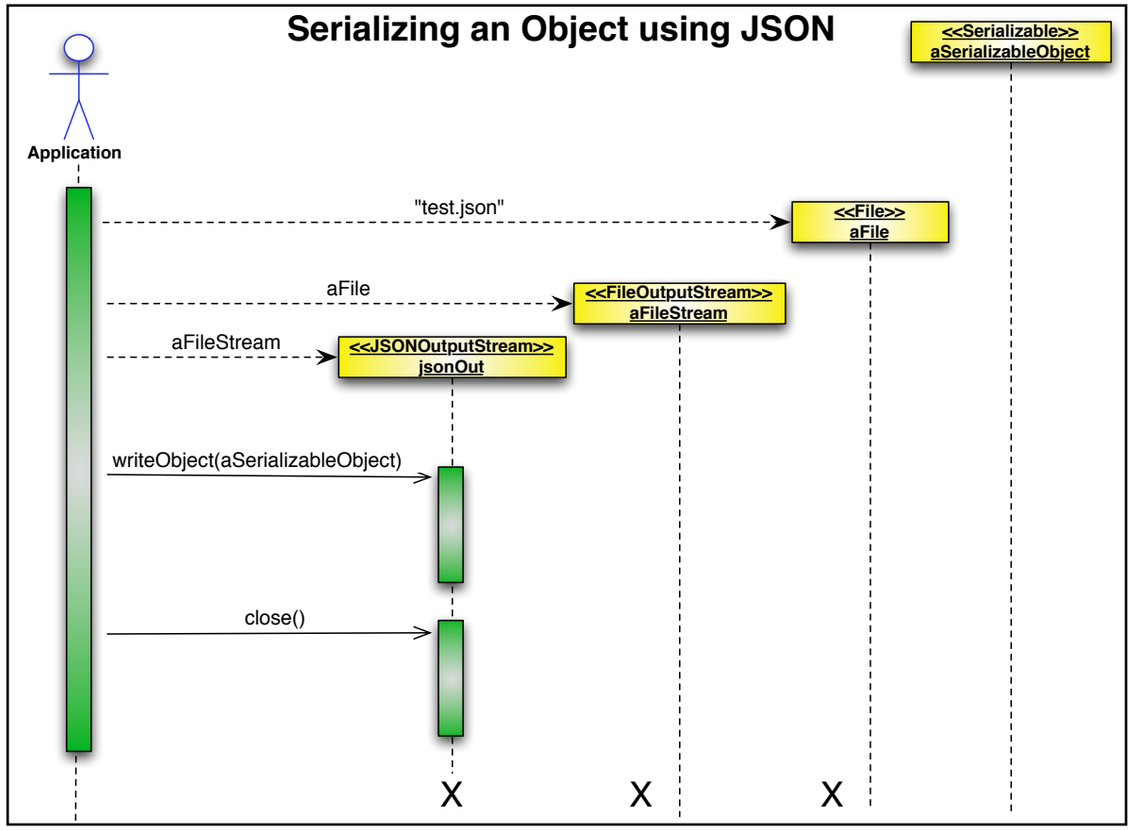


Diagram 2: a partial UML Sequence Diagram showing JSON IO

As with all UML Sequence diagrams time is shown flowing from top to bottom, elements that already exist, like the App and the aSe-

rializable 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 libraries 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 classes provided to accomplish this are the ***ObjectOutputStream*** and ***ObjectInputStream***. Just like JSON output and input streams these are used by wrapping some other stream. And just 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 Sample 5

Notice that if you use this method of serialization that the object read in can be directly cast to the type that was originally stored using the ObjectOutputStream writeObject method. This is one advantage of using the standard binary serialization streams. One 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 readable.

Another issue, at the time of the writing of this book, is that the object streams do not work in Android™ applications. There is no problem using them in standard or enterprise Java applications.

Since the data for the object is being serialized in a binary format it is not 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 save Java objects to and create them from XML. 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 could 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);
  XMLEncoder objectIn = new XMLEncoder(aFileStream);
  Person aPerson = (Person)objectIn.readObject();
  objectIn.close();
}
catch (Exception e) {
  e.printStackTrace();
}
```

Code Sample 6

Just 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 ***XMLEncoders' writeObject*** method will accept any type of Java object but both it and the ***XMLDecoders' 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 network socket streams that are needed to communicate between mobile devices or traditional computers.

Movie 3.1 An Analogy

Many Become One

Review

Review 3.1 Serialization and File IO

Question 1 of 5

JSON is

- **A.** Human readable
- **B.** Binary
- **C.** Small
- **D.** Both A & C
- **E.** Both B & C
- **F.** Both A & B

Check Answer

Suggested Further Exploration

- ☑ UML Sequence diagrams
- ☑ The XML specification
- ☑ Other Java JSON libraries
- ☑ Other Java XML libraries
- ☑ Tutorials regarding Object input and output streams
- ☑ Using Java Reader type objects

Self Check

- ☑ In what situations would you choose to use each of the types of serialization?
- ☑ Why is there more than one type of serialization?
- ☑ Why would you ever use a stream?
- ☑ Why might you wrap a stream in another stream?

CHAPTER 4

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 is done in many ways. Currently the most common uses the HTTP protocol. Regardless of the protocol used, sockets are used for the actual communication.

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 when they all connect to the same port?

Streams move the components needed for life.

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. Your browser is a client for the HTTP server that you point it at. Chat and video conferencing are often implemented as peer-to-peer. Regardless of the type of application they have at their root the same basic parts. These are 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 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 complete apache HTTPClient to the standard Java Socket class that the HTTPClient and all other options depend upon. 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 is HTTPClient 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.

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 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 object. Instead it uses integers to represent the 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 object 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 Sample 1 shows a client socket attempting to connect to a server using port 9292.

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

Code Sample 1

Code Sample 2 shows how to create a server socket that will wait for incoming connections on port 9292.

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

Code Sample 2

These two samples 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 used as the client and server communicate with each other after the initial connection is hidden from the programmer. They just don't need to know.

Streams and Sockets

Diagram 1 is a visual interpretation of the behavior of Sockets, ports, and streams.

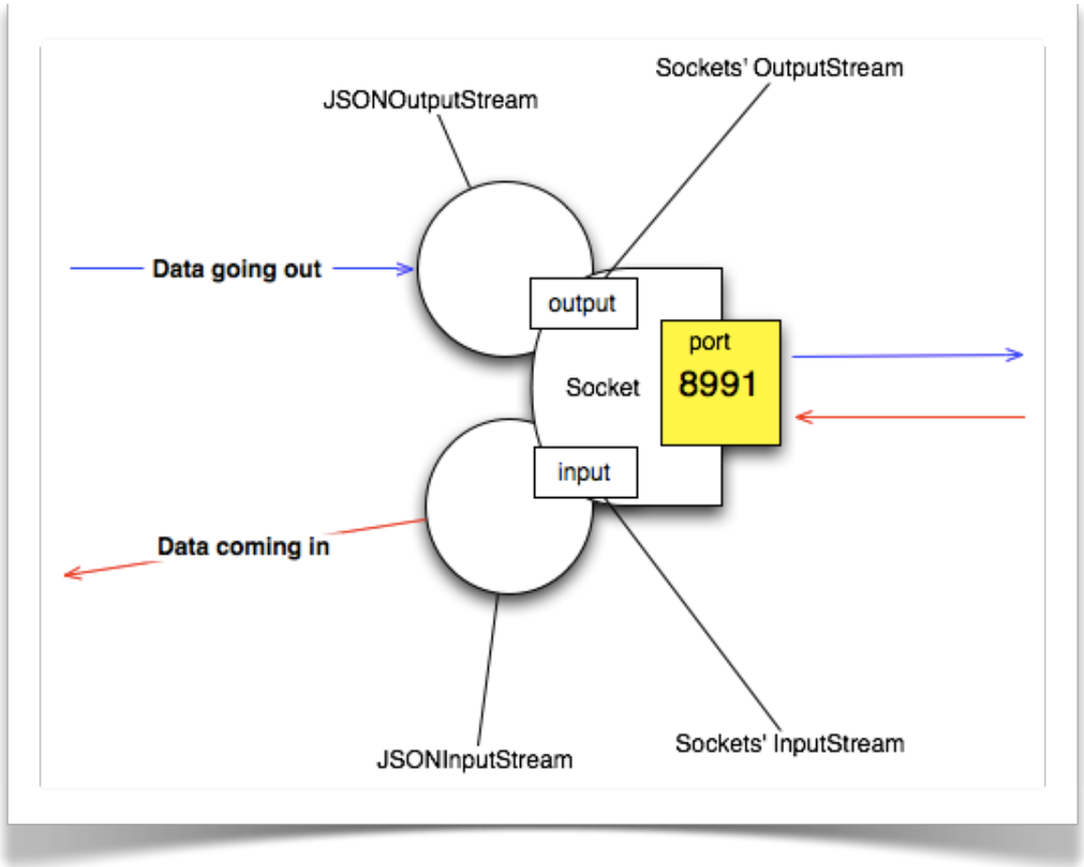


Diagram 1: Interactions of ports, sockets, and streams

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.

Usually, as seen in Diagram 1, they are 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***. 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*** using the ***writeObject*** method. The data is passed by the ***JSONOutputStream*** to the Socket's ***outputStream*** which sends the data out through the port. Data is pulled into the application using the ***JSONInputStream's readObject*** method.

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

This relationship holds true for both the Client and the Server. In both you use ***JSONInputStreams***, ***JSONOutputStreams***, ***Sockets***, and ports. Code Sample 3 shows JSON streams being wrapped around the input and output streams of the ***toServer*** socket shown in Code Sample 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 Sample 3

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

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

Code Sample 4

A Client and Server Pair

Insects can carry pollen from one flower to another.

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

Since the client is going to be an Android™ application there needs to be an Android™ GUI created. Android™ user interfaces are defined using XML. Code Sample 5 shows the code for a simple set of UI objects later used in the Android™ Java code examples.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
    <ScrollView android:id="@+id/ScrollView01"
    android:layout_width="fill_parent"
    android:layout_height="110dp">

    <TextView android:id="@+id/response"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Responses \nDisplayed \nHere." />
  </ScrollView>
  <EditText android:id="@+id/message"
  android:layout_width="300sp"
  android:layout_height="40sp"
  android:text="Enter your message here"
  android:singleLine="True"/>
  <Button android:id="@+id/sendButton"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Send Message" />
</LinearLayout>
```

Code Sample 5

simple UI XML as file

This layout for the single 'user view' we will be creating contains a scrollable region, ScrollView, for viewing results. This scrollable region contains a ***TextView*** where the actual textual results are displayed. Next there is an ***EditText*** region used to enter textual data and lastly a ***Button*** used to trigger communication with the server.

There are many other types of Android™ UI Elements but they are outside the scope of this book. There are many books, tutorials, and web sites that can help you discover other elements you may need.

Code Sample 6a contains the client-side socket code seen in previous samples being used within an ***Activity***. Android™ activities consist of the user interface declaration XML file, of which Code Sample 5 is an example, and the Android™ Java code that is used by that portion of the interface.

It is very common for an Android™ application to have several or even many instances of the ***Activity*** class. For this simple client socket example there is only one.

```
public class SimpleSocketExample extends Activity {
   int numMessagesSent = 0;


 /** Called when the activity is first created. */
 @Override
 public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.main);
   //get the UI items to interact with
   final TextView responseView =
   (TextView)this.findViewById(R.id.response);
   final EditText textInput =
   (EditText)this.findViewById(R.id.message);
   try{
   //connect to the server
       Socket toServer = new Socket("10.0.2.2", 9292);
       //setup the JSON streams to be used later.
       final JSONInputStream inFromServer =
     new JSONInputStream(toServer.getInputStream());
       final JSONOutputStream outToServer =
     new JSONOutputStream(toServer.getOutputStream());
```

Code Sample 6a

Notice, in Code Sample 6a, that the UI elements declared in the xml file need to be 'found' in order to use them. This is due to the XML being interpreted by the Android™ runtime and correspond-

ing objects created behind the scenes. You do not need to create them yourself.

Code Sample 6b shows an anonymous ***OnClickListener*** being added to a ***Button*** instance corresponding to the ***Button*** element declared in the xml file. This ***OnClickListener*** is the code that executes when the button is touched.

SimpleSocketExample source as file

```
//add the on click listener to the button
Button sendButton =
(Button)this.findViewById(R.id.sendButton);
sendButton.setOnClickListener(new OnClickListener() {
  public void onClick(View v) {
    numMessagesSent++;
    //prepare the bean
    ArrayList aDataList = new ArrayList();
    aDataList.add(numMessagesSent);
    aDataList.add(textInput.getText().toString());
    CommunicationBean aBean =
    new CommunicationBean();
    aBean.setCommand("Speak");
    aBean.setData(aDataList);

    try {
      //send the bean
      outToServer.writeObject(aBean);
      HashMap aMap =
      (HashMap)inFromServer.readObject();

      responseView.setText(aMap.toString());
    } catch (JSONException e) {
      e.printStackTrace();
      responseView.setText("Error: Unable to trade beans with
                                server.");
    }
  }
});

  }
  catch(Exception e){
  e.printStackTrace();
  responseView.setText("Error: Unable to communicate with
                            server. "+e.getLocalizedMessage());
  }
 }
}
```

Code Sample 6b

Each time the button's ***OnClickListener*** is activated by a user touch, as seen in Code Sample 6b, a JavaBean is created containing a ***String*** and an ***ArrayList***. The ***String*** is a command for the server

to interpret and the ***ArrayList*** contains any information needed by the server to accomplish the specific command.

Once the JavaBean is instantiated and its data set, it is sent to the server using the ***JSONOutputStream writeObject*** method. The client then immediately attempts to read a response from the server using the ***JSONInputStream readObject*** method.

Code Sample 7 is the code for a server application that can communicate with the client from Code Sample 6. It is written in standard Java SE and contains the ServerSocket discussed in the Sockets section of this chapter.

SimpleExampleServer source as file.

```
public class SimpleExampleServer {

    public static void main(String[] args) {

          try {
                //a socket opened on the specified port
                ServerSocket aListeningSocket =
                            new ServerSocket(9292);
                while(true){
                      //wait for a connection
                      System.out.println("Waiting for client connec-
tion request.");
                      Socket clientSocket =
                            aListeningSocket.accept();
                      //setup the JSON streams for later use.
                      JSONInputStream inFromClient =
         new JSONInputStream(clientSocket.getInputStream());
                      JSONOutputStream outToClient =
        new JSONOutputStream(clientSocket.getOutputStream());
                      //read until the client closes
                      //the connection.
                      while(true){
                            System.out.println("Waiting for a mes-
sage from the server.");
                            HashMap aMap =
                       (HashMap)inFromClient.readObject();
                            System.out.println("Just got:"
                        +aMap+" from client");
                            CommunicationBean aResponse =
                            new CommunicationBean("Done",
                      (ArrayList)aMap.get("data"));
                            outToClient.writeObject(aResponse);
                       }
                }
          } catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
                System.exit(1);
          }
    }
}
```

Code Sample 7

Since the example is a server it is assumed that it should continue running even if no clients are connected or an existing client closes its connection to the server. To accomplish this an outer infinite loop, ***while(true),*** is used to make sure the server keeps listening for new connections. There are other, more sophisticated, ways of keeping the server running but the infinite loop method works find for this simple example.

An inner infinite loop is used to ensure multiple communications with a connected client are possible. This inner loop uses the ***JSONInputStream*** to do a read from the client and a ***JSONOutputStream*** to do a write to the client. A JavaBean is written back out that contains the command 'Done' and echoes the data originally sent back to the client.

The server could have done some computation, storage, or forwarding but this is just a simple example of socket communication. It is not an example of what to do with the data once the server has it.

Client-Server Pair Types

A bunch of client-server pair types exist. For simplicity's sake we will lump them all into two categories. These two types cover the cases where the client connects to the server each time it needs or wants to send data, *connect-many-use-once*, and where the client connects once and uses the connection until the client exits, *connect-once-use-many*.

Connect-Many-Use-Once

Connect-many-use-once client server pairs are the type most people have experience with. Your web browser and the web servers providing access to web apps (often these servers are versions of Apache) are specific examples. Each time you click a link your web browser client creates a socket and connects to the server. After the connection is made a GET, POST, or other request is made, data is exchanged, and the socket connection is closed.

This type of client-server activity is commonly used in situations where the possibility exists for very large numbers of clients attempting to use the server at the same time. Since the number of server ports is limited (see the previous section) the client gives up the connection to allow another client to get a connection. This is why more clients than port numbers can appear to all be using the server at the same time. They aren't really. It just seems like they do because they are disconnecting.

A drawback to connect-many-use-once pairs is they use a larger proportion of computational cycles and time compared to connect-once-use-may pairs. Creating and destroying sockets and connections is computationally expensive. Each time a connection is needed a socket is generated and a connection is established on both ends. This requires CPU cycles and memory allocation on the server. Since your server is the limited resource this may be a bad idea in some situations.

Connect-Once-Use-Many

Connect-once-use-many client server pairs are different. They overcome the CPU cycle problem by only connecting once. The client

usually does this when it starts up and the connection is usually closed when the client exits. In between connecting and disconnecting, multiple requests send or get information from the server. If you have done much work with MySQL databases you have used a client server pair that acts like this. The MySQL CLI (Command Line Interface) and the MySQL server are a connect-once-use-many pair.

To use them you first "login." This is when the CLI connects to the MySQL server. After logging in you have an opportunity to execute many SQL statements, make requests for or send data, and then you close the connection by "logging out."

The drawback of connect-once-use-many pairs is the number of concurrent users is limited. There are only so many port numbers available to the database server and they can quickly be used up. The SimpleExampleClient and SimpleExampleServer are an easier to understand connect-once-use-many pair.

Do not try to create a client of one type, a server of the other and then use them together. If you do, your client and/or server will throw exceptions regularly. Exception creation and handling are computationally expensive. Use them only when necessary.

Review

Review 4.1 Clients and Servers

Question 1 of 5

A to a remote device requires three components. Which of these is NOT one?

- **A.** Socket
- **B.** Port
- **C.** MAC address
- **D.** IP address

Check Answer

Suggested Further Exploration

- Other types of Java streams that can be used for wrapping socket streams
- UML Use Cases and Use Case diagrams
- UML State diagrams
- System level testing
- Java data encryption support classes

Self Check

- Why might you use a JavaBean to send data back and forth between a client and a server application pair?
- Why might you choose to use the raw Socket streams?
- Why might you choose not to use the raw Socket streams?
- Why can multiple clients all contact a single server using the same port?

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 inherit from 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. Less 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 are used with ThreadPools and 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 for reuse later.

Inheriting from Thread

Code Sample 1 contains an example of inheriting from thread.

Code Sample 1 source as file

```
public class SillyThreadExample 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++) {
     SillyThreadExample aSillyThread =
     new SillyThreadExample();
     aSillyThread.start();
   }
    }
}
```

Code Sample 1

Each of your objects that inherits from ***Thread*** must override the ***public void run method***. The code in this method is what will execute when your custom thread object is started.

To start a thread running in parallel you must instantiate the custom thread and call the ***public void start()*** method it inherited from the base Thread class. You can see this in the ***main*** method of the ***SillyThreadExample*** class.

Using Runnables with Threads

Code Samples 2 and 3 contain code that does exactly the same thing as Code Sample 1 but using a custom ***Runnable*** instead of a

custom *Thread*.

```
public class SimpleRunnableStarter{

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

Code Sample 2

Code Sample 2 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.

```
public class SimpleRunnableExample 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 3

Code Sample 3 source as file

Using Executors, Thread Pools, and Runnables

Code Sample 4 uses the same ***SillyRunnable*** class seen in Code Sample 3 but in a different way. Here we create and use an ***Executor*** that contains a ***CachedTreadPool***. This means that each time the ***execute*** method of the ***Executor*** is called it runs a ***SillyRunnable*** in an available thread from the pool. A thread is available if it is not currently running a ***Runnable***.

If the ***Executor*** 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 has been available in the pool for 60 seconds it will be removed from the pool and deleted.

```
public class SillyThreadPoolExample {

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

Code Sample 4

Code Sample 4 as file

By dynamically adding and removing threads from the pool the ***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, default 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 class that inherits from ***Async-***

Task as an inner-class within the Android™ *Activity* which uses it. In normally or highly complex applications the other options are better options.

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 ***WeakReference***s. Many examples exist online showing the use of these classes. Look there for more understanding.

Multiple Clients == 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 a request from one server at a time. The same is true of PHP, ASP, JSP, Ruby, Python, Perl, or other servers. If they had to complete one request before starting another it would be a terrible waste.

All of these modern servers, regardless of the language being used, 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. This usually means that each time the ***ServerSocket*** listen method returns a ***Socket*** a thread is spawned to handle communication with the client.

Clients Need Multiple Threads

One problem with the client in Chapter 3 is that the ***writeObject*** and ***readObject*** JSONStream calls are locked in a specific order relationship. With that code if you want data from the server you must first send data to the server. Sometimes this is fine but in many applications the receiving of data from the server must be independent from the sending of data to the server.

In such situations it is a much better design to have one thread handle sending data to the server and another reading data. Using something like the ***CommandBean*** seen in the Client-Server application now becomes a very good design since the server can send a command to the client indicating what needs to be done and include the data that needs processing, displaying, etc. The client can do the same when it sends data to the server.

If such a design is used, processing a response to a request is just as easy as it was before, using the lockstep write/read design, but other types of processing (server originated, client originated without response, etc.) are easily handled as well.

Review

Review 5.1 Parallel Processing

Question 1 of 5

The Runnable interface has what method?

- **A.** go
- **B.** start
- **C.** execute
- ✓ **D.** run

Check Answer

Suggested Further Exploration

- ☑ Thread race conditions
- ☑ Thread deadlocks
- ☑ Synchronization
- ☑ Semaphores

Self Check

- ☑ Why might you or might you not need threads in your applications?
- ☑ Why might you choose to use custom Runnables?
- ☑ Why might you choose to use custom Threads?
- ☑ Why do all modern servers include parallel processing?

CHAPTER 6

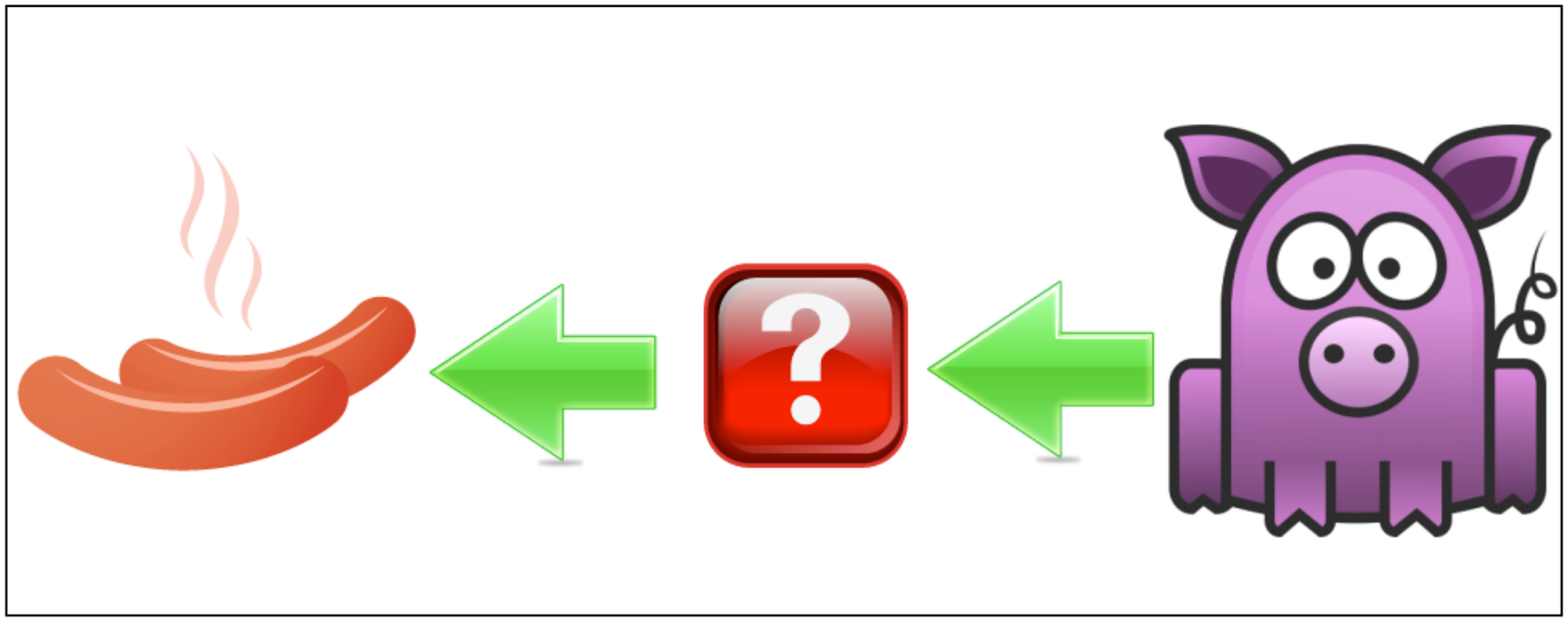
Small, Efficient, Flexible

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 to find out how to do a specific task in an applications' code. Often when we are looking for help 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 at-

tempting. 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 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 one design that is used in this kind of a situation. 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 your situation.

The Application Controller Pattern

Diagram 1 is an a UML class diagram for an 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***.

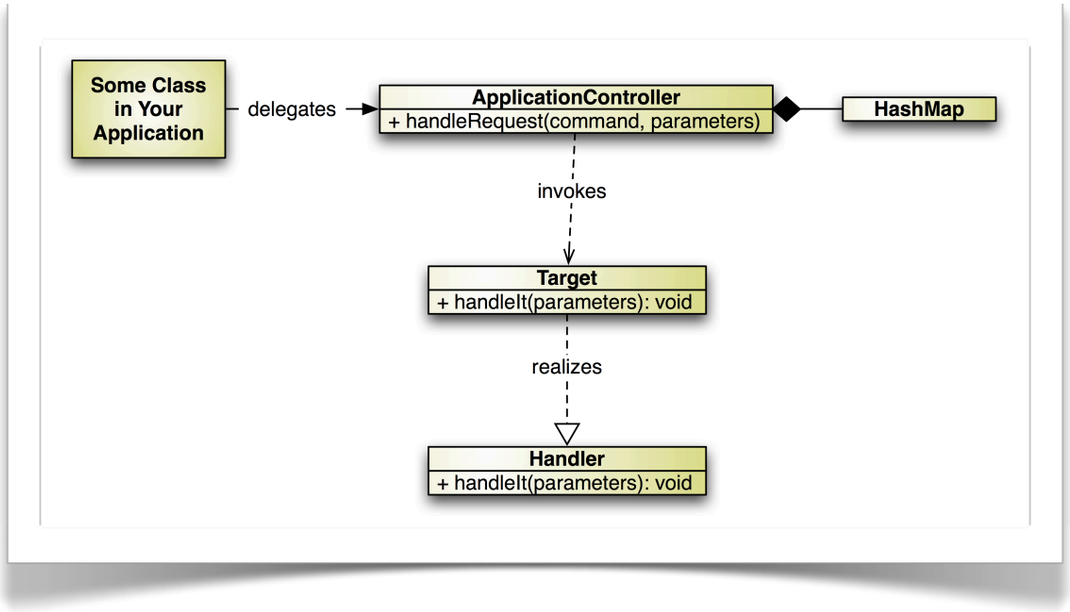


Diagram 1: the application controller pattern as a class diagram

The ***Target*** class in the diagram is a placeholder for any number of classes that implement 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 the ***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 some 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.

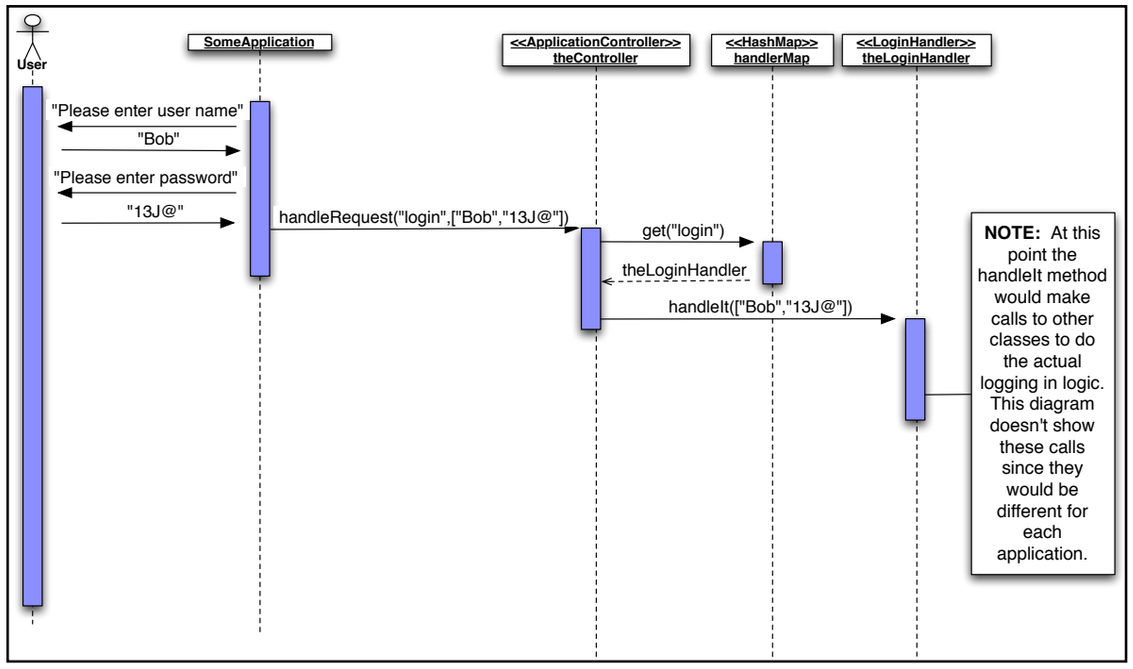


Diagram 2: a partial sequence diagram of an application controller

Diagram 2 as image

You can see the flow of the method calls in this diagram. The diagram shows that the handler specific to the ***login*** command is found in the ***HashMap*** and its ***handleIt*** method is called. If you were creating an online bank you may have a handler for "transfer funds" and many others. If your application was a calculator you could have a handler for "add" or "subtract" or some other function.

Regardless of how many handlers 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 6.1 An Analogy

Living in the Rocks

Review

Review 6.1 Small, Efficient, Flexible

Question 1 of 3

All if-else and switch statements should be replaced with and implementation of the application controller pattern.

- **A.** true
- **B.** false

Check Answer

Suggested Further Exploration

- ☑ Software design patterns

Self Check

- ☑ Why does the Application Controller Pattern work efficiently?
- ☑ Why is the Application Controller Pattern so flexible?
- ☑ Why might you choose not to use the Application Controller Pattern?
- ☑ Why might you choose to use the Application Controller Pattern?
- ☑ Why might you create your own interface and have classes that implement it?

CHAPTER 7

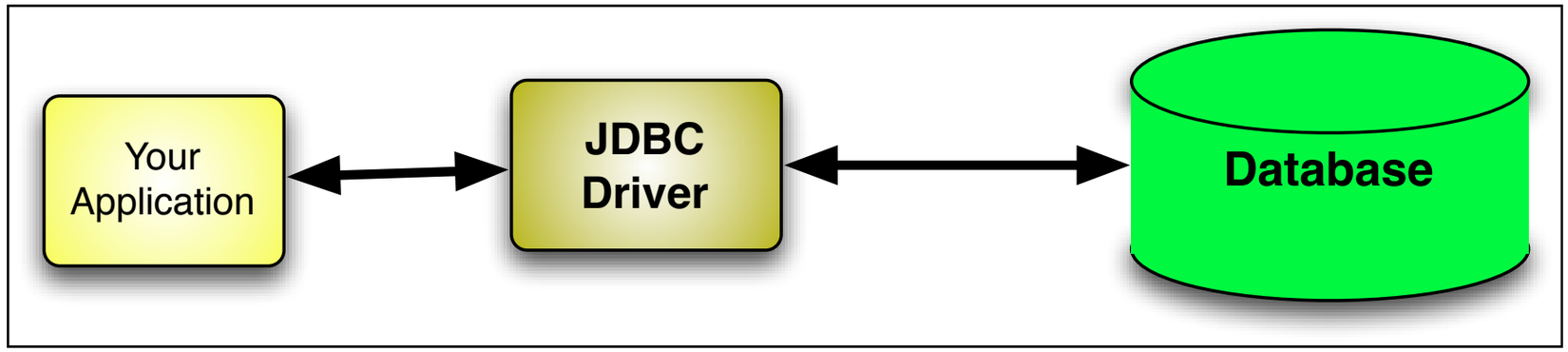
JDBC and the Model

Most applications require that data be persisted in some fashion. This may mean writing a file to disk, sending it to a web service to be persisted, or putting the data in a database. Most business applications persist data in databases. This lets businesses find and evaluate data much easier. They usually use relational databases for this persistence. JDBC is how this is done in Java.

JDBC

Points to Ponder

1. Why might your application choose to use a sophisticated database?
2. Why might you choose not to use a database in your application?
3. Why might you choose to use a prepared statement even though they are computationally intensive?



The Java Database Connectivity (JDBC) driver allows your application to talk to a relational database.

JDBC, Java DataBase Connectivity, is a standard defined in 1997 and modified in later years. Each database vendor produces its a JDBC driver for its database that matches this standard. Your application can use one or more database types. If you choose to use MySQL you will need to download the JDBC Driver for MySQL, Connector/J. For Oracle you would download JDBC driver for the database version you would be using. Many other databases also have JDBC drivers.

Anytime a Java application is storing and retrieving data from a relational database it is done using the JDBC API.

Which database you choose for your application will depend on the situation in which it must run. If the amount of data is large and the number of connections is high this would push you towards using an Oracle or IBM DB2 database. If cost is an issue then MySQL or even SQLite may become more of an option. Regardless of which

database you select the JDBC driver for it will contain the same classes and the same steps for connecting to and communicating with the database.

Connecting to a Database

To connect follow these steps:

1. Load the driver class
2. Create a database connection
3. Make one or more calls to the database
4. Close the database connection

Step 4 is very important. If you forget to close the connection to the database it will not be closed when the Connection object goes out of scope nor will it be closed when your application exits. The database will maintain the connection resource on its side until a specified timeout time has passed.

Connections to a database are a limited resource. There are not an infinite number available. If you fail to close the connections appropriately the database will run out of available connections and nothing will be able to connect. Do not forget to close the connection!

Do not open and close connections indiscriminately. The creation of database connections is a heavy user of time and computer resources. Open connections when needed and then keep them around as long as needed. A partial list of questions designed to help you to decide when to open and close database connections includes:

- Is the database running out of available connections?
- Is the application going to make more database requests?
- Is the database or the application using too many CPU cycles?

Statements and Prepared Statements

There are many kinds of SQL statements. They are used to create new database tables, drop existing tables, insert data into tables, modify data in tables, in addition to other items. This book is not a primer on SQL. There are many good SQL examples and tutorials online. An example of a simple select statement used in a login type situation could look something like this.

SELECT * FROM user WHERE name = 'bob' AND pword = '13J@';

This select statement requests the values found in all of the fields in a record where the ***name*** field has the value ***bob*** and the ***pword*** field has the value ***13j@***. It is not uncommon to find a situation where the SQL statement is assembled as a Java String. Code Sample 1 shows an example of how the SQL above might be created.

```
String sqlString = "SELECT * FROM user WHERE uname = '"
      +aName+"' AND pword = '"
      +aPassword+"'";
```

Code Sample 1

Such an assembled SQL string can be executed in the database using the ***executeUpdate*** method of a JDBC ***Statement*** object. You

should NEVER assemble such an SQL statement. If you do, a hacker will find out that you have. This hacker will then execute what is called an SQL injection attack and your database and application will be compromised.

Never assemble SQL even if you think you will come back and replace it later. Sooner or later you will forget to replace it and then your career may be over. No one wants to hire an engineer or programmer that cost their last company millions of dollars in lost money due to fraud or paying for remediating identity theft and law suits.

If you are not assembling an SQL statement then using the ***executeUpdate*** method is very valid. Hardcoded SQL statements can not be exploited by hackers.

The security problem just identified is easily resolved if you use a JDBC ***PreparedStatement*** object. Code Sample 2 shows a SQL string that is used with the ***prepareStatement*** method of the JDBC ***Connection*** class.

```
String sqlString = "SELECT * FROM user WHERE uname=? AND pword=?";
```

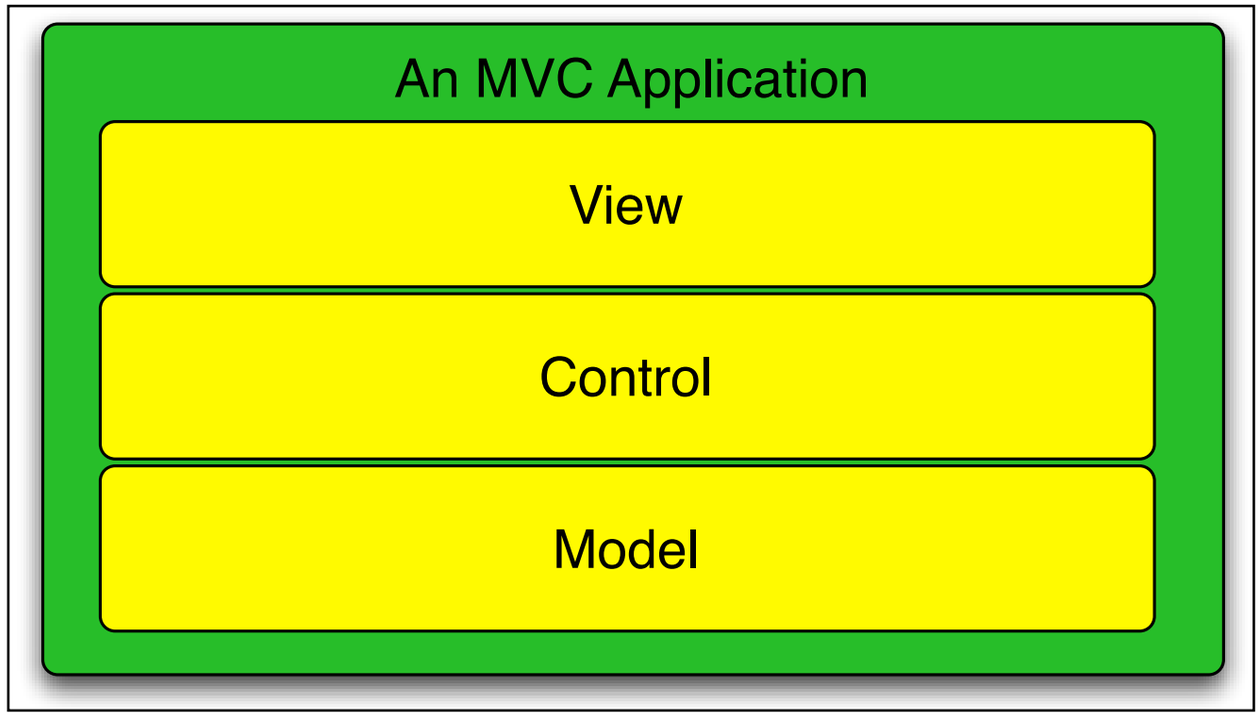
Code Sample 2

Notice that the SQL includes question mark (?) characters that are place holders for values that will be defined later. The ***prepareStatement*** method parses this SQL prior to the addition of the values. This means that regardless of the values that replace the ? characters later, they will never be parsed as SQL. Because of this one feature, SQL injection attacks never succeed if you have used prepared statements. Use prepared statements.

Modularity and the Model

Lorem Ipsum

1. If your application uses a good modular design, why can you change the internal behavior of the application's model portion without effecting any other parts?



Model-View-Control (MVC) applications employ modularity

The model in a Model-View-Control (MVC) application is the data the application uses or a representation of the that data. In applications that use databases the data is actually managed by the database. The application then usually has some sort of internal representation of the data. This representation could be JavaBeans, ArrayLists, HashMaps or some other sort of collection, or some combination of these. The VisitModel created in response to Case 1 is an example of a MVC model.

If the model in an application is designed correctly then its internal behavior is hidden from the rest of the application. This is the standard

Object Oriented Design principle called encapsulation. Diagram 1 is a graphical representation of encapsulation.

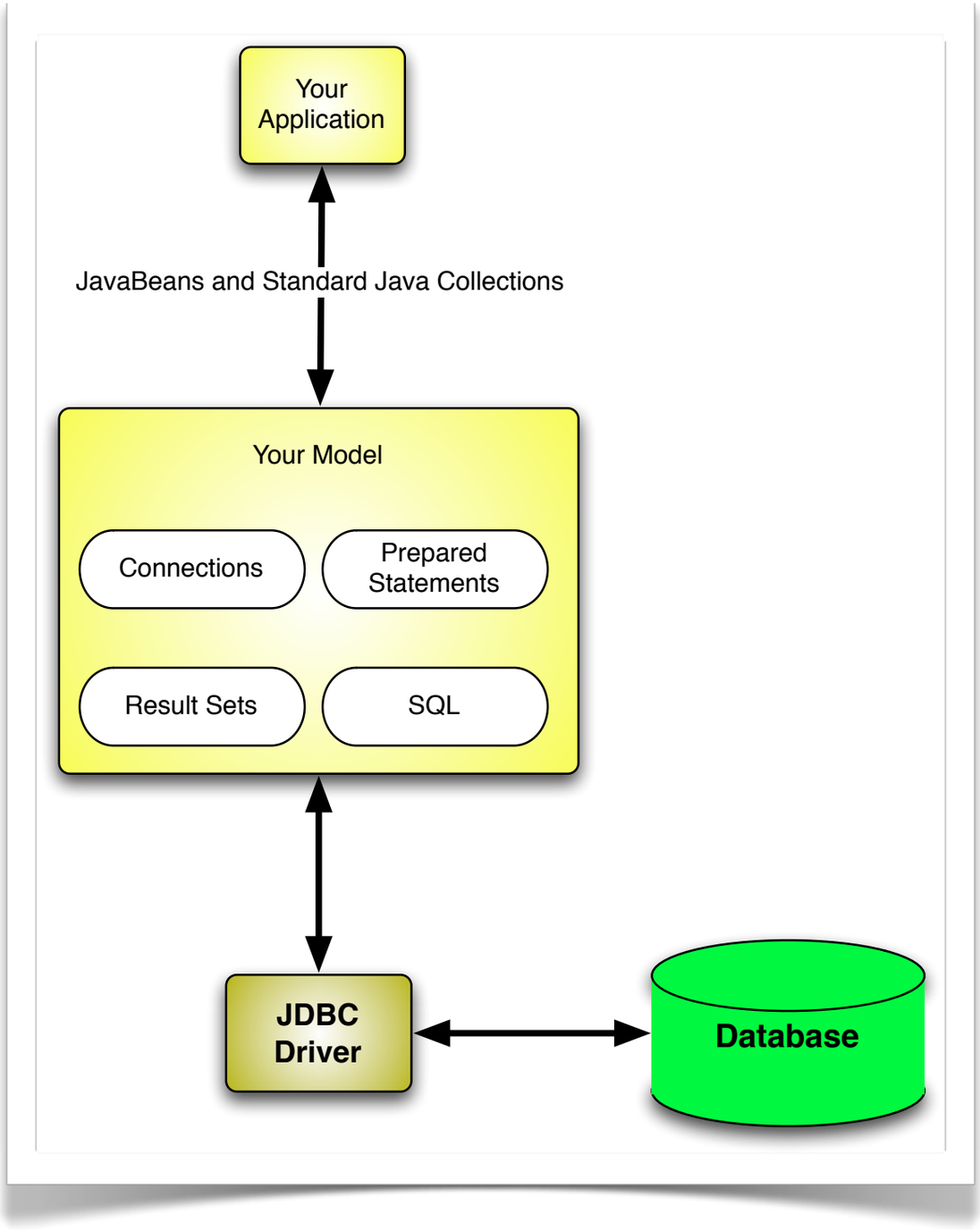
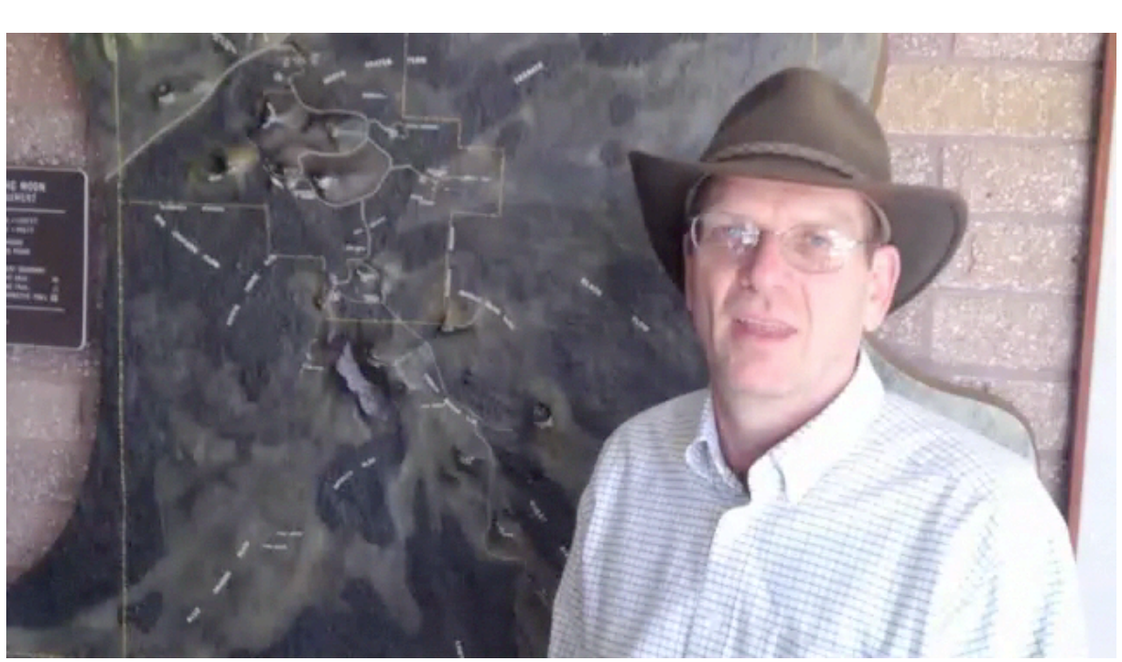


Diagram 1

The application seen in Diagram 1 has no idea if the model does or does not use a database. All information regarding the connection string, the connections themselves, the SQL, and any other items used to interact with the database such as the JDBC ResultSet object that is the result of querying the database for information, must not be exposed or returned from the model.

The model must encapsulate and hide any behavior that would indicate if a database, JSON file, text file, spreadsheet, or XML file was being used to persist the data. If this is not hidden from the application the model is not a model and rules for modularity have been violated.

Movie 7.1 An Analogy



Models are not the same as reality. They represent reality.

When modularity rules are violated the software industry refers to the result as spaghetti code. Spaghetti code is always a problem since fixing a defect in one part of the application can easily create one or more defects in other parts of the application. This is due to not hiding how things are done in your model or other modular portions of the application. The software becomes a tangled mess, hence the name spaghetti code.

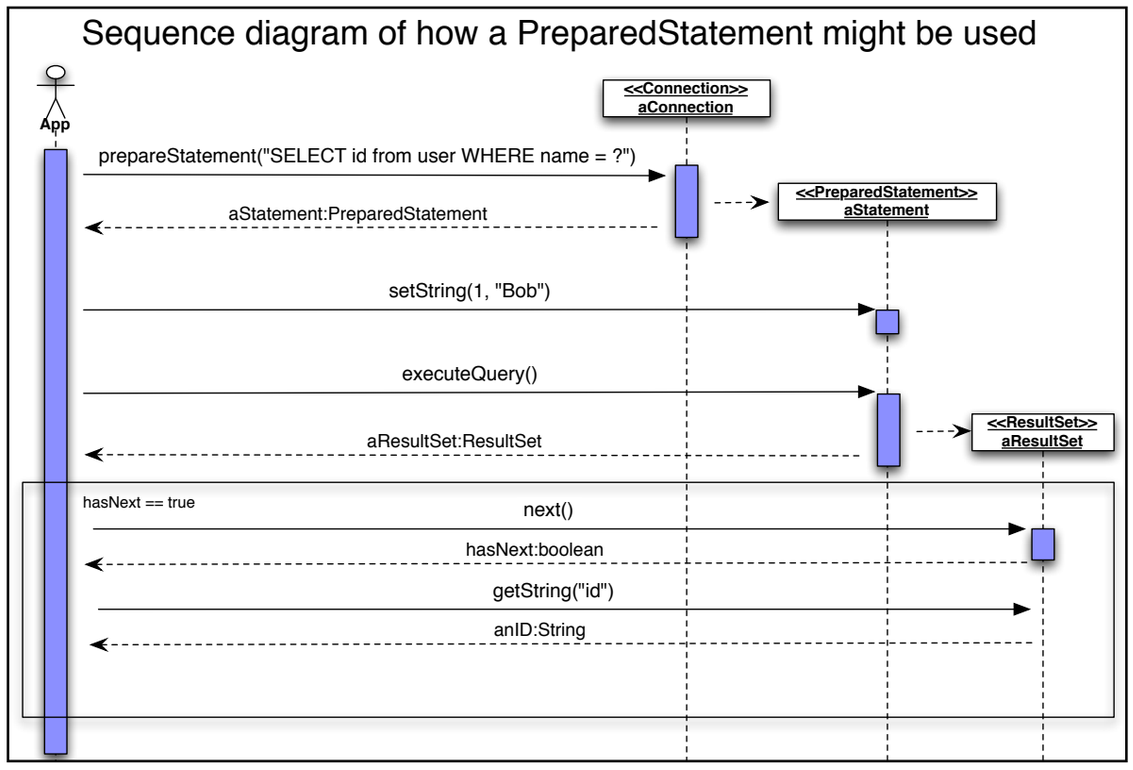


Diagram 2

Diagram 2 as image

Review

Review 7.1 JDBC and the Model

Question 1 of 3

A model should return a JDBC ResultSet object when a query is executed.

- **A.** true
- **B.** false

Check Answer

Suggested Further Exploration

- ☑ SQL insert statements
- ☑ SQL select statements
- ☑ The JDBC Connection API
- ☑ The JDBC Statement API
- ☑ The JDBC PreparedStatement API
- ☑ The MySQL JDBC Connection URL
- ☑ Installing MySQL
- ☑ Installing Oracle
- ☑ Designing and creating database tables and schemas

Self Check

- ☑ Why might you choose to use a standard JDBC Statement objects?
- ☑ Why might you choose to use a standard JDBC PreparedStatement object?

- ☑ Why should your model not return anything specific to using databases?
- ☑ Why should your model not accept any information from the rest of your application that is specific to using databases?

CHAPTER 8

Hibernate

One issue with using JDBC in an application's model is referred to as data impedance. Data impedance means that the data must be converted from database records to objects for each query and from objects to SQL for each insertion. This can be bug prone and slow data transfer. Hibernate helps get the data from your database to your app and from your app to your database.

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.

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 objects created from the 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.

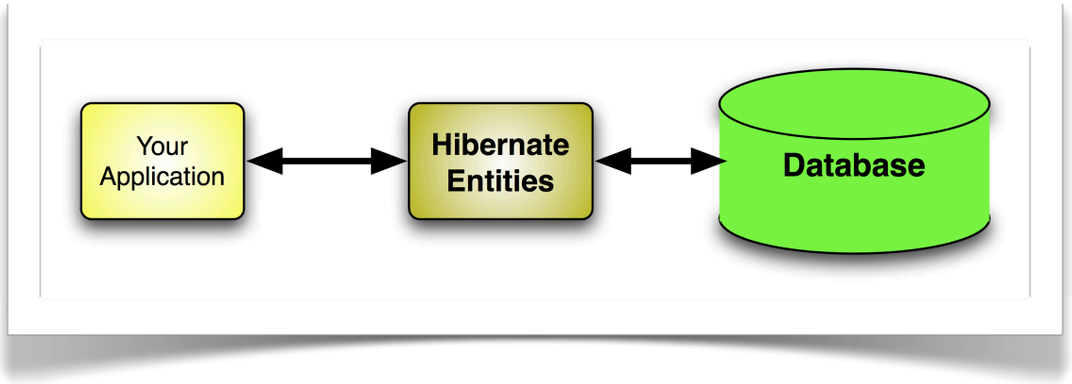


Diagram 1

Hibernate uses normal Object relationships rather than SQL joins for one and many to many relationships. A to many relationship is modeled as a class attribute of type ***Set***. Each attribute has a an accessor and a mutator method. When the getter is called the Hibernate libraries take over and execute the correct prepared statement query against the database.

The 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***, and ***pword*** fields. The ***phone_number*** table has ***id*** and ***phone*** fields. The special ***user_number*** table has ***user_id*** and ***phone_id*** fields.

The ***id*** fields 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.

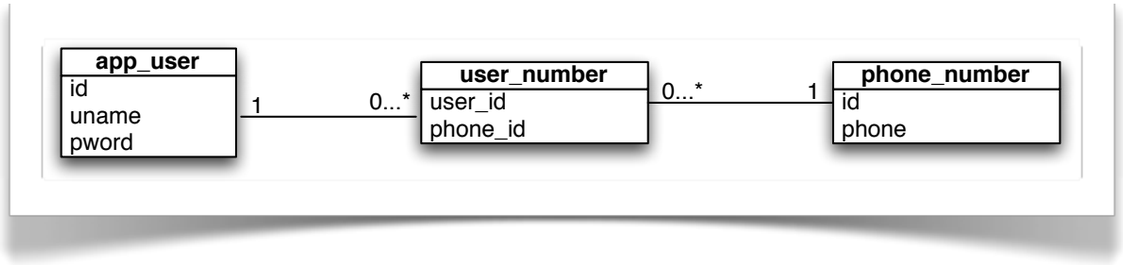


Diagram 2

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. Id 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 well, the special third table, ***user_number***, is needed. This table is referred to as a transition, junc-

tion, 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. As such they have private attributes. A getter and setter method for each attribute is also included. So far it sounds just like a JavaBean.

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 Eclipse™.

Multiple of these annotations can be listed prior to a single line of Java code and they will all act on that single line. Code Sample 1 shows how Plain Old Java Object (POJO) can be changed into a Hibernate entity.

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

Code Sample 1

Sample 1 illustrates how to use the ***@Entity*** annotation to declare the ***User*** class to be a Hibernate entity. The ***@Table*** annotation also acts on the ***User*** class since the declaration of the class is the next line of Java code.

The ***@Table*** annotation states that the ***User*** class represents the ***app_user*** table. This means that instances of ***User*** represent individual records in the ***app_user*** table.

Attributes in a Hibernate entity represent fields in the respective table. Code Sample 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 so.

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

  @Id
  @GeneratedValue
  private Integer id;
  private String uname;
  private String pword;
```

Code Sample 2

Notice another pair of annotations are declared right before the private ***id*** attribute. Since the ***id*** attribute is the first line of Java code after the annotations they both act on the ***id***.

The ***@Id*** annotation indicates that the ***id*** attribute is the primary key in the table. The ***@GeneratedValue*** annotation indicates that the ***id*** attribute will get set to a value generated by the database. ***uname*** and ***pword*** are also fields in the database.

There is yet one more attribute of the ***User*** class. As stated earlier each user can have multiple phone numbers and each phone num-

ber can belong to multiple users. In Java we represent this as a ***Set*** of ***PhoneNumber*** objects. You can see the declaration of this attribute in Code Sample 3.

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

  @Id
  @GeneratedValue
  private Integer id;
  private String uname;
  private String pword;

  @ManyToMany(cascade=CascadeType.ALL)
  @JoinTable(
       name="user_number",
       joinColumns = { @JoinColumn( name="user_id") },
       inverseJoinColumns = @JoinColumn( name="phone_id")
  private Set<PhoneNumber> phoneNumbers;
```

Code Sample 3

The annotations prior to the ***phoneNumbers*** declaration tell the code how to access any 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 compiler that it should use the ***user_number*** table to find any phone numbers. It also describes the fields found in the ***user_number*** table. These fields are described by the annotations as joinColumns, ***user_id***, meaning that the ***user_id*** field in the records 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 this is confusing it is understandable if you have no database design experience. Hang in there and hopefully this will all come together when you run the example code.

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.

User class source as file

The ***PhoneNumber*** class is much simpler and found in Code Sample 4.

```
@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 4

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_number*** table. It is not needed.

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

PhoneNumber class source as file

Movie 8.1 An Analogy

Aunts as templates and children as proxies

Using Hibernate Entities

Now it is finally time to instantiate some objects, store them in the database, modify them, and delete them from the database. An example application called SillyHibernateUseExample has been created and is discussed here.

SillyHibernateUseExample class source as file

Code Sample 5 shows two instances of the User class being instantiated and then saved into the database. Notice that a transaction is begun before and committed after the Users are created and saved. Saving Hibernate entities doesn't commit them in a complete fashion. It notifies the database to do a provisional percisting. The ***commit*** method of the transaction forces the database to permanently persist them.

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

```
/*
 * show how to add records to the database
 */
private void addNewUsers() {
  Session session =
 HibernateUtilSingleton.getSessionFactory().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: "
                                      + aUser.getId());
  System.out.println("anotherUser generated ID is: "
                          + anotherUser.getId());
}
```

Code Sample 5

The ***HibernateUtilSingleton*** class that you see at the beginning of the ***addnewUsers*** method is a helper class that can be downloaded as the [HibernateUtilSingleton.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 and you may be able to use it in your application.

Having now seen how to create and add Hibernate entities to the database you need to know how to query them. Code Sample 6 contains the ***SillyHibernateUseExample*** 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 somewhat like SQL but is in fact Hibernate Query Language (HQL). HQL is based on the classes declared earlier to be Hibernate entities. This is why we see ***User*** declared rather than the ***app_user*** table. The string listed says, "Get me all of the User objects and order them by their id".

Once the query has been created it is executed in this method 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 the example only. You may want to 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 =
 HibernateUtilSingleton.getSessionFactory().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 Sample 6

Modifying data in the database is done in much the same way as getting data from the database. In this example only one object is to be changed so the 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 =
 HibernateUtilSingleton.getSessionFactory().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 database that
   * the instance is ready to be permanently stored.
   */
  session.merge(leeUser);

  /*
   * call the transaction commit method. This tells the
   * database that the change should be permanently stored.
   */
  transaction.commit();
 }
```

Code Sample 7

Instead of using the ***Query*** class' ***list*** method the ***uniqueResult*** method is used. This guarantees that only one ***User*** object is returned.

The ***User*** instance, ***leeUser***, is then modified by changing the ***userName***. Instead of calling ***save*** like we did when new ***Users*** were created the session's ***merge*** method is used. This causes the existing

record in the database to be updated rather then a new record being created.

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 Code Sample 8.

In this example two unique users are retrieved from the database and a new PhoneNumber is instantiated. The PhoneNumber is then added to ***Sets*** of both 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 =
   HibernateUtilSingleton.getSessionFactory().getCurrentSession();
    Transaction transaction = session.beginTransaction();
    /*
     * create a PhoneNumber instance
     */
    PhoneNumber sharedPhoneNumber = new PhoneNumber();
    sharedPhoneNumber.setPhone("(546)222-9898 ");


    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();
    /*
    * 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);

    session.save(sharedPhoneNumber);

    session.merge(joshuaUser);
    session.merge(aNameUser);

    transaction.commit();
}
```

Code Sample 8

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

```
private void deleteAddedUsers() {
  // TODO Auto-generated method stub
  Session session =
  HibernateUtilSingleton.getSessionFactory().getCurrentSession();

  Transaction transaction = session.beginTransaction();
  int numUsers = users.size();
  for(int i = 0; i < numUsers; i++){
    System.out.println("deleting user "
                              +users.get(i).getUname());
    User aUser = users.get(i);
    session.delete(users.get(i));
  }
  transaction.commit();
}
```

Code Sample 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 samples or, as in this example, the references were previously stored in an ***ArrayList***. Either way, the session is retrieved, the transaction started and then the session's ***delete*** method is called. Once ***transaction.commit()*** is executed the matching record is removed permanently from the database.

Hibernate entities may seem daunting at this point but as you use them you will begin to see a repeating pattern that saves you a great deal of time and effort. You were going to create the beans anyway so add the annotations in, 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.

Review

Review 8.1 Hibernate as a Model

Question 1 of 3

A model hides how data is stored and retrieved.

- **A.** true
- **B.** false

Check Answer

Suggested Further Exploration

- ☑ The Singleton Pattern
- ☑ The HibernateUtilSingleton
- ☑ The Proxy Pattern
- ☑ Object Relational Mapping (ORM)
- ☑ HQL
- ☑ Many-to-many relationships
- ☑ One-to-many relationships

Self Check

- ☑ Why might you choose to use Hibernate instead of JDBC?
- ☑ Why might you choose to use JDBC instead of Hibernate?
- ☑ Why are ORM's used in industry?

CHAPTER 9

Appendices

Hibernate Example Files

Hibernate is a powerful tool used to interact with relational database management systems using objects. [Hibernate Home]

In this appendix you will find the source code for an example test application that uses some simple Hibernate functionality. This is just example code and the source is not intended to be inserted directly into your application as working code.

You will also find an sql batch file that can be used to create the MySQL schema used by the example code.

SQL database script source

download source as file.

```
/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;

/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;

/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;

/*!40101 SET NAMES utf8 */;

/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;

/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0 */;

/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;


CREATE DATABASE IF NOT EXISTS test;


USE test;


DROP TABLE IF EXISTS test.app_user;


CREATE TABLE test.app_user (

id int(11) NOT NULL AUTO_INCREMENT,

uname char(20) NOT NULL,

pword char(20) NOT NULL,

PRIMARY KEY (id)
```

```
) ENGINE=MyISAM AUTO_INCREMENT=87 DEFAULT CHARSET=utf8;

/*!40000 ALTER TABLE app_user DISABLE KEYS */;

LOCK TABLES app_user WRITE;

INSERT INTO test.app_user VALUES (37,'sally','somePass'),
(38,'dude','hey2dude');
UNLOCK TABLES;

/*!40000 ALTER TABLE app_user ENABLE KEYS */;

DROP TABLE IF EXISTS test.phone_number;

CREATE TABLE test.phone_number (
id int(11) NOT NULL AUTO_INCREMENT,
phone varchar(20) NOT NULL,
PRIMARY KEY (id)
) ENGINE=MyISAM AUTO_INCREMENT=10 DEFAULT CHARSET=utf8;

/*!40000 ALTER TABLE phone_number DISABLE KEYS */;
```

```
LOCK TABLES phone_number WRITE;

INSERT INTO test.phone_number VALUES (1,'(120)345-6789'),
(2,'(208)321-4567'),
(3,'(201)456-0987');

UNLOCK TABLES;

/*!40000 ALTER TABLE phone_number ENABLE KEYS */;

DROP TABLE IF EXISTS test.user_number;

CREATE TABLE test.user_number (
user_id int(11) NOT NULL,
phone_id int(11) NOT NULL,
id int(11) NOT NULL AUTO_INCREMENT,
PRIMARY KEY (id)
) ENGINE=MyISAM AUTO_INCREMENT=17 DEFAULT CHARSET=utf8;

/*!40000 ALTER TABLE user_number DISABLE KEYS */;

LOCK TABLES user_number WRITE;
```

```
INSERT INTO test.user_number VALUES (37,1,1),

(38,3,7),

(38,2,8);


UNLOCK TABLES;


/*!40000 ALTER TABLE user_number ENABLE KEYS */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;

/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;

/*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;

/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;

/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;

/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;

/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
```

User.java source

download User.java source as file

```
/* User.java source code
 * The Many-To-Many relationship between User and PhoneNumber objects requires two tables
 * app_user and phone_number with a transition table user_number that maps phone numbers to users.
 * The @ManyToMany annotation shows how to make the User object aware of its phone numbers.
 */
package com.java.life.real.examples;
import java.util.List;
import java.util.Set;
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
import javax.persistence.OneToMany;
import javax.persistence.Table;
@Entity
@Table(name = "app_user")
public class User {

  @Id
  @GeneratedValue
  private Integer id;
  private String uname;
  private String pword;

  /*
   * one User can have many phone numbers. CascadeType.ALL causes associated
   * phone numbers to be delted when a User is deleted.
   */
  @ManyToMany(cascade=CascadeType.ALL)
  @JoinTable(
        name="user_number",
        joinColumns = { @JoinColumn( name="user_id") },
        inverseJoinColumns = @JoinColumn( name="phone_id")
        )
  private Set<PhoneNumber> phoneNumbers;
  public User() {
```

```
    // TODO Auto-generated constructor stub
  }

  public String toString() {
    return "User [id=" + id + ", pword=" + pword + ", uname=" + uname + ", phoneNumbers]";
  }

  public Integer getId() {
    return id;
  }
  public void setId(Integer id) {
    this.id = id;
  }
  public String getUname() {
    return uname;
  }
  public void setUname(String uname) {
    this.uname = uname;
  }
  public String getPword() {
    return pword;
  }
  public void setPword(String pword) {
    this.pword = pword;
  }
  public Set<PhoneNumber> getPhoneNumbers() {
    return phoneNumbers;
  }
}
```

PhoneNumber.java source

download PhoneNumber.java as file

```
/*
 * PhoneNumber.java source code
 */

package com.java.life.real.examples;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
@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;
  }
}
```

SillyHibernateUseExampleApplication.java source

download SillyHibernateUseExampleApplication.java source as file

```
/*
 * The purpose of this class is to show how some of the abilities of Hibernate
 * are used. Covered in this example application are; adding records to tables,
 * modifying records in tables, removing records from tables, and
 * using a Many-To-Many relationship between tables.
 */
package com.java.life.real.examples;
import java.util.Iterator;
import java.util.List;
import java.util.Set;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.Query;
import org.apache.log4j.Logger;
public class SillyHibernateUseExample {
  final static Logger logger = Logger.getLogger(ExampleApplication.class);
  private List<User> users;

  public SillyHibernateUseExample() {
    // TODO Auto-generated constructor stub
  }
  /**
   * @param args
   */
  public static void main(String[] args) {
    // TODO Auto-generated method stub
    SillyHibernateUseExample aExampleApplication = new SillyHibernateUseExample();
    aExampleApplication.addNewUsers();
    aExampleApplication.showAllUsers();
    aExampleApplication.modifyUser();
    aExampleApplication.addSharedPhoneNumber();
    aExampleApplication.deleteAddedUsers();
  }
  /*
   * show how to add records to the database
   */
  private void addNewUsers() {
    Session session = HibernateUtilSingleton.getSessionFactory().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: " + aUser.getId());
    System.out.println("anotherUser generated ID is: " + anotherUser.getId());
  }

  /*
   * show how to get a collection of type List containing all of the records in the app_user table
   */
  private void showAllUsers() {
    Session session = HibernateUtilSingleton.getSessionFactory().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();
  }

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

    Session session = HibernateUtilSingleton.getSessionFactory().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 database that the instance is ready to be perma-
nently stored.
     */
    session.merge(leeUser);

    /*
     * call the session merge method for the User instance in question. This tells the database that the instance is ready to be perma-
nently stored.
     */
    transaction.commit();
    /*
     * permanently store the changes into the database tables.
     */
    showAllUsers();
  }

  private void addSharedPhoneNumber() {
    Session session = HibernateUtilSingleton.getSessionFactory().getCurrentSession();
    Transaction transaction = session.beginTransaction();
    /*
     * create a PhoneNumber instance
     */
    PhoneNumber sharedPhoneNumber = new PhoneNumber();
    sharedPhoneNumber.setPhone("(546)222-9898");

    /*
     * get two User instances from the database using HQL. This is NOT SQL. It is object based.
     */
    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();

/*
 * 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();
/*
 * show that the database was updated by printing out all of the User instances created by a HQL query
 */
showAllUsers();
}
private void deleteAddedUsers() {
// TODO Auto-generated method stub
Session session = HibernateUtilSingleton.getSessionFactory().getCurrentSession();
Transaction transaction = session.beginTransaction();

int numUsers = users.size();
for(int i = 0; i < numUsers; i++){
  System.out.println("deleting user "+users.get(i).getUname());
  User aUser = users.get(i);
  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.remove(2);
    users.remove(2);
    /*
     * 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);
  }

}
```

HibernateUtilSingleton.java source

download HibernateUtilSingleton.java source as file

```
package com.java.life.real.examples;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;
import org.hibernate.service.ServiceRegistryBuilder;

/*
* This class follows the singleton pattern.
*/

public class HibernateUtilSingleton

{

private static final SessionFactory sessionFactory;

  static{

    Configuration config = new Configuration();

    config.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLDialect");

    config.setProperty("hibernate.connection.driver_class", "com.mysql.jdbc.Driver");

    //change the next line of code to match your MySQL url

    config.setProperty("hibernate.connection.url", "jdbc:mysql://localhost/test");

    //change the next two lines of code to match your MySQL user name and password.

    config.setProperty("hibernate.connection.username", "dude");

    config.setProperty("hibernate.connection.password", "hey2dude");

    //change the pool size to reflect how many users you expect your application to have initially

    config.setProperty("hibernate.connection.pool_size", "1");

    config.setProperty("hibernate.connection.autocommit", "true");

    config.setProperty("hibernate.cache.provider_class", "org.hibernate.cache.NoCacheProvider");
```

```
        /*
         * un-comment the next line of code if you want to be able to drop and recreate tables for your data classes listed below.
         * This is generally a bad idea for security reasons.
        */

        //config.setProperty("hibernate.hbm2ddl.auto", "create-drop");

        config.setProperty("hibernate.show_sql", "true");

        config.setProperty("hibernate.transaction.factory_class", "org.hibernate.transaction.JDBCTransactionFactory");

        config.setProperty("hibernate.current_session_context_class", "thread");

        /*
        * Add your classes here that you want to match your database tables
        * The example has a User and a PhoneNumber class.
        */

        config.addAnnotatedClass(User.class);

        config.addAnnotatedClass(PhoneNumber.class);

        ServiceRegistry serviceRegistry = new ServiceRegistryBuilder().applySettings(config.getProperties()).buildServiceRegistry();

        sessionFactory = config.buildSessionFactory(serviceRegistry);
    }

    public static SessionFactory getSessionFactory()

    {
      return sessionFactory;
    }

    //make a private default constructor so that no other HibernateUtil can be created.

    private HibernateUtilSingleton(){}

}
```